DESIGN AND EVALUATION OF A PREDICTABLE EMBEDDED PROCESSOR FOR USE IN TIME-TRIGGERED APPLICATIONS

A thesis submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy

By

Zemian Hughes

Embedded Systems Laboratory

University of Leicester

2009

Design and Evaluation of a Predictable Embedded Processor for use in Time-Triggered Applications

ABSTRACT

Embedded processors play a key role in many safety-critical applications including medical, automotive and aerospace systems. In such systems an inability to provide guarantees that the design will meet its requirements can have catastrophic consequences. To ensure that guarantees can be made, it must be possible to predict both the functional and temporal properties of the system at design time.

The trend in modern embedded system design is currently leading towards unpredictable processor architectures in order to achieve increased performance. This trend presents fundamental challenges for the designers of timing analysis tools who are finding the accuracy and safety of timing estimations produced by new tools are getting worse. The consequence of this is that it is increasingly becoming harder to provide guarantees that the system requirements will be met. The primary causal factor is put down to the developments in modern processor architecture.

This thesis attempts to address these problems with a novel, highly predictable embedded processor design for systems with a "time-triggered" (TT) system architecture.

Even with a predictable processor, a real-time operating system (RTOS) implemented in software can itself complicate the temporal predictability of the system. To address this issue a predictable hardware TT scheduler is implemented in hardware.

In order to overcome the possibility of the application programmer writing temporally unpredictable code, a set of software-based error-detection (and recovery) mechanisms is implemented as a "task guardian" to deal with issues of task overruns in TT systems. The performance and complexity of the initial software implementation leads to an innovative hardware task guardian solution.

Overall, the implication of the studies presented in this thesis provide the system developer with an effective set of software and hardware components which are shown to provide a highly-predictable platform for the execution of time-triggered tasks sets.

I dedicate this thesis to my parents

Michael & Sue Hughes

and to my brothers and sister

Kerry, Javan and Naomi Hughes

Acknowledgements

This thesis would not have been possible if it were not for the help and support of many people.

Firstly I would like to express my thanks and gratitude to my supervisor, Professor Michael J. Pont, for his guidance, help and enthusiasm throughout the past few years. I am very grateful for the support and opportunities he has provided. I am also fortunate to be able to see parts of this work having application outside of this work.

I would like to thank the Engineering and Physical Sciences Research Council (EPSRC) who provided funding throughout the first few years of this work.

I owe a great deal of thanks to both Adi Maaita and Devaraj Ayavoo for their friendship and support on numerous occasions, especially during times when I felt bit overwhelmed. This work would not have been possible if it were not for their help and nowhere near as much fun without them.

I am indebted to my many of my friends that have supported me over the years. Special thanks goes out to David Ridley, Robin Hughes, Rich Elliott-Skene and Phil Clark for their friendship and kindness that saw me through some of the darkest hours and pushed me forwards to the end. I am very grateful for all the enjoyable times together and for their determination to make sure that I went out to play.

Furthermore, I would like to thank all my colleagues from the Embedded Systems Laboratory. I have enjoyed working with you all and have many fond memories, especially the afternoon coffees, BBQ's and very wet trips to Wales.

I owe my deepest gratitude to my family for their support, encouragement and unfailing belief that I could achieve goals that would not have ordinary seemed possible. I am very thankful for the many battles and hard times that my parents, Michael and Sue Hughes, have endured throughout my upbringing, especially times when they were told that I would not make it into state education. It is an honour to be part of such a loving and caring family. It is also difficult to overstate my gratitude to my brothers Kerry, Javan and sister Naomi Hughes for their support and good times together. I have had a great time growing up with you all.

I am very thankful for my grandparents, Edith Hughes and Leslie Handford, who have made available their generous support in a number of ways. I would also like to show my gratitude to my further family who have encouraged and cared for me throughout the years.

It is a pleasure to thank the numerous teachers throughout my education that have inspired me and played a vital role in helping me to develop to where I am now. In particular I would like to thank the teachers at Eccles Hall School and the inspirational lecturers at Peterborough Regional College.

I also feel God has had a hand in this, to which I am very thankful.

Table of Contents

List of Figures

List of Tables

List of Listings

List of Publications

Hughes, Z. M. and Pont, M. J. (2008), ***"Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed"***, Transactions of the Institute of Measurement and Control.

Hughes, Z. M. and Pont, M. J. (2005), ***"Time-triggered co-operative hardware scheduler"***, patent application filed (UK), 9 September 2005.

Hughes, Z.M., Pont, M.J. and Ong, H.L.R. (2005) ***"The PH Processor: A soft embedded core for use in university research and teaching"***. In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp.224-245. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Hughes, Z.M., Pont, M.J. and Ong, H.L.R. (2005) ***"Design and evaluation of a "time-triggered" microcontroller"***. Poster presentation at DATE 2005 (PhD Forum), Munich, Germany, March 2005.

Hughes, Z.M. and Pont, M.J. (2004***) "Design and test of a task guardian for use in TTCS embedded systems"***. In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.16-25. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Abbreviations

AOT	Allowed Overrun Time
ASIC	Application-Specific Integrated Circuit
BCET	Best Case Execution Time
BDS	Branch Delay Slot
BTFN	Backward Taken, Forward Not taken
CISC	Complex Instruction Set Computer
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
CRPD	Cache Related Pre-emption Delay
DRAM	Dynamic Random Access Memory
EX	Execute
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
GPT	Guaranteed Processor Time
ID	Instruction Decode
IF	Instruction Fetch
ILP	Instruction Level Parallelism
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine

LED	Light Emitting Diode
LTE	Long Timing Effects
LUT	Look Up Table
MEM	Memory
MIPS	Microprocessor without Interlocked Pipeline Stages
MMU	Memory Management Unit
NOP	No Operation
RAM	Random access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTOS	Real-Time Operating System
SRAM	Static Random Access Memory
TG	Task Guardian
TLB	Translation Lookaside Buffer
TT	Time-Triggered
TTC	Time-Triggered Co-operative
TTH	Time-Triggered Hybrid
VLIW	Very Long Instruction Word
VLSI	Very-Large-Scale Integration
WB	Write Back
WCET	Worst-Case Execution Time

Chapter 1 Introduction

Beginning with Cantor, mathematicians have shown the uncertainty of mathematics and human logic (Dauben 1990; Davis 2004). Boltzmann showed the disorder of physics through fluid dynamics and that things can only be described in terms of probability (Boltzmann and Brush 1995). Gödel showed the incompleteness of pure mathematics proving that all systems of mathematical logic were limited, that there would be things that while true, would never be able to be proved true (Smullyan 1992). Turing recast incompleteness in terms of computers and showed that since they are logic machines, incompleteness meant there would always be some problems they would never solve (Turing 1939). A machine fed one of these problems would never stop and furthermore, Turing proved there was no way of knowing beforehand which problems these were. This condition was defined as 'incomputable' (Boolos, Burgess et al. 2007). Turing then related this to the human mind and the limitations of logic and stated that *"mathematical reasoning may be regarded rather schematically as the combination of two faculties, which we may call intuition and ingenuity"* (Turing 1939). The then notions that there was a perfect logic that governed a world of certainties had unravelled itself, that logic had revealed the limitations of logic and certainty had revealed uncertainty (Malone 2008).

This may appear to be obvious to us in the current modern world, especially where many software projects appear to be plagued by numerous bugs. However, this is a serious problem when designing safety critical systems where the failure or malfunction may result in death, injury, loss or damage to equipment and environmental harm. It is therefore desirable to achieve some form of certainty to determine if a particular machine will operate correctly.

Whilst there exist many mechanisms and formalisms that attempt to describe, model and test safety critical systems (Bowen and Stavridou 1993), it is not always possible to be completely certain that these machines are indeed safe (Isaksen, Bowen et al. 1997). This is because logic machines like computers have a subset of mathematical logic to which software programs can exist and they will often have to interact with their environment, which as fallible humans we often struggle to describe.

It is because of a range of human factors (Beaty 1995) and reasons such as those just described, that many safety critical systems don't rely solely on testing or formal specifications but on a combination of systems including redundancy and fault tolerant techniques (Nett, Streich et al. 1996; Lundqvist, Srinivasan et al. 2005; Boussemart, Ouimet et al. 2006; Short and Pont 2007). As such it will be argued in this thesis that, if the probability of an error occurring as a result of a systems design is to be reduced, designers must strive to eliminate as much uncertainty as possible. Therefore it is pertinent to develop systems based on highly predictable components. This becomes particularly significant as the complexity of a system increases (Wolfgang 2004).

The emphasis throughout this thesis will be focused on the issues of predictability when designing and building applications where safety is critical. In particular, this issue will be examined in regard to the field of embedded systems.

1.1 Embedded Systems

An embedded system is defined as a system in which a computer is encapsulated by the device it controls (Wolf 2008). In short, an embedded system is typically a computer that does not look like your conventional desktop PC (Pont 2002). In many cases the user may even be unaware that a computer exists within the device.

One of the first recognizably modern embedded systems was the Apollo Guidance Computer which was used for the NASA missions to the moon (Figure 1) (Hoganson 2007).

Figure 1: Apollo Guidance Computer (Image courtesy of Computer History Museum)

Today embedded systems can form a core component of a vast range of everyday items (cars, aircraft, medical equipment, factory systems, mobile phones, DVD players, music players, microwave ovens, toys etc). Over the recent years, the embedded systems market has increased significantly. For instance, in 1999 it was estimated that for every desktop computer sold, there were approximately 100 embedded processors sold (Turley 1999). Estimates state that in 2010 there will be 3 embedded devices per person on earth (ARTEMIS 2004).

A large portion of the embedded market serves the safety critical sector. This can be highlighted by the predictive growth of the automotive industry which has been the fastest growing European semiconductor consumer (Leen, Heffernan et al. 1999).

Initially embedded processors were first used in cars for engine management units (EMU) after stringent legislations in California 1961 to reduce exhaust emissions (Flis 1983). Now they can be found throughout the car from anti-lock braking systems, active suspension, adaptive cruise control, to GPS navigation and environmental climate control (Figure 2).

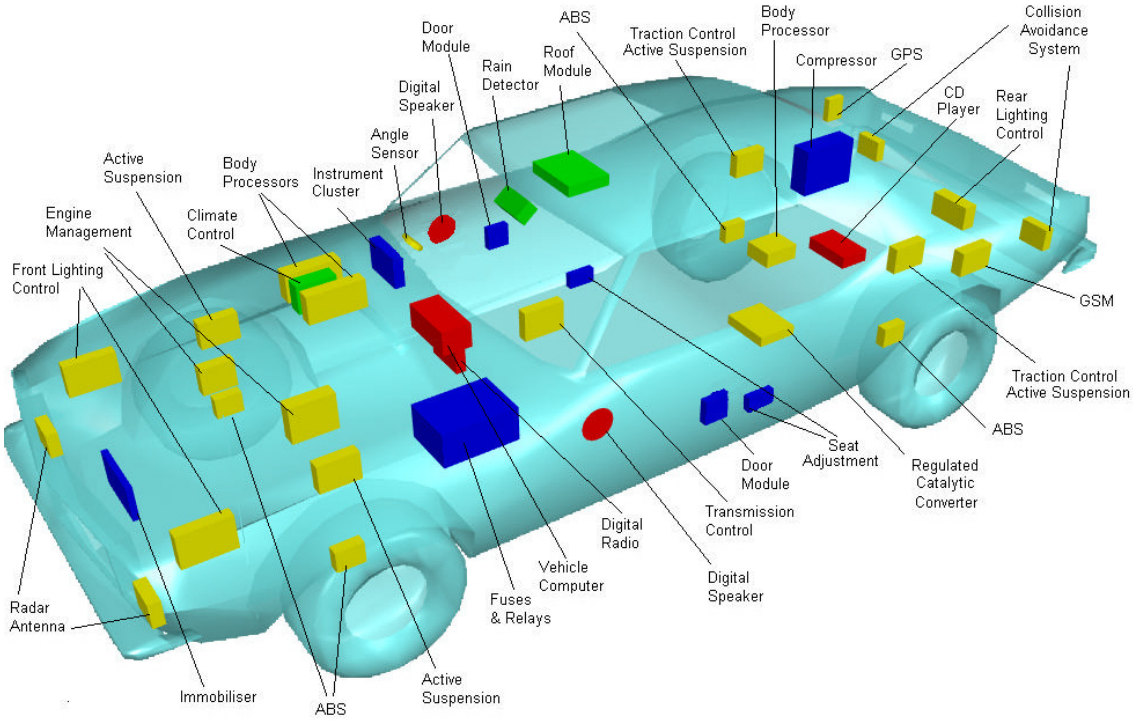


Figure 2: Available electronic devices in high-end cars (Leen, Heffernan et al. 1999)

For modern luxury vehicles the cost of electronics can amount to more than 23% of the total manufacturing cost. Analysts estimate that more than 80% of all automotive innovation now stems from electronics (Leen and Heffernan 2002).

To support these systems, the number of microprocessors in the average modern car has increased to about 40 - 100 (Turley 2003). So much so that the weight of the wiring looms to connect these devices together and the electrical power requirements are becoming a significant design issue (Leen and Heffernan 2002).

The automotive sector is an example of one area where the use of embedded systems is gaining greater emphasis - not just in a multimedia capacity - but also in a safety critical role. Many of these safety critical systems, such as drive-by-wire, have their heritage born in the aviation sector where a new aircraft such as the Airbus A380 can cost around 205 million GBP (Kaminski-Morrow 2008). However, the modern day car is just a fraction of the cost and meeting similar safety requirements within a competitive price point can be particularly challenging. This can be highlighted by the number of safety standards that a modern car must now conform to and still attain a relatively low time to market (SAE 1993; SAE 1994; Hardung, Kölzow et al. 2004; MISRA 2004).

The problem is further compounded as the complexity of the applications continues to increase. Unfortunately the scale of the problem of reliably developing safety critical embedded systems can be highlighted by the number of incidents that have occurred in the space and aviation industries, which by comparison generally have tighter more stringent standards, greater budgets and development times (Garman 1981; Carlow 1984; Fernando 1991; Reeves 1998; Stewart 2001; Driscoll, Hall et al. 2003; Charette 2005).

1.2 Challenges of Real-time Embedded Systems

Many safety critical embedded systems are generally categorised as real-time systems in which the system must be responsive to its environment.

By definition a real-time system is a computer-based system where the timing of a computed result is as important as the actual value (Liu 2000). Real-time systems do not always require that the value must be produced as quickly as possible, in many cases the desired property is steady and predictable behaviour (Stankovic 1988).

Hard real-time systems are a type of real-time system where the penalty incurred for missing a deadline can lead to catastrophic consequences (Buttazzo 2005). Examples of hard real-time systems are the devices used for protecting transformers from lightning surges on overhead power lines. These systems have to take the transformer offline within a millisecond from detecting a lightning strike. If the system meets this deadline then the transformer is protected. If the deadline is not met, then severe damage can occur. In this example, there is no extra value gained from the system being faster than required to meet the deadline (Engblom 2002).

For other hard real-time systems the computation time should have minimal variance (jitter) in producing the result. For example, the control systems in engine controller units (ECU) must regulate the amount of fuel and control the timing of spark plugs. If the spark plugs fire too early then the engine could be damaged. If they fire too late then the engine performance will decrease. Therefore the control algorithms should compute results at a precise time from when the measurements are taken as this is necessary in order to maintain good controller performance (Marti, Villa et al. 2001). Throughout this thesis, we will be concerned with these types of hard real-time system.

In order to guarantee that these systems operate correctly, the worst case behaviour must be analyzed and accounted for. Therefore it must be shown that all tasks can meet their respective deadlines even in the case that all tasks consume the maximum amount of time.

For completeness, a distinction is sometimes made between "hard" designs and "soft" real-time systems (Liu 2000). In soft real-time systems, the occasional failure to meet a deadline does not usually result in severe effects. For instance, skipping a frame in video playback is not fatal and might not even be noticeable by the user. Therefore, missing soft real-time deadlines often means that the system still provides a useful service but that the quality of the service is reduced.

1.3 Predictable Computer Systems

When describing the timing behaviour of a program, there are a number of common measures used. The best-case execution time (BCET) is the shortest time the program can ever take to execute. The worst-case execution time (WCET) is the longest time that a

program can take to execute. The average case execution time is a value that lies between the BCET and WCET (Figure 3).

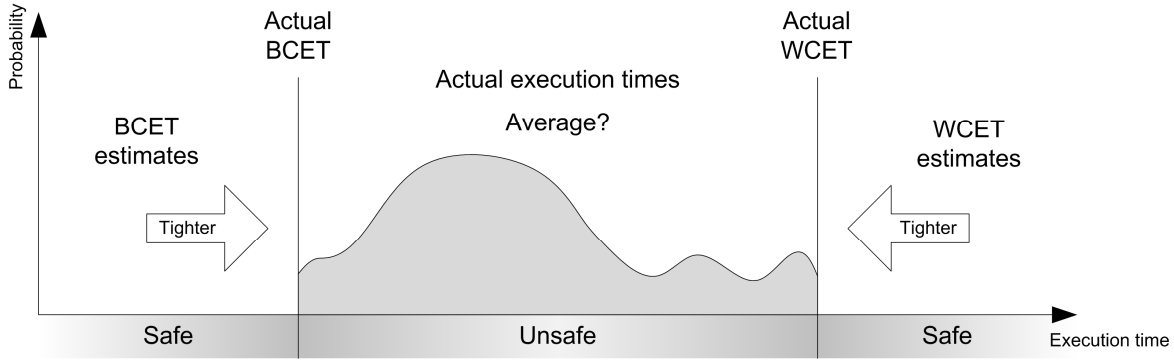


Figure 3: Execution time measurements, reproduced from (Engblom 2002)

Since a program can receive a number of inputs at run time, it can often be very difficult to determine the exact BCET and WCET values. It is often even harder to determine the average times as these can depend on the distribution of data and not just the extremes of program behaviour (Engblom 2002).

When building hard real-time systems to specification, timing analysis tools aim to produce estimates of the actual WCET and BCET values. In order for a WCET estimate to be safe, it must be greater than, or ideally, equal to the actual WCET. Similarly, the BCET has to be less than or equal to the actual BCET. This is shown by the estimates lying within the safe regions on Figure 3, any other WCET or BCET estimate is unsafe. This is because producing an underestimate WCET or overestimate BCET will result in the system relying on a false assumption which can ultimately fail (Engblom 2002; Puschner 2002; Wilhelm, Engblom et al. 2008).

For timing estimates to be useful they should be conservative and close (or tight) to the actual values, as shown by the 'tighter' arrows in Figure 3. An over estimated WCET value may be safe but can lead to an incredible waste of resources. However, large variation in the execution times and control flow of software can make producing estimates very complex. Therefore great care must be taken when producing safe and optimum values.

A system that has variable execution times and accurately defined BCET and WCET's, can be considered as predictable. However, within this thesis, predictability in the first place will be defined as the ability of the system to provide guarantees that its tasks will meet their designated timing constraints. Secondly, predictability must provide for minimal

timing deviations from the optimal timing scenario in which the system state can be accurately predicted down to the instruction execution level at any instance of time.

Therefore the temporal predictability of a system can be quantified as the jitter or variation in execution, and the summation of various causes of jitter which make it very hard to predict accurate execution times.

1.4 Building real-time systems

When building real-time systems, two types of models are commonly used, event-based and time-based. An event-based system is triggered by external events, whist time-based systems read their input signals periodically (usually driven by some clock). Another definition of time-triggered systems is that it is possible to determine in advance what the system is doing at every moment of time during the lifetime of the system (Pont 2002). This makes time-triggered systems particularly attractive for applications requiring high predictability.

Time-triggered (TT) architectures are often promoted for use in safety critical applications. For instance, Kopetz reports that *"the control system for Japan's Shinkansen bullet train uses a large time-triggered fault-tolerant system"*, and he claims that *"Safety-critical real-time computer applications for flight control, nuclear power plant shutdown, and so on, have to be fault tolerant and are therefore based on the time-triggered paradigm"* (Kopetz and Grünsteidl 1994).

A drawback of building TT real-time systems is that the synchronous execution of tasks needs more preplanning in the design phase and is less flexible than asynchronous tasks in event-based systems. For instance, care has to be taken to ensure that the system is both responsive to its environment and that each task is has enough available CPU time available to it. In the case that a task exceeds its allocated WCET the resulting effect may cause the other tasks in the system to miss their respective deadlines. These conditions are known as task overruns.

On the other hand, TT designs tend to have a very simple architecture which makes them easy to understand and maintain (Liu and Layland 1973), thus the maintenance costs can be substantially reduced. Furthermore, due to the predictable nature of TT designs, certification authorities (and company lawyers) tend to look favourably on systems using this architecture for safety-critical applications (Pont 2001).

TT scheduling architectures can differ between pre-emptive and co-operative (or non-pre-emptive) algorithms. Rate monotonic (TTRM) and deadline monotonic (TTDM) define scheduling algorithms that are fully pre-emptive, whilst cyclic executives and time-triggered co-operative (TTC) algorithms define co-operative based systems (Bate 1998; Pont 2001).

A key benefit of co-operative algorithms is the absence of deadlock situations and the associated complexity from dealing with shared resources among concurrent tasks. As a result, TTC architectures are often regarded as one of the simplest forms of time-triggered scheduling which are easier to use due to its low complexity and has been shown to be highly effective for a wide range of applications (Pont 2001; Kopetz 2008).

Overall, the focus of this thesis is on predictability and therefore the rest of the document will be based on time-triggered co-operative architecture. A more detailed discussion of the TTC architecture is given in Chapter 2.

1.5 Problems making real-time systems predictable

As discussed in the previous section, TT architectures can be expected to provide highly-predictable behaviour. However, assumptions about such behaviour rely (often implicitly) on knowledge of the operating characteristics of the underlying hardware. For instance, Engel. et al. notes that: *"Fulfilling an embedded application's real-time requirements depends largely on knowing the timing properties of the underlying hardware and OS."* (Engel, Kuz et al. 2004). It is therefore essential - that if predictable timing behaviour is to be obtained - that the temporal characteristics of the systems hardware can be easily understood.

Often embedded systems are built upon highly predictable electronics driven by a high precision oscillator to provide accurate and synchronous clock signals. However, as the layers of abstraction (such as processor architecture, operating system and application software) are applied, the predictable timing behaviour generally appears to decrease (Edwards and Lee 2007).

The developments in microprocessor architectures have often been designed to meet the classical objective of maximising (average) resource utilisation. Embedded processors are slowly following this trend by integrating some of the features commonly found on desktop and server architectures, where the goal is performance rather than predictability. As a result, these architectures are displaying greater variations in the execution time of

tasks and can make determining the systems temporal properties highly complex. Features that contribute to this complexity include caches, DMA, pipelines, branch prediction and instruction level parallelism (Berg, Engblom et al. 2004).

Due to these temporal complexities, modern timing analysis tools are struggling to keep up with modern computer design (Rapita 2008; Wilhelm, Engblom et al. 2008). The difficulty is being able to accurately model the processor hardware for static analysis or take timing measurements that are known to be both accurate and safe (as described in Section 1.3). This results in reduced confidence in the guarantees that the system will meet its objectives.

The problem can then be summarised as follows: if time-triggered systems are to be considered as predictable but run on processors that are regarded as hard to predict, it follows that in essence that time-triggered systems cannot be predictable.

Therefore, a predictable processor architecture is required in order to allow time-triggered co-operative systems to run predictably and alleviate the problems associated in providing reliable guarantees that the system will meet its requirements.

1.6 Aims of this thesis

The aim of this thesis is to design and evaluate a highly predictable embedded processor specifically suited for time-triggered co-operative systems in which each instruction has a single temporal response under all conditions in order to facilitate simple static program analysis and highly accurate time predictability.

1.7 Key contributions

- Development of a novel processor core for time-triggered systems.
- Development of a hardware-based scheduler for use with TT systems.
- Development of a set of software-based error-detection (and recovery) mechanisms to deal with issues of task overruns in TT systems.
- Translation of the above task overrun solution into hardware, thereby improving performance and reducing software complex

1.8 Thesis Overview

The rest of this thesis is organized as follows:

- Chapter 2 gives an overview of the key parameters and characteristics involved in selecting a predictable scheduler for hard real-time embedded systems.
- Chapter 3 reviews how the features in modern processor architecture have an impact on the ability of the system designer to predict how the system will behave.
- Chapter 4 presents a design for a predictable processor by first considering the architectural decisions and then adding new features to support predictable execution of TT architectures.
- Chapter 5 presents a design for a hardware-based TTC scheduler which minimises the complexities of scheduling overheads.
- Chapter 6 presents a set of software-based error-detection (and recovery) mechanisms to deal with issues of task overruns in TT systems.
- Chapter 7 presents a hardware solution to the task overrun problem in order to improve performance and reduce software complexity.
- Chapter 8 draws conclusions from the work presented throughout this thesis and discuss the future work in the area of increasing predictability in embedded systems.

1.9 Conclusions

This chapter introduced the challenges involved with creating predictable embedded systems and reviewed the main contributions of the research presented in this thesis. These contributions are centred on achieving highly predictable behaviour for time-triggered real-time embedded systems.

Chapter 2 Software architectures for embedded systems

2.1 Introduction

Chapter 1 discussed that for embedded systems and in particular for hard real-time systems to be predictable, the underlying architecture must be predictable. In software this relates to the way code is executed.

For many systems the functional requirements are often broken down into a group of smaller tasks. For hard real-time systems, these tasks must then be proved to be correct (Section 1.4).

In order to achieve the required system behaviour, a scheduler algorithm can be used to execute the tasks in the system in a specific order based on a set of constraints. Buttazzo breaks these constraints down into three key areas; timing, precedence and resource constraints (Buttazzo 2005).

2.1.1 Timing constraints

For each task in the system there are a number of commonly used timing parameters to define its temporal behaviour. These parameters can be met by an appropriate scheduling algorithm.

The "arrival time" or "release time" parameter is defined as the time at which the task is ready for execution. The start time defines the point when the task actually begins execution. The finishing time is the time in a particular run where the task actually finishes execution. The relative deadline is the maximum delay from the release time at which the task must complete. BCET and WCET parameters are described in Section 1.3.

Tasks are further broken down based on their arrival characteristics into periodic and sporadic.

For periodic tasks a new instance is executed periodically after a fixed duration between each task release (Buttazzo and Caccamo 1999). A benefit for periodic systems is that the future release times of tasks are known in advance. Also many real-time applications are periodic in nature such as data acquisition and control systems (Halang and Stoyenko 1994;

Krishna and Shin 1997; Buttazzo 2005). Periodic systems are often promoted for hard real-time systems (Spuri, Buttazzo et al. 1995; Pont 2001).

Sporadic tasks differ from periodic tasks in that the tasks are activated at irregular rates (Jeffay, Stanat et al. 1991). Examples of sporadic tasks are non-periodic device interrupt handlers and user requested task activations such as graphical user interfaces (GUI) (Spuri, Buttazzo et al. 1995; Nolte 2003; Buttazzo 2005). Sporadic tasks are also generally considered to be unpredictable (Kopetz 1991).

2.1.2 Precedence constraints

Another requirement is that tasks are executed in a particular order. This relation is known as the precedence (Mohammadi and Akl 2005). An example could be a data acquisition system in which there might be an input sensor task and computational task followed by an output task. This characteristic is common in systems such as simple cruise control (Ayavoo, Pont et al. 2005).

For groups of tasks that have precedence relations, the temporal behaviour is not just based on the individual task but by the set of tasks in the relation. Therefore all tasks must complete before the overall deadline (Halang and Stoyenko 1994; Ramamritham and Stankovic 1994). Precedence can also apply to task sets across multiple processors (Jensen, Locke et al. 1985).

2.1.3 Resource constraints

Tasks can also be classified as independent or dependent. Discounting the CPU, independent tasks do not share any other resources (Blake 1992; James, Hawick et al. 1999). Alternatively, dependant tasks require mutually exclusive access to shared resources (Cottet, Delacroix et al. 2002; Merrick, Wang et al. 2005). Shared resources are considered to have negative impacts on temporal predictability due to the need to consider the timing impacts of the locking mechanisms and the associated tasks that also share the resource (Audsley and Burns 1990).

2.2 Real-time task scheduling

In order to manage the various task constraints a suitable scheduling mechanism can be used. This section will discuss the differences in scheduling approaches and the impact they have on the predictability of the system.

2.2.1 Time-triggered versus event-triggered

Scheduling architectures can be broken down into time-triggered and event-triggered systems. A time-triggered system releases tasks based on specific time instances which are known before the systems starts (Liu 2000). Event-triggered systems release tasks based on a set of events such as external interrupts.

Event-triggered systems are useful for applications requiring high responsiveness, flexibility and the capability to handle sporadic events (Kopetz 1997). However, since the frequency, timing and possibility for simultaneous events are hard to predict, the run-time behaviour is unknown until the system starts and therefore the system is unpredictable (Kopetz 1991).

Time-triggered systems detect external events by polling the event sources periodically (Pont 2001). The responsiveness of the system is then dependent on how frequently the polling tasks are called. If the tasks are called very frequently then the system may be loaded unnecessarily (Pont 2001). Therefore, in time-triggered systems more pre-planning is required in the design stage. However, since the task executions are based on static pre-determined schedules, the system loads and systems response times are known in advance (Locke 1992; Kopetz 1997; Liu 2000). The state-synchronisation on time-triggered systems is also better as the tasks are based around a synchronised global clock (Obermaisser 2004).

2.2.2 Offline versus online

Schedulers also differ in the time at which scheduling decisions are made. Scheduling decisions can be made using either offline or online algorithms (Liu 2000; Cottet, Delacroix et al. 2002). Offline algorithms contain a static plan of task parameters which can be stored as a table of procedure calls (Stankovic, Spuri et al. 1995; Nolte 2003). The availability of the pre-runtime schedule makes the system more predictable and easier for the designer to ensure that the constraints of the system will be met (Xu and Parnas 1993).

Online algorithms differ in that the task schedule is determined dynamically based on the parameters that are known at runtime (Cottet, Delacroix et al. 2002). A benefit to this behaviour is that the schedule is flexible and can evolve to the demands of varying environmental conditions, especially where the future system loads are unknown. However, online algorithms are unpredictable and require the additional overhead of calculating the schedule between task activations (Liu 2000).

2.2.3 Co-operative versus pre-emptive

Scheduling strategies can differ between co-operative and pre-emptive systems (Liu 2000). In co-operative scheduling, tasks are given control of the processor and must co-operate with the system by returning control to the scheduler once the task has completed (Pont 2001). Therefore since tasks are not interrupted and run to completion complex locking mechanisms are not required as resources are not shared.

In pre-emptive scheduling a task may be interrupted whilst higher priority tasks are given control of the processor (Audsley, Burns et al. 1995; Nissanke 1997). This involves saving the current tasks state so that it can be paused and restarted once the higher priority tasks have completed (Liu 2000).

2.2.4 Fixed-priority versus dynamic-priority

Real-time scheduling schemes execute tasks based on a set of priorities. The priority assignment for tasks can be fixed or dynamic in nature (Audsley, Burns et al. 1991; Buttazzo 2005).

In fixed priority assignment, tasks are given a priority that is maintained throughout the lifetime of the system (Burns 1991). Since the priority of tasks are known before the system runs, fixed priority systems are promoted for the use in hard real-time systems (Tia, Liu et al. 1996).

Dynamic priority assignment allows tasks to be given different priorities during the lifetime of the system (Liu 2000). These systems will adapt the task schedule to the conditions of the environment to meet the required demands. Whilst dynamic priority is more flexible than static priority, it is also less predictable (Bini and Buttazzo 2004).

2.3 The time-triggered co-operative scheduler

Based on the criteria in the previous sections, a predictable hard real-time system should contain a time-triggered co-operative (TTC) scheduler with offline and statically prioritized tasks. In fact, in literature this type of system is known as a cyclic executive (Baker and Shaw 1988; Locke 1992; Bate 1998) and is widely used in the design of safety critical systems in areas such as the automotive and avionics sector (Carlow 1984; Kopetz and Grünsteidl 1994; Tindell, Kopetz et al. 2003).

There are a few drawbacks to TTC scheduling. For instance, whilst the implementation is simple, the structure can be considered to be rigid and inflexible (Locke 1992), implementing tasks to handle sporadic events through polling can be expensive (Bate 1998) and the schedule has to be re-analysed for every change made. However, the TTC scheduling framework is highly predictable and has been shown to be suitable for a wide range of applications (Pont 2001; Kopetz 2008).

While such a scheduler is not suitable for all systems, when compared to other architectures, time-triggered, co-operative scheduled (TTC) systems are known to provide a simple, low-cost and highly predictable platform. For instance, various studies have demonstrated that when compared to pre-emptive schedulers, co-operative schedulers have a number of desirable features, particularly for use in safety-related systems. For example, (Nissanke 1997) notes: *"[pre-emptive] schedules carry greater runtime overheads because of the need for context switching – storage and retrieval of partially computed results. [Co-operative] algorithms do not incur such overhead. Other advantages of [co-operative] algorithms include their better understandability, greater predictability, ease of testing and the inherent capability for guaranteeing exclusive access to and shared resource or data."*. (Allworth 1981) notes: *"Significant advantages are obtained when using this [co-operative] technique. Since the processes are not interruptible, poor synchronisation does not give rise to the problem of shared data. Shared subroutines can be implemented without producing re-entrant code or implementing lock and unlock mechanisms"*. Also (Bate 2000) identified the following four advantages of co-operative scheduling when compared to pre-emptive alternatives: The scheduler is simpler; The overheads are reduced; Testing is easier and Certification authorities tend to support this form of scheduling.

Within this thesis, the most important constraint is predictability and therefore the TTC scheduler will be used throughout the work described here[2].

2.3.1 Implementation

The particular TTC scheduler implementation which will be used throughout this thesis is be based around the design described in (Pont 2001). This is because the source code is well documented and the subject of a number of research papers (Pont 2003; Phatrapornnant and Pont 2006; Vidler and Pont 2006; Gendy and Pont 2008). An additional benefit is that the source code is freely available for use in research projects.

A TTC scheduler is characterised by a periodic scheduler tick in which one of more tasks can execute as long as they return in a time less than the tick interval. In Figure 4, a scheduler tick is denoted by the upward arrows indicating the points when the system is interrupted. The tick interval is the periodic time between the scheduler ticks. Tasks are scheduled after a tick and must return before the end of the current tick interval.

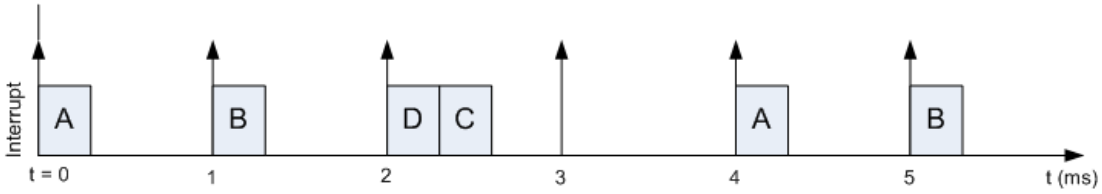


Figure 4: Example Time-Triggered Co-operative schedule

The scheduler tick is often implemented through a periodic timer interrupt which calls a scheduler interrupt service routine (ISR) called Update. The Update function will increment a tick count value which is required by a scheduler Dispatch function in order to determine when a task can be released. At runtime the Dispatch function is called from the main function inside an infinite while loop. Using the tick count value, a task queue is updated and the tasks are released based on the their period and fixed priority. An example of a typical TTC schedule is shown in Figure 4.

[2] It is noted that other less predictable and more flexible pre-emptive scheduling architectures such as rate-monotonic, deadline-monotonic, earliest deadline first and least laxity first could be considered but is felt to be out of the scope of the work presented here.

2.4 The Time-Triggered Hybrid Scheduler

An alternative to TTC can be to include support for an optional single pre-emptive task and allow the co-operative tasks to extend beyond the tick interval. This allows for a mixture of long and short task executions in order to further expand the range of applications that can be supported.

For instance, whilst the TTC architecture has many attractive features, the solution is not always appropriate. As Allworth has noted: *"[The] main drawback with this [co-operative] approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response [of the] system is not to be impaired."* (Allworth 1981). This concern can be expressed slightly more formally by noting that if a system is being designed which must execute one or more tasks of (worst-case) execution time e and also respond within an interval t to external events then, in situations where $t < e$, a pure co-operative scheduler will not generally be suitable.

In such circumstances, it is tempting to opt immediately for a fully pre-emptive design. Indeed, some studies seem to suggest that this is the only alternative (Locke 1992; Bate 1997). However, another design option is to include the support for a single, time-triggered, pre-emptive task that can be added to a TTC architecture, to create a "time-triggered hybrid" (TTH) scheduler (Pont 2001; Maaita and Pont 2005).

Use of a TTH scheduler allows the system designer to create a static schedule made up of (i) a collection of tasks which operate co-operatively and (ii) a single – short - pre-emptive task. In many designs, the pre-emptive task will be used for periodic data acquisition, typically through an analogue-to-digital converter or similar device: such requirements is common in, for example, a wide range of control systems (Buttazzo 2005).

The operation of the TTH architecture is illustrated schematically in Figure 5. This figure shows the situation where a single short pre-emptive task is executed every millisecond, while a co-operative task (with a duration greater than 1 ms) is "simultaneously" executed every 3 milliseconds. Note that pre-emptive task will interrupt the co-operative task when it is due to run and therefore shared resources must be handled with care.

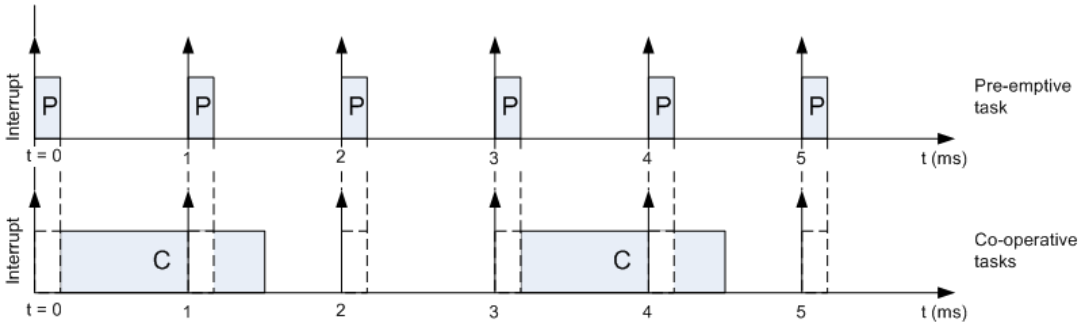


Figure 5: Illustrating the operation of a TTH scheduler

The TTH architecture will, in many cases, be used to implement a common "rate-monotonic" schedule: although it should be emphasised that this architecture only supports a single pre-emptive task. As a consequence, in a resource-constrained embedded system, it is a very attractive proposition because it allows creation of a scheduler with minimal resource requirements which is precisely matched to the needs of many practical applications. However, for the most part, this thesis will be concerned with TTC scheduling since a TTH framework reduces the predictability slightly due to the inclusion of pre-emption and the issue of dealing with shared resources.

2.5 Conclusion

This chapter has discussed some of the key parameters and characteristics involved when selecting an appropriate scheduler for high-reliability embedded systems. By taking the most predictable design choices an offline and static priority time-triggered co-operatively scheduled system was selected in order to provide highly predictable and deterministic system behaviour. Therefore the rest of this thesis will be primarily based around the TTC scheduling architecture.

The next chapter considers the issues of temporal predictability in the design of microprocessor architecture.

Chapter 3 The challenges involved in creating "predictable" microprocessor hardware

This chapter reviews the challenges in designing predictable microprocessors and the current work that has been done to improve temporal predictability. Anyone not familiar with modern processor architecture may wish to take a look through Appendix D which describes how computer hardware has developed over the past 50 years.

3.1 Introduction

This chapter aims to demonstrate the problems of understanding and predicting the temporal properties of software as it executes on modern processor hardware. Figure 6 illustrates this problem where an application programmer will typically combine a set of tasks and an RTOS to form the code that will run on the processor. However, the temporal properties of executing that code is often unknown, leading to an output behaviour that is difficult to predict.

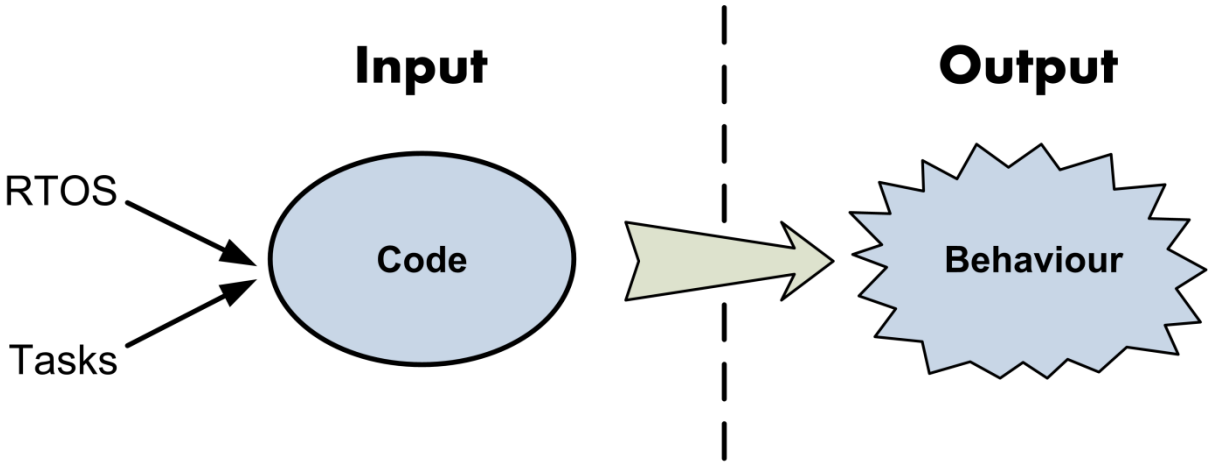


Figure 6: Mismatch between code input and anticipated temporal behaviour

As mentioned in Section 1.3, one of the key temporal properties required for hard real-time systems is WCET, therefore this chapter begins by considering the challenges involved in WCET analysis.

3.2 Difficulties in WCET analysis

One of the key issues in timing analysis is attempting to solve the problems of obtaining accurate and tightly bounded execution times. For hard real-time systems, the most crucial of these is to obtain the upper bounds (WCET) of a program in order to guarantee that deadlines can be met.

In fact, Wilhelm and colleagues have argued that *"Unfortunately, it is not possible, in general, to obtain upper bounds on execution times for programs."* (Wilhelm, Engblom et al. 2008). Whilst using a restrictive form of programming by bounding loops and removing recursion can help, they argue that determining the worst case inputs to the system can be very complex to derive. They state that even if the worst case inputs can be derived then the state space is often too large to explore all possible executions.

A common approach to obtain timing estimates is to take measurements for a set of test cases. However, these values will usually under estimate the correct BCET or WCET. The problem is knowing if the worst case or best case times have indeed been measured. Therefore, measurement based techniques are not guaranteed to be safe and are generally unsuitable for hard real-time systems, especially as a primary source of timing analysis.

The only way to achieve accurate bounds on execution times is to explore the computed time for all possible task executions. However, since the state space is so large, a more feasible approach is to abstract the system model (Rapita 2008). A problem with this form of static analysis is that while safe, the abstraction loses information about the system states and will usually produce overestimated timing values.

The way this relates to the processor hardware is that in order to obtain safe and tight bounds, all the system states must be explored or abstracted conservatively. This therefore requires an accurate model of the underlying hardware and its features. In some cases the best model of the processor is the processor itself (Bernat, Davis et al. 2007).

The ability to create accurate models of the hardware depends largely on the complexity and temporal predictability of the techniques used. Some of these features can reduce predictability due to the use of speed-up mechanisms like caches, instruction pipelines, parallel instruction-execution units, and branch-prediction. Some of these mechanisms are complex in their implementation and have mutual interferences in their timing. In addition,

the timing properties are rarely documented to protect the manufacturer's intellectual property. These issues make it very difficult to build static WCET analysis tools for modern processors that can be guaranteed to model the timing of the processor correctly (Puschner 2002).

This chapter aims to explore some of the underlying temporal predictability problems in modern processor architecture.

3.3 Principles for a time predictable processor

Identifying the complexities in temporal predictability for modern processor architecture are best described through the principles for a time predictable processor.

(Thiele and Wilhelm 2004) identified 4 key threats to predictability in processor architectures:

Non-Deterministic behaviour - A significant cause for low temporal predictability can arise through the systems sensitivity to interfering and non-deterministic behaviour. For example, through the reception of sporadic interrupts.

High Variability – A number of features such as caches, pipelines and out of order execution can cause a high variation in execution times. These variations can combine together to have a larger impact on the temporal properties of the system.

Non-Analysable – Some components can exhibit properties which are non-analysable and can make determining accurate execution times impossible.

Complexity - There are some components that can be analysed but the computational effort to do so results in such high complexity and low performance that the effort required is not practical.

These properties have been observed and found to be responsible for temporal predictability in a number of system components (Heckmann, Langenbach et al. 2003) and will be described in more detail in the following sections.

3.4 Issues with current processor architecture

Some of the problems with timing analysis have been shown to come about due to the following processor features:

- Instruction Set
- Pipelines
- Branch Prediction
- Memory systems
- Documentation

These issues will be discussed in more detail in this section.

3.4.1 Instruction Set

Often one of the early decisions in a processor design is to choose an appropriate instruction set architecture (ISA) (Patterson and Hennessy 2005). Instructions sets can differ between size, CISC and RISC, and modern instruction sets also differ between variable and fixed length.

Current embedded system trends indicate that variable length instruction sets have a better code density which is an important design parameter for many real-time systems. Examples of processors with variable length instruction sets can include some incarnations of ARM and MIPS processors. These processors can contain a mixture of 16bit and 32bit instructions.

A problem with variable length instruction sets is that they can make the predictability for cache analysis more complex, for instance by adding extra instruction cycles for branches to misaligned instructions when only part of the instruction is cached. Due to the code density of variable length instruction sets the number of memory cycles are decreased, giving better average performance than fixed length instruction sets.

3.4.2 Pipelines

A design feature that can link closely to the instructions set is the option to utilize a pipelined architecture.

The predictability of a particular pipeline system can depend on the power of the available analysis methods. With pipeline analysis two levels of complexity have been defined, one-shot and fixed-point analysis (Berg, Engblom et al. 2004).

One-shot analysis is a fast method which is only suitable for simple pipelines as each instruction is assumed to have a single and deterministic pipeline behaviour. Fixed-point analysis uses a collection of abstracted pipeline states which are used to evaluate all possible states. This makes fixed-point analysis more suitable for complex pipeline architectures but entails higher computational costs. Some architectures feature characteristics that are not suitable for either methods. For example, a PowerPC 755 processor was found to have cases where an instruction could have up to 1000 states (Heckmann, Langenbach et al. 2003). Furthermore, some systems suffer from long timing effects (LTEs) and can cause an unmanageable amount of complexity in fixed point analysis and render one-shot analysis completely infeasible.

LTEs are the effect when the timing behaviour of the current instruction depends on the execution history of previous instructions which are not its direct neighbour in the instruction flow. This effect can range from between a few machine cycles in simple pipelines to propagating for potentially unbounded sequences of instructions, for example in caches (Engblom and Jonsson 2002). The result is high variability and local non-determinism which can affect the execution times of tasks and the ability to meet deadlines.

A more alarming problem is the issue of timing anomalies which display a counterintuitive influence on the expected timing behaviour between the local execution of instructions and its effect on the global execution of tasks (Lundqvist and Stenstr 1999; Reineke, Wachter et al. 2006). An example of this is the condition where a cache hit would normally translate into faster system execution, but in fact leads to the opposite. The ColdFire 5307 processor was observed to have this behaviour (Heckmann, Langenbach et al. 2003).

The first timing anomaly to be observed was in the instruction scheduler of superscalar out-of-order pipelines (Graham 1969). In such architecture, a speed up in one instruction could lead to a less efficient schedule which can affect many future instructions. The state space for such a system becomes very large and thus becomes a highly computational heavy problem for analysis tools.

The potential for timing anomalies has a large effect on the ability of timing analysis tools to provide accurate and safe estimates (Heckmann, Langenbach et al. 2003).

3.4.3 Branch Prediction

Dynamic branch prediction is generally considered unsuitable for use in real-time systems (Heckmann, Langenbach et al. 2003). This is because a number of the employed techniques break the recovery requirement which states that it is possible to recover the knowledge of the ISA or timing behaviour when it becomes unknown (Berg, Engblom et al. 2004). For dynamic branch prediction this is a problem because the history of previous branches affects the predictions of future branches (Engblom 2003).

When combined with caches, branch prediction has also been shown to be a cause of timing anomalies (Ferdinand, Heckmann et al. 2001). In such cases, executing a loop for more iterations could result in a reduction of execution time (Engblom 2003).

Dynamic branch prediction is also shown to break the non-interference principle where the prediction can cause parts of the cache contents to be replaced by instructions that are not executed. A large quantity of branches in a program can result in a significant disruption to the cache analysis and consequently the timing estimates (Berg, Engblom et al. 2004).

On modern processors such as the AMD Athlon, dynamic branch prediction is shown to make WCET analysis very difficult (Petters 2002). An alternative approach is to use static prediction based on statically known information, such as branch direction or special hint bits in the instruction. An example of a static predictor is the BTFN (backward branches taken, forward branches not taken) which is shown to be 60-70% accurate for typical embedded applications (Gwennap 1995; Levy 2002).

3.4.4 Cache Predictability

According to (Basumallick and Nilsen 1994; Sebek 2001) there are two main types of cache behaviour, intrinsic and extrinsic. Intrinsic behaviour is the condition where two parts in the same task compete for the same area in cache. As a result the two parts invalidate each other and the performance is reduced. Extrinsic behaviour is the result of separate tasks invalidating each other, mainly due to pre-emption where the contents of the cache are replaced with the contents of a new task. This is also known as the cache related pre-emption delay (CRPD) (Sebek 2001).

A common perception is that the use of cache is not suitable for applications requiring predictable behaviour. As a result it can be common practice to disable cache for critical sections of code where predictability is required (Mueller, Whalley et al. 1993).

An alternative solution to the problem is the use of a locally fast on-chip RAM area in which the compiler or programmer can selectively map key sections of code and data. However, the use of large amounts of on-chip RAM comes at a cost penalty and performance depends on the effectiveness to map values that will provide a maximum benefit.

Despite some of the negative aspects for the predictability of cache, there are some researchers who argue that modern techniques can provide accurate estimates. For instance once the cache is loaded with the values then the analysis is said to model the real behaviour accurately (Ferdinand, Heckmann et al. 2001). Instruction caches can also be predicted quite accurately when the program flow and memory accesses are known (Mueller, Whalley et al. 1993). However, the assumption is usually made that the code being analysed is between two points (such as context switches) where the cache is assumed to be invalidated.

There are also claims of confidence in the predictability of data cache (Berg, Engblom et al. 2004). For example, one study (Ferdinand, Heckmann et al. 2001) shows that in benchmarks supplied by Airbus found over 90% of the data accesses to be predictable.

However, for systems in which the instruction and data accesses share a unified cache, an interference problem occurs. This is where instruction and data accesses can invalidate

each other and like the similar problem in branch prediction it can affect the overall performance (Berg, Engblom et al. 2004).

Cache are also known for exhibiting timing anomalies on out-of-order pipelines where a cache hit can result in the worst case timing. Furthermore a miss penalty can be higher than expected due to its effect on the instruction scheduler which can cause older instructions to execute earlier than normal (Lundqvist and Stenstr 1999).

A few solutions have been provided to get round the problems of extrinsic cache behaviour from task pre-emption. A simple solution is to flush the cache on each context switch so that each execution run starts in a known condition (Niehaus, Nahum et al. 1991). However the performance can be significantly impacted if pre-emption occurs very frequently. An alternate solution to flushing the cache after every context switch is to make the pre-emption points known so that the disruption to cache can be modelled in the analysis (Simonson and Patel 1995). Kirner and Puschner used this idea in combination with an instruction counter rather than a timer to identify the exact points in the code where the pre-emption would occur (Kirner and Puschner 2007). This solution was shown to be predictable but it does require that the task schedule be static or deterministic so that it can be included within the model.

Another approach to increasing the cache predictability is the use of partitioning to split the cache into segments which can be assigned to individual tasks (Kirk 1989; Kirk and Strosnider 1990; Mueller 1995). This can be achieved either through hardware or in software by carefully mapping the tasks to memory locations that do not compete for the same cache lines. A problem with cache partitioning is that each task is only given a small portion of the available cache and the task scheduler must incorporate mechanisms for protecting and allocating the cache resource.

A solution proposed by (McFarling 1989) uses a dedicated bit in the instruction set to indicate if a particular instruction should be cached. This provides a solution for systems where only certain portions of the code contain time critical sections and where the rest of the system can benefit from the full use of the cache. (Chi and Dietz 1989) used a similar technique in load and store instructions to indicate if data should be cached.

Another approach is the use of cache locking mechanisms where values are loaded into the cache and locked to prevent further change (Akgul and Mooney 2002). Two methods exist,

static and dynamic cache locking. Static cache locking allows the values to be loaded once during system start up and are then never changed during the systems lifetime (Campoy, Ivars et al. 2001). Dynamic cache locking allows the values to be loaded and then changed at specific points, for instance when tasks are pre-empted (Campoy, Ivars et al. 2002). Experiments show that static locking is more predictable but has limited performance gains compared to dynamic locking (Campoy, Perles et al. 2003; Vera, Lisper et al. 2003; Vera, Lisper et al. 2007).

3.4.5 DRAM

Unlike SRAM, DRAM can cause problems to temporal predictability due to its need to refresh the capacitors that it uses to store data. Unfortunately the refresh cycles occur asynchronously to program execution and are an example of a non-analysable component. If the refresh occurs on the same data that is being accessed then the processor will be halted until the refresh is complete. The problem is being unable to know when the refresh cycles will collide with the data accesses and how often. Therefore assuming the worst case time on each memory access will usually result in a large overestimation of the WCET (Atanassov and Puschner 2001).

Initial measurements on the impact of DRAM refreshes showed that the effects were larger than expected (Park and Shaw 1990). However, more recent studies have shown the average effect to be about a 2% increase on execution time (Atanassov and Puschner 2001).

A proposed method to increase DRAM predictability includes a compiler based system to synchronise memory accesses with refresh cycles (Panda, Dutt et al. 1997). However, this knowledge of when the refresh occurs is unknown.

3.4.6 Direct Memory Access (DMA)

For DMA transfers there are two modes of operation, cycle stealing and burst mode (Colnaric and Halang 1993). In cycle stealing mode the DMA takes its turn with the CPU to make use of the data bus. In order to predict when the DMA operation will finish requires knowing the arbitration scheme of the bus. A more predictable approach is the burst mode where the CPU is paused until the transfer operation is complete. However, during this period it will reduce the processors ability to react to external events. Therefore

the DMA must be factored into the timing analysis much like any other task in the system (Pitter and Schoeberl 2007).

3.4.7 Memory Management Unit (MMU)

Memory management units provide two main functionalities. These are to make the main memory appear larger than the physical memory by offloading less frequently used pages to a slower memory medium, and to provide memory protection (Berg, Engblom et al. 2004). For hard real-time systems the features of memory protection are of most practical use as expanding the memory area beyond the physical memory is usually unpredictable in hardware and slow in software. Some recent work has been done to increase the predictability on directing the MMU page transfers through a compiler (Puaut and Hardy 2007). Another issue with using an MMU is the translation lookaside buffer is a form of cache memory for holding page table entries and therefore suffers from the similar problems as cache (Bennett and Audsley 2001). If all the relevant table entries are located in the TLB before the task starts then the behaviour is said to be predictable (Niehaus 1994).

3.4.8 Comprehensive Documentation

Processor documentation is scarcely catered for hard real-time systems in which deep information for the processors states and temporal qualities are required (Berg, Engblom et al. 2004). The reasons behind this lack of information might be due to competitive reasons. In some case even the documented information has been found to be incorrect, as (Atanassov and Puschner 2001) found in the case of DRAM timing. The hardware documentation for the C167 processor was also found to be inaccurate, leading to a 5% difference in measured and estimated times (Atanassov, Kirner et al. 2001).

If predictability is to be maintained and analysis tools are to provide safe and accurate estimates, then the processor hardware states should be both predictable and well documented.

3.5 Symptoms of unpredictable hardware

In the previous section the reduced predictability in processor architectures are shown to be the result of speed-up mechanisms such as caches, pipelines, branch prediction,

instruction level parallelism and out-of-order execution. This problem is made worse when the effects of these mechanisms combine to produce LTEs and timing anomalies. Furthermore the documentation rarely provides adequate information to produce accurate processor models. These combined issues then make it difficult to build static WCET tools and make guarantees that deadlines will be met (Puschner and Burns 2002). This also relates to a further problem in the ability to validate that the WCET tools are themselves correct (Engblom, Ermedahl et al. 2001).

When using static tools the analysis for a simple processor may take in the order of a few minutes, however the analysis of a complex processor, even when using an abstract model, has been shown to take about a day (Souyris, Pavec et al. 2005). As a result, the complexity and size of abstract models are increasing. With regard to modern tools, Wilhelm, Engblom et al. state that: *"Benchmarks published earlier offer better results regarding the degree of overestimation, although significant methodological progress has been made in the meantime"* (Wilhelm, Engblom et al. 2008). This is put down to the divergence between unpredictable processor architectures and the capabilities of modern static analysis tools. Therefore if applications use modern hardware and software architectures the complexity of making timing guarantees are inescapable (Puschner 2002).

If the analysis tools can provide safe WCET times for a modern processor, then the problem of increasingly large variation between the average and worst case execution times means that for a large proportion of the time the CPU is just idling. This leads to an underutilised processor and a waste of resources. One approach to the problem is to mix hard and soft real-time tasks so that the hard real-time tasks will always have the available CPU time and the soft real-time tasks will benefit when the hard real-time tasks perform the average or best case times. However, this can have limited appeal as some hard real-time designers may prefer to physically separate soft real-time tasks onto a separate hardware in order to simplify the verification process.

The next section looks at some of the work that has been carried out to help improve the predictability of processors.

3.6 Work that has been done to improve time predictability

Research is currently being undertaken in various areas to help improve the timing analysis of processors. These include techniques to generate formal specifications from hardware descriptive languages (Wilhelm 2004); model based software design (Feiler, Lewis et al. 2000) and single path programming techniques (Puschner and Burns 2002; Puschner 2003; Gendy and Pont 2007); to formal verification of pipelines (Ravi, Ganesh et al. 2003) and full processor models such as the ARM6 (Anthony 2003).

A few techniques have been used to improve the predictability of specific hardware components. These include work by (Stankovic, Niehaus et al. 1991) to provide a means in which sporadic interrupts can be handled through the use of a master processor core to schedule appropriate tasks on a collection of slave cores. Work has also been done to improve the predictability of cache (Kirk 1989; Stärner 1998; Akgul and Mooney 2002; Vera, Lisper et al. 2007) and memory systems (Pitter and Schoeberl 2007).

There are number of papers which highlight the design decisions for a predictable processor (Colnaric and Halang 1993; Ortega 1994; Colnaric, Verber et al. 1995; Berg, Engblom et al. 2004; Thiele and Wilhelm 2004). However, only a few complete processors have been designed with predictability at the forefront of the architectural decisions.

(Whitham and Audsley 2006) produced a processor design, named MGREP, as an alternative to high performance superscalar ILP through the use of microcode controlled reconfigurable logic. Whilst the system is shown to provide a high throughput, the design assumes that the reconfigurable logic can be reconfigured in a predictable and fast manner. However, they state that runtime reconfiguration is not well documented and that the system could be unresponsive during reconfiguration.

The CAR-SoC project builds on a multithreaded core design to run hard real-time tasks in a separate hardware thread than soft real-time tasks (Uhrig, Maier et al. 2005). This helps to increase responsiveness and partitioning between tasks although they state that the address pipeline is made complicated by the microcode engine. What the work does not do is to resolve the problems of execution times for individual threads.

The SPEAR processor design represents a 3-stage simple pipelined processor where data hazards are resolved by a forwarding unit and control hazards result in a pipeline stage

being flushed (Delvai, Huber et al. 2003). Each instruction is one word in length and has deterministic execution times. The processor utilizes separate on chip memory for both instruction and data accesses with the option to allow the use of lock cache memory. Interrupts have a maximum jitter of one clock cycle and a latency of 3 clock cycles. One of the key features that make this architecture appealing for timing analysis is that it provides support to help with single path programming by utilizing instruction condition codes. However, this architectures predictability relies on each instruction executing within a single cycle and therefore it does not include more advanced instructions like multiply or divide which are becoming more commonly used in embedded processors (Heath 2002; Großschädl and Savaş 2004).

Another approach to the problem is the virtual simple architecture (VISA) which will execute hard real-time tasks on a complex processor and will switch to a simpler processor if it does not finish in time (Anantaraman, Seth et al. 2003; Muller 2004). While this mechanism will provide safe upper limits to the WCET, the variations of the execution time could be large due to the difference in performance between the complex core and the simple core. Therefore the WCET values will not be optimal if considering the condition where all tasks miss there deadlines and must execute on the simple core.

A different approach is the design of a precision timed machine (PRET) which aims to implement a MIPS based processor with the inclusion of special timeout (deadline) instructions which ensure constant time periods between sections of code (Ip and Edwards 2006; Edwards and Lee 2007). If a section of code completes faster than expected, the deadline instruction will halt the processor for the requested number of clock cycles from the last deadline instruction. A problem with this approach is that it cannot handle interrupts and multitasking very well as the clock cycle counter may exceed the deadline amount by the time the interrupt has returned.

3.7 Conclusion

As described in Chapter 2, time-triggered software architectures can be used in a wide range of systems where high predictability is required. However, one of the problems is that its predictability is determined and in some cases undermined by the underlying hardware architecture.

This chapter has reviewed how many of the features in modern processor architectures have an impact on the ability of the system designer to know how the system will behave at any particular point in time. The un-deterministic temporal characteristics of these architectures reduce the ability of static analysis tools to obtain both accurate and safe timing estimates. It was shown that some architectures exhibit the presence of LTEs and timing anomalies which can make static timing analysis infeasible. These problems were found in components such as pipelines, branch prediction and cache memory. As a result, modern timing analysis tools are struggling to keep up with modern processor designs (Rapita 2008; Wilhelm, Engblom et al. 2008). Therefore in order to achieve "ultimate predictability" and reduce the complexities required by timing analysis, it is important that the execution of the instructions in the processor can 'in all circumstances' be predicted right down to the clock cycle level.

As discussed in Section 3.6, a few microprocessor designs have tried to address some of the issues of temporal predictability. However not all designs provide a full solution or include the support for multi-cycle instructions which are common for control and data acquisition tasks. Some of the widely used instructions in this category are integer multiply and divide.

The aim of the work presented in this thesis is to ensure that the processor will be predictable at the instructions level (even when incorporating multi-cycle instructions such as multiply and divide which can result in variable latencies). Previous designs have not achieved this level of predictability. In addition, the work here is focused on processors which support time-triggered architectures: previous studies have not considered the needs of such systems.

The next chapter will present a novel processor design which will be used throughout the remainder of this thesis to support predictable execution of time-triggered systems.

Chapter 4 Design of a predictable TT processor

4.1 Introduction

This chapter aims to explore the design for a highly predictable multi-cycle capable processor design targeted towards supporting time-triggered systems after reviewing the various architectural design choices by a using a bottom up approach. This implementation will put predictability as the highest design criterion before performance or cost. In some cases it may be simpler to adopt an archaic architecture. However, an archaic design is unlikely to meet the performance requirements of modern systems. The emphasis here will be on finding a balance between performance and cost without impinging on the first criterion of predictability.

4.2 Design choices

A summary of the design requirements for a predictable processor based on the architectural details laid out in Section 3.4 is listed below and will be described in greater detail within this Section.

- Fixed Length RISC Instruction Set
- Simple scalar pipelined architecture
- Static branch prediction
- Pipeline forwarding unit
- Harvard architecture for separate data and instruction buses
- Static access time SRAM

Due to the current trend and popularity for higher performance 32-bit embedded processors, the design here will be based on a 32-bit architecture. In Section 3.4.1, it was clear for reasons of simplicity which in turn relate to ease of analysis and predictability (Wolfgang 2004) that a fixed length RISC instruction should be used.

Whilst RISC is not always synonymous with pipelined architectures, it is possible to utilize a pipeline as long as it's possible to employ one-shot analysis and reduce the effect of long timing effects. Part of these requirements is that there must be no hardware interlocks which are made easier by keeping the pipeline length fairly shallow. The number of pipeline stages with variable length should also be kept to a minimum and the use of parallel pipelines should be avoided to prevent an explosion of potential instruction combinations. Using a simple pipeline will enable the processor to reach the kinds of levels of performance found in many modern processor architectures and can also help to keep power consumption low.

A common issue with pipelined architectures is that structural hazards can occur. This is when an instruction cannot be executed because the hardware is unable to support the current combination of instructions. An example of this can be when a data and instruction fetch occurs in the same cycle over a single bus. Therefore a conflict arises due to multiple pipeline stages fighting for memory access along the single shared resource. As a result, one of the pipeline stages must be paused until both stages can complete their memory accesses. To avoid such a conflict, the use of a true Harvard architecture with separate memory buses is proposed. This has the added benefit of isolating the program and data memory and can also be particularly useful in preventing attacks such as buffer overflows and accidental memory writes.

In memory hierarchies there are a number of arguments stating that cache flow can be modelled under certain conditions. Although many of these solutions are highly complex and are often not practical as they rely on clauses such as 'no use of interrupts' or 'multi-path programming'. However, as described in Section 3.4.4 there are some techniques such as cache locking and cache partitioning which avoid some of these problems. If used correctly, these techniques can also provide a reasonable speed increase. For reasons of simplicity, the use of caches will not be used at this stage.

Ideally the memory technology used for the processor should run at the processors clock speed. This is often possible in the form of on chip SRAM. Nevertheless, it's recognised that this can be costly especially if required in large quantity. Cheaper slower memory systems can also be used as long as the access times are constant. This is not the case for the popular DRAM chips which are known to suffer delays due to unpredictable memory refreshes. Using slower memory will have a significant impact on the overall systems

performance and as such, some memory chips make use of a pre-fetch buffer. Whilst the pre-fetch buffers can reduce the performance loss, care must be taken to understand fully the features of the particular implementation to ensure that it can be modelled correctly.

Since the design here is intended for highly predictable safety critical systems and cache will not be used at this stage, the cost of using fast on chip SRAM is deemed acceptable as an initial solution.

The use of a single cycle instruction set which is of fixed length can make analysis far simpler and can allow for fixed interrupt response times as is the case in the SPEAR design. Where possible this will be used, although as will be described later, it is not always practical to implement all instructions into a single cycle implementation.

Another effect of pipelined systems is the issue of control hazards. This occurs when an instruction has been fetched into the pipeline which is not going to be executed due to a flow control change. For example, when the outcome of branches are unknown until a later pipeline stage. In this scenario, any instructions that have already been fetched which do not follow the branch path will be flushed. Due to this problem, many modern processor architectures employ forms of branch prediction to reduce the wasted cycles from loading unwanted instructions.

As described in Section 3.4.3 dynamic branch prediction reduces predictability as its prediction relies on a history of previous branches which is difficult to model. On the other hand static branch prediction will make the same prediction on all instances of execution but at the cost of performance.

A method to guarantee an equal execution time, regardless if the branch is taken or not, is to pause subsequent instruction fetches until the branch outcome is known. However, this method always imposes the worst case performance and has the added requirement of hardware to pause instruction fetch stage. Thus it is advantageous to determine the branch direction in an early as possible pipeline stage.

An alternate method which also guarantees equal execution time across branches is the branch delay slot technique. This technique relies on the compiler placing a suitable instruction directly after the branch which will be executed regardless if the branch is taken or not. If no suitable instruction can be found, then a NOP instruction is used instead. The

performance for this method is based on the number of delay slots and the compilers ability to reorder instructions into these slots. This technique will be used in the design here as it simplifies the analysis process.

One final effect of a pipelined architecture is data hazards. A data hazard occurs when an instruction cannot execute because a previous instruction holds data that it requires to complete execution. In most cases this problem can be resolved by the use of a forwarding unit to make the most recent data value available to the execution pipeline stage. However, if an instruction relies on data that is currently being fetched from memory by the previous instruction, then it must wait until the value is ready. In such a scenario the instruction is often paused and is generally avoided by the compiler. However, when such a condition occurs, the execution time can be modelled by simple analysis of the object code. Other architectures can use the compiler to insert an instruction after the load which is not dependant on the value being fetched. This process is known as the load delay slot.

4.3 Selecting a processor platform

A key aspect of a processors design is the ISA. This is because the ISA links closely to the underlying hardware and the abstraction that it provides for software programs to run. A common requirement is to provide enough data, logical and conditional instructions to efficiently support and run a high level language such as C. The complexity of making this possible also comes from developing supporting tools such as compilers and debuggers. It is therefore advantageous to base the processor design around an existing ISA in order to reduce the design effort required for providing appropriate tools.

Some of the common and well known RISC ISAs are Alpha, PA-RISC, PowerPC, ARM, Thumb, SuperH, SPARC, M32R and MIPS. However, a number of these ISAs are the intellectual property of various companies, therefore only open ISAs can be considered. Out of this list, SPARC and MIPS are commonly used as templates for a number of processor designs. The SPARC ISA is an open standard and the MIPS ISA was born out of research by Patterson and Hennessy before being taken up by MIPS Technologies Inc. As a result, most of the MIPS instruction set is open apart from a few patented instructions which relate to unaligned memory accesses.

The MIPS architecture is fairly well understood due to the many books published about it, in particular by its creators Patterson and Hennessy (Patterson and Hennessy 2005; Hennessy and Patterson 2006), whilst SPARC which is also well documented has an open specification and certification procedure.

For the most part, both SPARC and MIPS have more architectural similarities than differences. One of the main differences between the architectures is the use of register windows. MIPS has a fixed 32 x 32-bit register file, whereas SPARC has 24 of its 32 registers overlapped by different register windows. The aim was to reduce register to memory traffic on procedure calls through the use of register windows. Thus the SPARC ISA allows for cores to be synthesised with 2 - 32 register windows, where only 32 of the registers are accessible at any one time. Another defining feature is that early MIPS architectures make use of the BDS instruction whereas SPARC will always flush out the BDS instructions in the case when the branch is taken. Early incarnations of SPARC also did not have hardware multiply and divide instructions.

Table 1 gives a more detailed overview of the architectural differences between the architectures.

MIPS	SPARC
32 x 32bit registers	8 global + 24 overlapping register windows.
Integer to Floating Point register move instructions	Integer to Floating Point load/store instructions
Single precision FP	Single + Double precision FP
Single load/store addressing mode	Two load/store addressing modes
Compare and branch in one instruction	Requires two instructions to do the same
Instruction in delay slot is executed	Hardware will flush the instruction if branch is taken
Requires compiler to avoid a load data instruction directly before an instruction using that data.	Hardware will pause the pipeline in such a scenario.
Can move values between single and double FP registers	Can only move values between single FP registers
Has integer multiply and divide/remainder instructions	Not available in SPARC
No FP square root instruction	Has square root instructions
No conditional trap instruction	Has conditional trap instructions
TLB managed in software	Memory management done in hardware
Immediate constant field is 16bits	Immediate constant field is 13bits
32-entry, 32bit FP register file which can hold up to 16 double precision values	Can hold a mixture of 32 single-precision values, 16 double precision values or 8 quad precision values

Table 1: Comparison of MIPS and SPARC, reproduced from (Robert, Shing et al. 1991)

When taking into account the architecture requirements for predictability, as was highlighted in Section 5.2, it can be seen that both architectures could be used.

Throughout this thesis the MIPS ISA was chosen as template for the processor design for the following reasons.

- One of the benefits of using MIPS over SPARC is that branch instructions do not incur additional cycles when the branch is taken. Instead the MIPS compiler will, where possible, insert a useful instruction into the branch delay slot which will be executed regardless if the branch is taken of not, and thus will always incur the same number of CPU cycles. For SPARC, the branch delay slot instruction is flushed when the branch is taken and hence performance is reduced. A similar

problem exists in SPARC for load hazards. In this scenario the pipeline is paused until the load hazard no longer exists. On the other hand the MIPS compiler will reorder instructions to avoid such occurrences.

- The MIPS architecture is also not bounded by the complexity that comes from SPARCs use of register windows which adds another dimension to modelling and timing analysis, even though it can help performance for code with many subroutine calls.

- Early and simple incarnations of the SPARC architecture did not include the multiply and divide instructions until the more complex SPARC v8, where as MIPS have had these instructions since the early MIPS R2000.

4.4 Considering existing soft cores

With the reduction in cost and greater popularity of FPGAs against the escalating costs of ASIC designs, there has been an increase in building systems-on-chip with embedded processor cores (Figure 7).

Since this project began there has been a flurry of new soft processor cores which were not previously available. It is therefore appropriate to consider if some of the more recent cores would have been a better starting point.

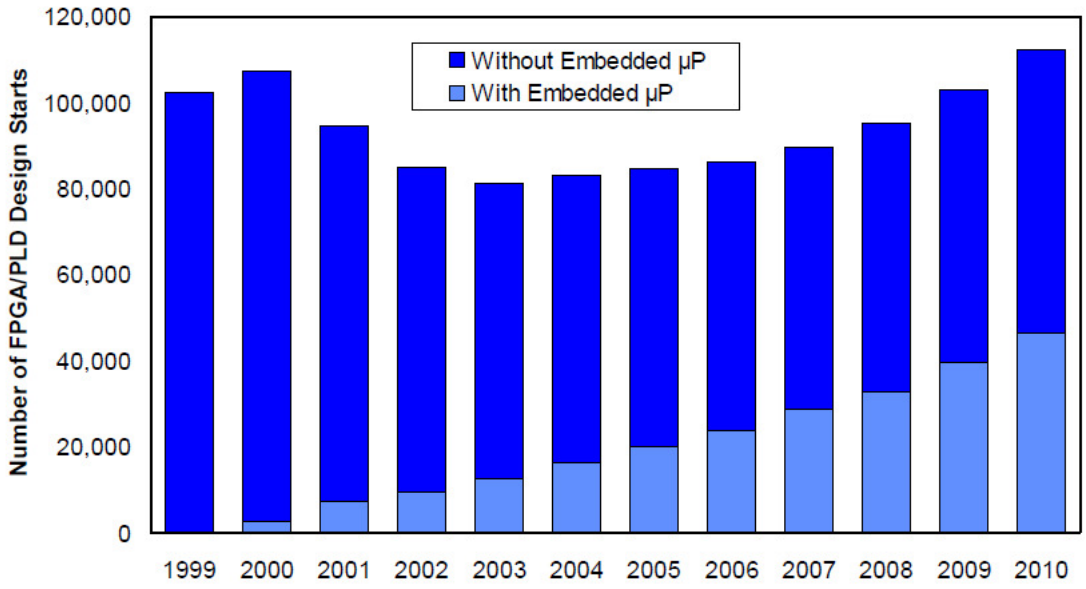


Figure 7: FPGA Design Starts With Embedded µP - Source: Gartner, August 9, 2005

A number of open MIPS like soft cores are based on the DLX architecture which were born out of the books published by Patterson and Hennessey (Patterson and Hennessy 2005) and contains a subset of MIPS instruction set. However, modern DLX compilers are not currently being kept up-to-date. Another incarnation of the DLX architecture is the closed source 32-bit Xilinx Microblaze soft core which includes a number of additional instructions. Whilst there exists an open source Microblaze clone, it was decided that the simplest solution would be to design a soft core based on a very simple, early and unmodified version of the MIPS ISA. For this reason the R2000 ISA was chosen as it includes the multi-cycle multiply and divide instructions which were not present on the DLX.

There are a number of advantages to the MIPS design approach. As its name MIPS (Microprocessor without Interlocked Pipeline Stages) suggests, the aim was to increase dramatically the speed of a processor by the use of deep instruction pipelining whilst reducing interlocks that were required for multi-cycle instructions. The motivation then was that the hardware required to set up these locks were generally large and complicated which had a significant impact on the speed of processors (Hennessy, Jouppi et al. 1982). Therefore Hennessy's approach was to create a simple RISC instruction set by eliminating a number of useful complex instructions such as multiply and divide that take multiple clock cycles to execute, and create an instruction set where all instructions take only one clock cycle (Hennessy, Jouppi et al. 1981). In doing so the pipeline no longer required the complex interlock mechanisms and formed an efficient processor design. Such a design is at the heart of many modern MIPS and RISC processors designs used in areas of research and devices such as Sony Playstations, PDA's and large physics processing computers (MIPS-Technologies 2009).

Whilst the instruction set has a bearing on the architectural characteristics of the hardware, the same ISA can result in a differing number of implementations. Therefore it was decided that the core would be built from the ground up. This would also have the added benefit of being able to fully control and understand all the architectural implementation decisions.

4.5 The PH Processor

For the purposes of this research, the PH processor[3] IP core was create as an implementation of a cut-down version of the original MIPS R2000 processor. The core is compatible with the MIPS I ISA (Instruction Set Architecture) (Kain and Heinrich 1992), except for the few patented non-aligned memory access instructions. The design was based on the architecture and organization outline provided by Patterson and Hennessy (Patterson and Hennessy 2005) and is therefore named the "PH Processor". Briefly, it is a 32-bit processor with 32 registers, a 5-stage pipeline, and separate instruction and data memory banks.

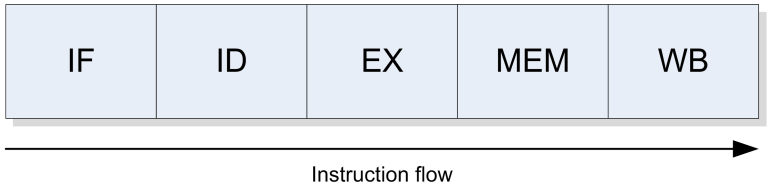


Figure 8: Typical 5-state MIPS pipeline (Patterson and Hennessy 2005)

The 5 pipeline stages of the PH core are shown in Figure 8. These stages execute in parallel and as one instruction finishes in its current stage it moves onto the next stage. The 5-stages of the processor are, the instruction fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM) and Write Back (WB) stages. A more detailed diagram of the control and data paths of the processor is available in Appendix B.

Figure 9 shows an outline of the PH core and peripheral components which was implemented in VHDL on a Xilinx Spartan 3 400 FPGA. The tools used in this process consisted of the Xilinx ISE and Modelsim toolsets.

[3] A version of this PH core has been commercialised through TTE Systems Ltd.

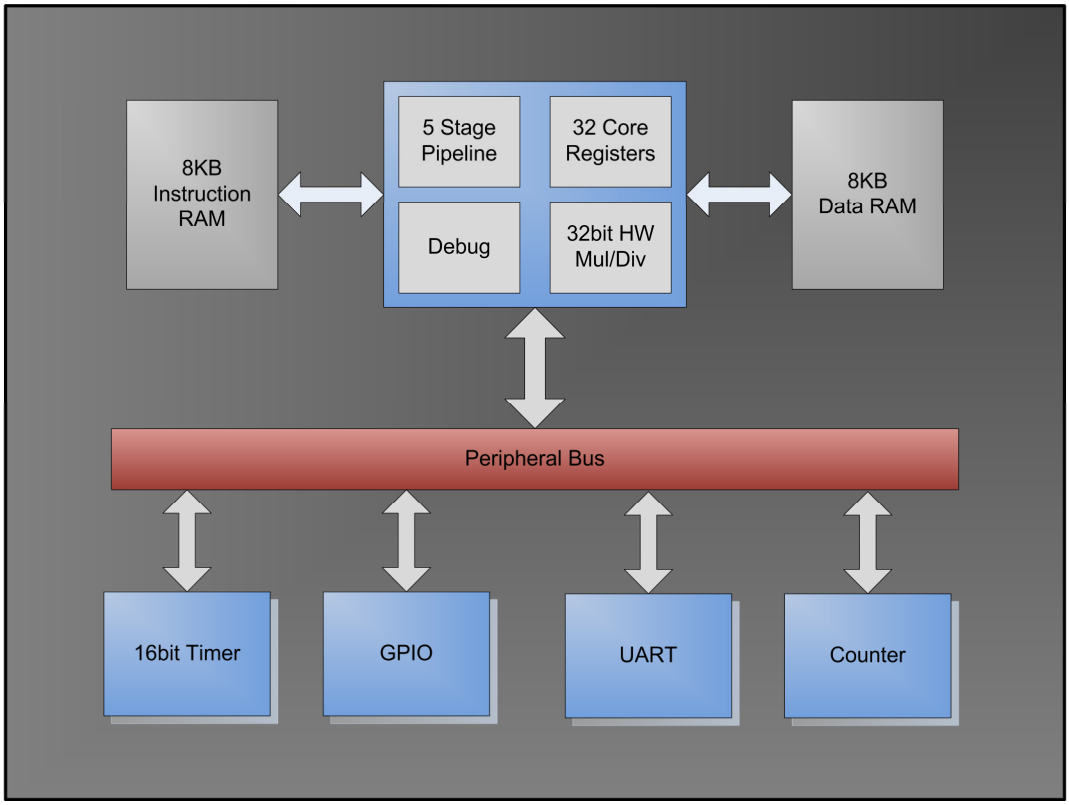


Figure 9: PH processor implementation

Whilst the PH - MIPS R2000 based – processor core retains many of the key features for a predictable processor design as was outlined in Section 4.2, there are still a few adverse side effects in this architecture that need to be resolved. These effects relate to task execution and interrupt latency variation which is made complicated when using a pipelined architecture. The aim is to have static execution times and a predictable and known interrupt overhead. The processor should also have support for at least standard integer multiply and divide operations which can in certain implementations take multiple cycles to complete as these operations can be common in embedded applications. These issues will now be explained in the following Sections.

4.6 Making the PH processor predictable

Often the terms interrupts and exceptions are interchanged in text, however here exceptions will be defined as internal interrupts that occur often due to some sort of hardware malfunction. Interrupts will be used to describe externally triggered hardware interrupts.

It is common in modern architectures to implement precise exceptions. Precise exceptions are defined as a situation where the pipeline can be stopped such that the instructions before the faulting instruction are completed, and those after it are restarted when the handler returns. This has many benefits such as making the debugging of code far easier. Also, paging, virtual memory and IEEE arithmetic handlers strongly motivate the support for precise exceptions (Hennessy and Patterson 2006).

Further details on precise exceptions can be found in Section D.3.5.

4.6.1 Implementing the interrupt system

As highlighted in Section D.3.5, in order to implement precise exceptions - which are beneficial for predictable interrupt handling - a solution can be to service the exception only when the instruction reaches the MEM stage before any state changes are committed. This can be seen in Figure 10 where an exception only gets raised when the instruction enters the memory pipeline stage which causes the instructions in that stage and all the previous stages to be flushed.

When this is implemented, it leads to the problem that the instructions which were flushed before entering into the exception handler must be re-executed on return from interrupt. This in turn means that the interrupted code execution time has been extended by several additional cycles to reload the pipeline.

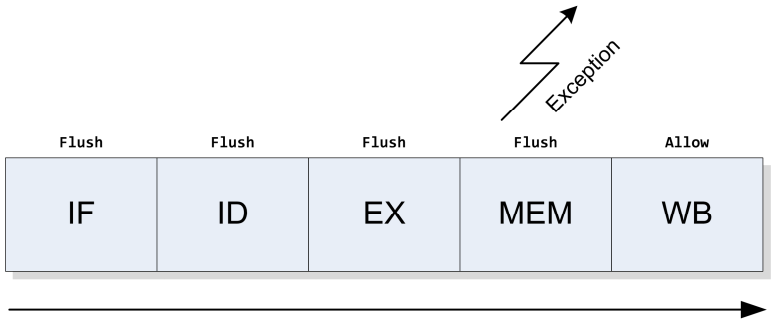


Figure 10: Instructions flushed from 4 pipeline stages when an exception occurs

For error conditions, the additional CPU cycles may be acceptable. Although, ideally for normal timer interrupts there should be no hidden overhead to the interrupted mechanism other than that of the interrupt handler instructions.

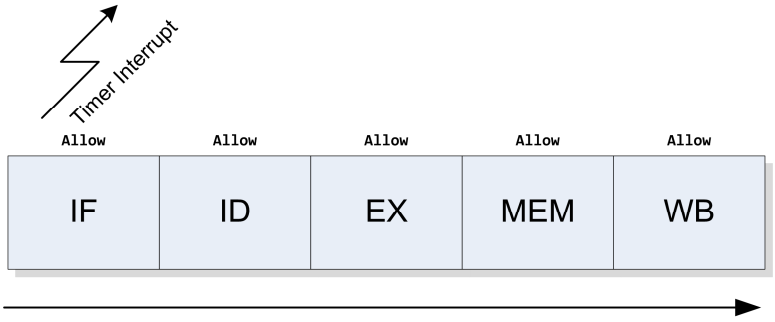


Figure 11: Timer interrupts could be allowed to occur in the first pipeline stage

At first glance, a solution might be to move the timer interrupt handling response to the first pipeline stage and leave the other exceptions to be handled as normal in the later MEM stage. This scenario is depicted in Figure 11 where the timer interrupt will allow the instructions in all the pipeline stages to continue and will redirect the program counter to the interrupt handler address on the next cycle. However, in certain scenarios the timer interrupt may start to be handled either at the same time or before a pending exception has reached the MEM stage, as shown in Figure 12.

Since it is regarded that exceptions are error conditions which in the most case will not be recoverable, this may not be such a serious problem.

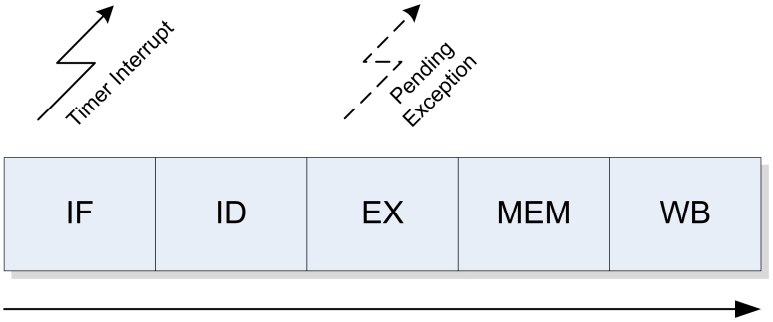


Figure 12: An exception could be pending at the time when the timer interrupt occurs

A problem that is not addressed so far is the scenario when an interrupt occurs on an instruction in the branch delay slot, see Figure 13. The normal behaviour in MIPS is to re-execute the previous branch instruction on return from the interrupt handler in order to re-evaluate if the branch should have been taken. However, this means that the overhead imposed by the interrupt mechanism is no longer constant as there will be cases when an instruction must be executed twice.

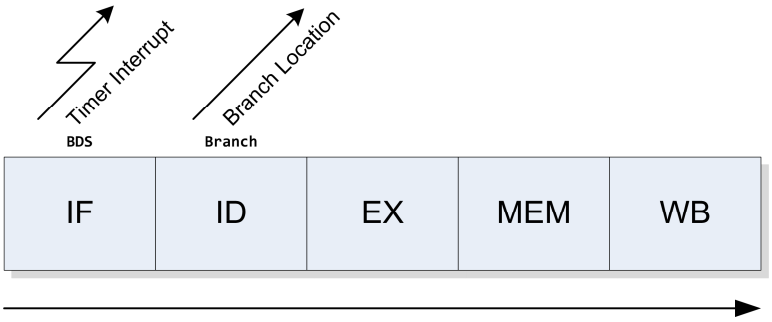


Figure 13: Problems of interrupt on a BDS instruction

A way to avoid re-evaluating the branch condition might be to automatically store the branch condition and the branch address into architectural registers when the interrupt occurs, see Figure 14.

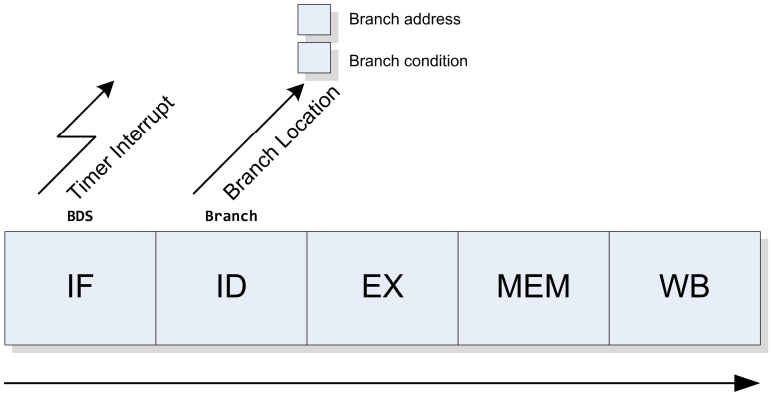


Figure 14: Store branch condition and branch address

Whilst the method above provides a reasonable solution, albeit with some slightly unnatural behaviour when exceptions and interrupts occur at the same time, there is a fundamental problem when it comes to supporting multi-cycle instructions.

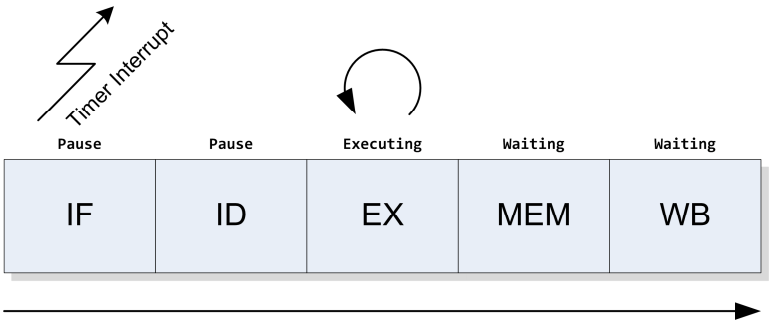


Figure 15: Interrupt paused by multi-cycle instruction

The problem arises when an interrupt occurs whilst a multi-cycle instruction is executing or is about to be executed in the EX stage. This can be seen in Figure 15, where the EX

stage processes for a number of cycles by pausing the previous pipeline stages. Since the interrupt response has been moved from the MEM stage to the IF stage, the interrupt can no longer abort the multi-cycle instruction. Instead, it must wait for the multi-cycle instruction to complete before it can respond and begin loading the interrupt handler instructions into the pipeline.

Both conditions of aborting the currently executing instruction or waiting for the instruction to complete are not ideal. When an instruction is aborted, the processor can respond to an interrupt no variation to the interrupt latency. However, this is at the cost of additional cycles which are passed on to the interrupted code because of the need to reload and re-execute the aborted instructions. In Figure 16 the variation of reloading and executing the flushed instructions is depicted by the time variation between points 'a' and 'b'.

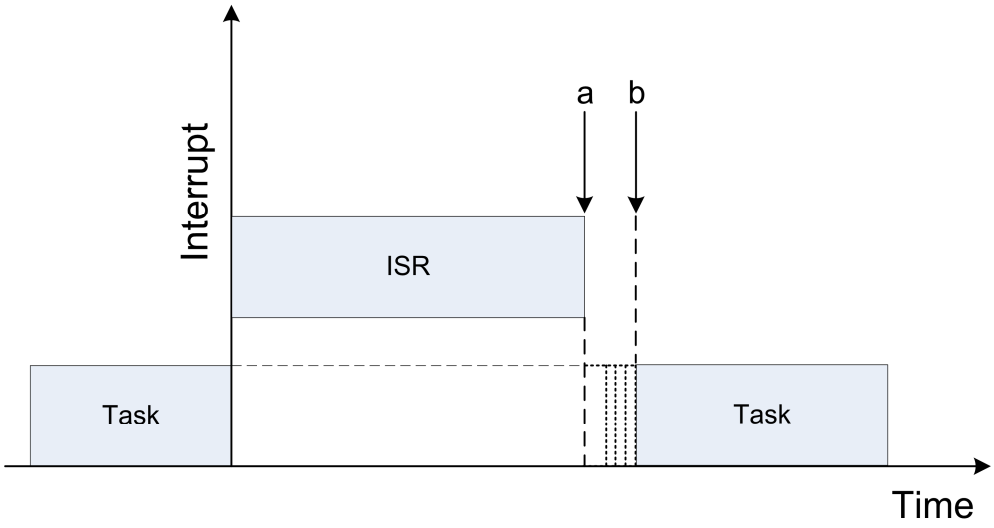


Figure 16: Post interrupt jitter

For the scenario when the interrupt occurs in the IF stage and the instructions which are loaded into the pipeline are no longer aborted, then the interrupt response time can still be variable due to the need to wait for the current instructions in the pipeline to complete. This time may vary depending on which instructions are loaded in the pipeline at that particular time when the interrupt occurred. This is because different instructions may take a different number of clock cycles to execute. This variation of the interrupt latency can be shown by the time between points 'a' and 'b' in Figure 17.

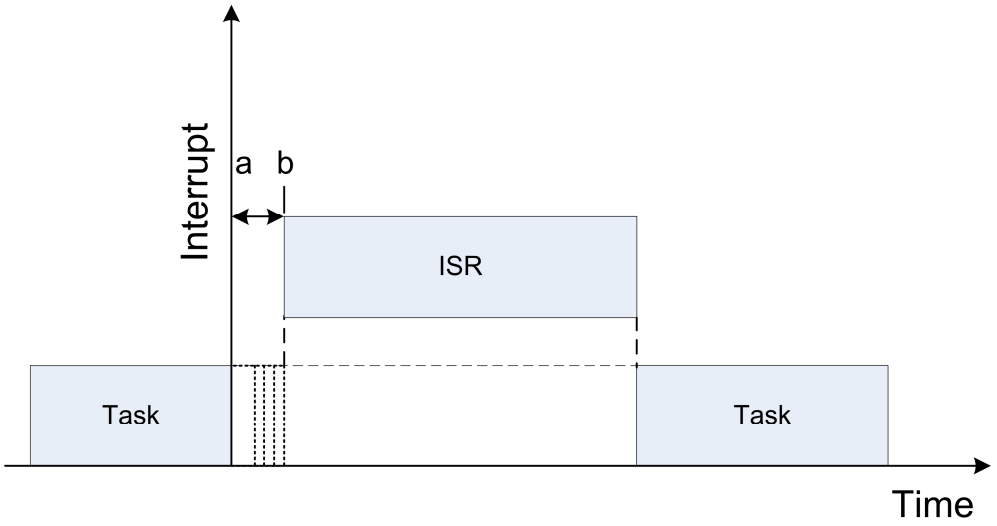


Figure 17: Interrupt latency jitter

Both the scenarios shown in Figure 16 and Figure 17 add a variable amount of additional execution time to the overall task length which is above that of the interrupt handler instructions. Therefore, a solution must be found to implement precise exceptions whist obtaining static interrupt latency time and a constant post interrupt overhead which is imposed on the interrupted task. At the same time, it is important to support the use of commonly used multi-cycle instructions such as multiply and divide as these operations are popular in many embedded systems.

To summarise the modifications required to make the PH processor core predictable, especially when dealing with precise exceptions, are outlined in Table 2.

Requirements
1. **Precise exceptions**
2. **No interrupt latency jitter**
3. **Constant post interrupt overhead**
4. **Support for multi-cycle instructions**

Table 2: Predictable processor requirements

4.6.2 Dealing with Multi-cycle Instructions

For architectures which support integer multiply/divide and floating-point operations, it is generally impractical to create a fully single-cycle design. Whilst it could be possible to accept a slower clock rate or use large amounts of silicon, the normal approach is to support multi-cycle operations (Patterson and Hennessy 2005). The use of multi-cycle instructions is likely to increase as "spare logic" found surrounding intellectual property (IP) cores in FPGAs can be used to implement application-specific operations. For instance, it may be desirable in some applications to include some form of Fast Fourier Transform (FFT) instructions which can execute much faster in hardware than a software library but still require a number of cycles to compute.

The implementation of integer multiply/dividers can vary from completely serial designs normally requiring the number of cycles as bits to multiply, to fully parallel designs which can operate in one cycle. The implementation largely depends on the amount of silicon usage deemed acceptable and the strength of desire for speed. In FPGAs, the implementation might be based on available leftover logic or number of available on chip multiplier blocks.

For simplicity and the purposes of demonstration, the PH processor was designed with a fully serial multiplier/divider which takes 33 cycles to complete an operation. Figure 18 shows the basic outline of the architecture where the values are placed in the multiplicand/divisor and LO register. The result appears in the HI and LO registers after 33 cycles.

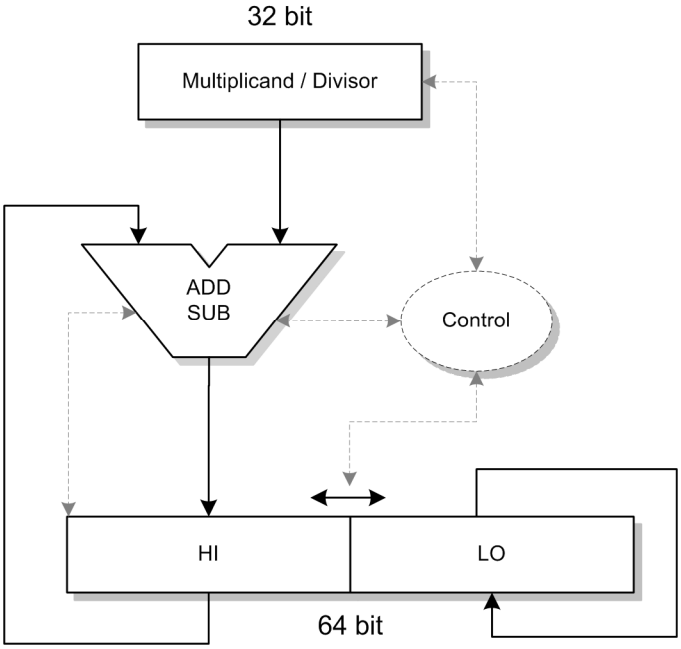


Figure 18: Serial multiply and divide unit

If the processor follows the rule of servicing an interrupt only after the current instruction has completed, then in the case of interrupting the MUL or DIV operation, there can be a significant amount of jitter from the latency varying between 1 and 33 cycles. The amount will depend on how much of the operation is left at the time when the interrupt occurs.

In an attempt to solve this problem there are a number of solutions that could potentially be considered.

4.6.2.1 Abort the instruction

As mentioned previously, one way to get round the problem of interrupt response jitter is to abort the currently executing instruction mid execution and restart the instruction on return from the interrupt. This has the benefit of eliminating the response time jitter but at the cost of shifting that jitter to the interrupted task.

There are also situations where implementing this type of solution is not possible due to instructions that modify the processor state during execution and therefore the changes cannot be undone if aborted. If this occurs, certain registers or memory locations may become corrupt.

In addition, some memory bus systems may not allow an instruction to be aborted once a memory request has already been started.

4.6.2.2 Run in parallel

Another method might be to service the interrupt immediately, but instead of aborting the instruction, it could be allowed to complete in parallel (Figure 19). In certain situations, this results in speeding up the interrupted code execution time. The increase in speed is not desirable for predictability where the interrupted code is impacted in a positive way but by an unknown quantity.

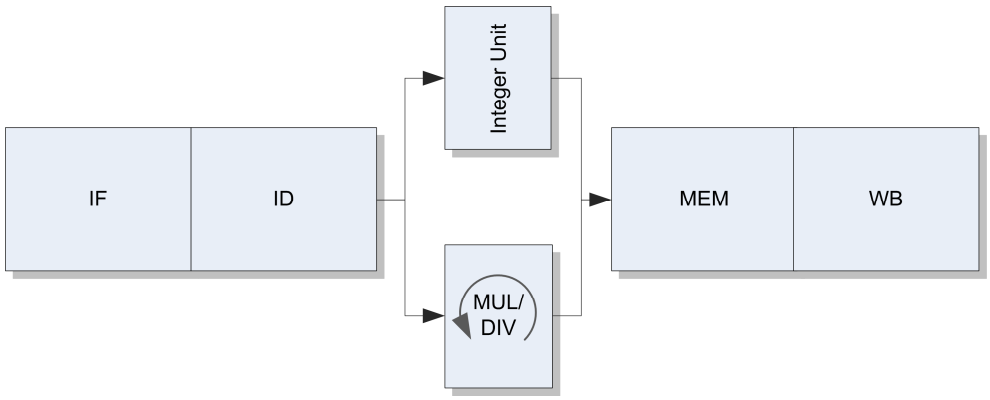


Figure 19: Multi-cycle instructions running in parallel to integer instructions

This method also runs on the assumption that handler code will not require use of the multi-cycle execution unit before the instruction has had time to complete. In the scenario when the execution unit is required, a resource conflict arises requiring the handler to stall until the current multi-cycle operation has completed. The system also needs to ensure that the previous result values in the execution unit remain present on return from interrupt.

4.6.2.3 Execute if time

Another method might be to look ahead to an earlier stage in the pipeline and prevent the multi-cycle operation from being executed when there are less than the required cycles until the timer overflows (Figure 20). This would give zero jitter to servicing the interrupt but consequently add additional cycles to the interrupted code leading to the same effect as the instruction abort method. It also relies on there being a timer which is driving the interrupt signal and its registers are accessible to the processor control unit.

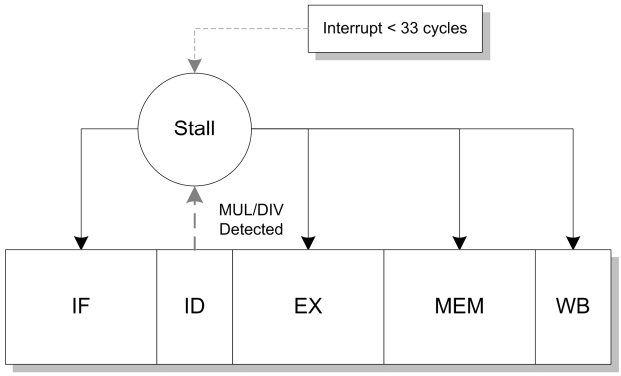


Figure 20: Stall the pipeline when there is no time to complete a multi-cycle instruction.

4.6.2.4 Fixed maximum interrupt latency

A similar solution might be to always incur the maximum delay from when the interrupt is generated to the point when it gets serviced by using a similar technique to code balancing. This technique is far from efficient and not easily expandable as it requires knowledge of the worst case execution time of the included execution units. In some processors, additional instructions are added as external or on-chip co-processors, therefore the WCET instruction time might not be known at processor design time. However, this method does solve the problem of interrupt response jitter and impacts the interrupted code by a fixed amount.

4.6.2.5 Multithreaded pipeline

While the previous methods have found solutions to some of the problems, only the fixed maximum interrupt latency method provides a solution to solve both variable interrupt response and task overhead latencies but at a significant cost to performance. There is also the issue of dealing with the variability of interrupting on a BDS instruction.

One of the modern techniques for increasing performance by making greater use of silicon rather than an increase in core frequency is the design of multi-threaded cores (Gulati and Bagherzadeh 1996). The idea is to speed up multithreaded systems by providing a way in which the processor can change task contexts at the instruction level. This is implemented in hardware with very little switchover time. Each thread is given its own register file and instruction counters, but shares the pipeline and execution units.

The uses of multithreaded pipelines are common in modern desktop processors such as Intel's Hyperthreading technology (Marr, Binns et al. 2002). They are also beginning to be

implemented in various new embedded processors such as the MIPS MT core (Kissell 2008).

One of the benefits of using a multithreaded core is that it can avoid the performance problems from various pipeline hazards by automatically switching to another thread until the pipeline hazard for the current thread has been resolved. As a result, maximum pipeline usage can be obtained in all pipeline stages. According to Intel, this can result in a 30% increase on Pentium processors (Bulpin and Pratt 2004).

Whilst the intention is to seek performance through greater pipeline utilization and support fast context switching times between threads, a similar idea could be used for increasing predictability of switching between a single task and interrupt handler.

The idea would be to allow the interrupt to be handled in the MEM stage and in addition to the duplicate instruction counter and register file the pipeline stages would also be mirrored. The mirroring of the pipeline stages would avoid the need for flushing pipeline stages during an interrupt. Instructions such as branch could then remain in the pipeline and continue execution on return without the need for re-execution. In essence, one pipeline would be used for normal task execution whilst the other is used for the interrupt handler (Figure 21). The result, is that the interrupted task could be paused so that the interrupt handler could run immediately and the task could resume execution at the exact point where the interrupt occurred leaving the task pipeline stages intact. This is generally only possible for time-triggered type systems where there is only one interrupt, as supporting multilevel or nested interrupts would require duplicate numbers of multithreaded pipelines.

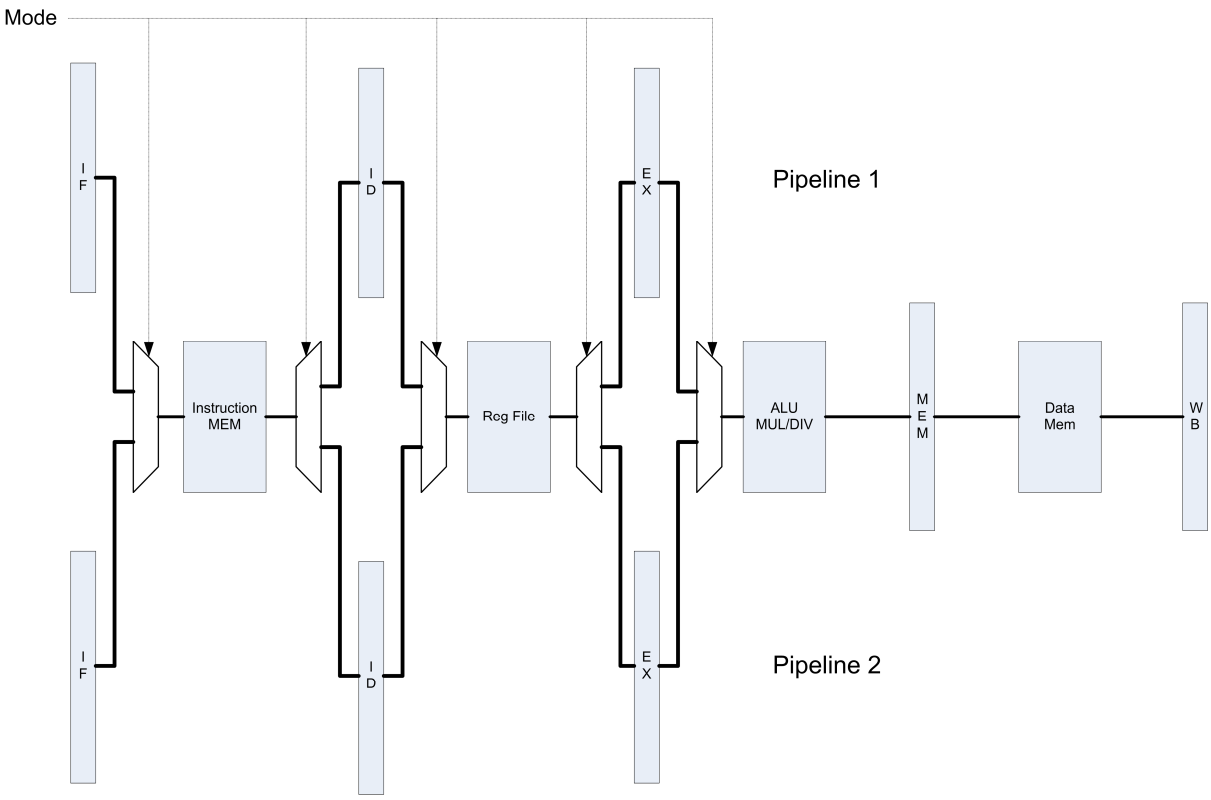


Figure 21: Predictable processor pipeline

Whilst the solution maybe more silicon intensive compared to the previous methods, it does solve the problem of response and overhead jitter and removes the variation of interrupting on a BDS instruction. The method therefore reduces the complexity of factoring interrupt overheads to timing analysis. However, to enable this technique to work for multi-cycle instructions the additional execution units must also have a way of pausing and resuming their state. This can be achieved by implementing shadow registers which are switched in when the processor enters and returns from the interrupt thread (Figure 22).

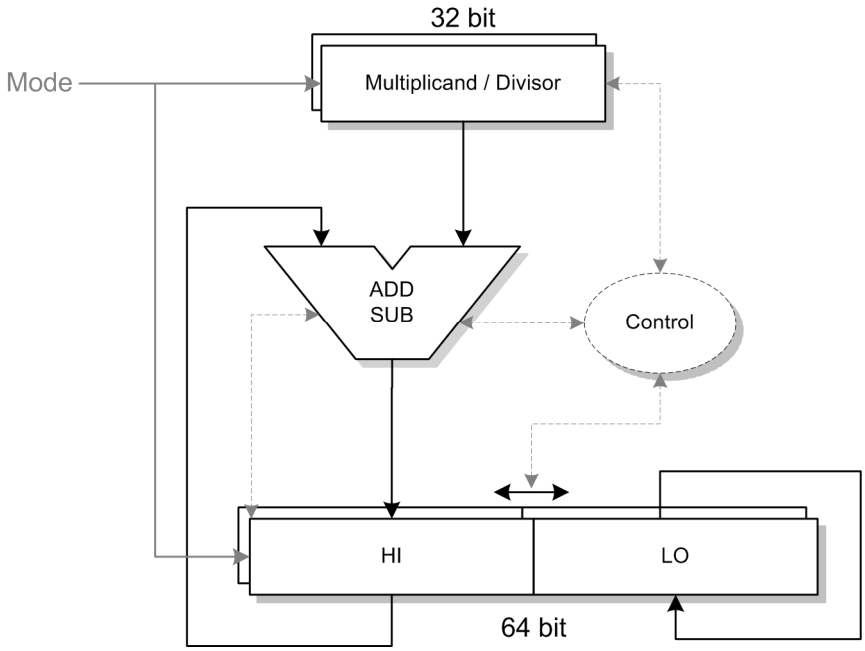


Figure 22: Multiply and Divide unit with controllable shadow registers

In summary, through selective design choices and the techniques described above a predictable pipelined processor with reasonable performance can be achieved. Some sacrifices have been made in performance by abstaining from ILP, caches and branch prediction. Also, silicon usage has been increased through the implementation of a multithreaded like system. However, it is believed that the performance of the processor will still be respectable when compared with many common embedded COTS processors.

4.7 Making the processor TT

The interest throughout this thesis has been on the development of a processor which is not only predictable but also supports time-triggered (TT) architectures.

Currently, many generic COTS processors support a wide range of interrupt sources, while the use of a (pure) time-triggered software architecture generally requires that only a single interrupt be supported by each processor. This then leads to software design 'guidelines', like the 'one interrupt per micro-controller rule' (Pont 2001). Such guidelines can be adhered to by the use of appropriate tools in software creation. However, it is possible for changes to be made in the software (for example, during software maintenance or upgrades) that lead to the creation of unreliable systems.

More specifically, it is possible that developers of time-triggered software designs (or people who subsequently maintain or upgrade systems based on time-triggered software designs) may be unaware of the need to employ only a single interrupt source with such designs. If (as a result of this lack of knowledge or lack of experience) an attempt is made to use multiple interrupts in such a system, then this may lead to highly 'unpredictable' behaviour. The aim is therefore to improve the processor hardware in supporting time-triggered software by removing the chance and possibility for more than one interrupt being enabled. Note, that the use of general software exceptions is not appropriate in such a design: instead, only serious faults (such as page faults, arithmetic errors and undefined instructions) should be handled in this manner.

The operation of a time-triggered system generally operates by a source of periodic "ticks" to which the processor will schedule one or more periodic tasks to execute, see Section 2.3.1. The design should allow for the tick interrupts to be obtained from only 1 of a number of different sources, depending on the nature of the system (see Figure 23).

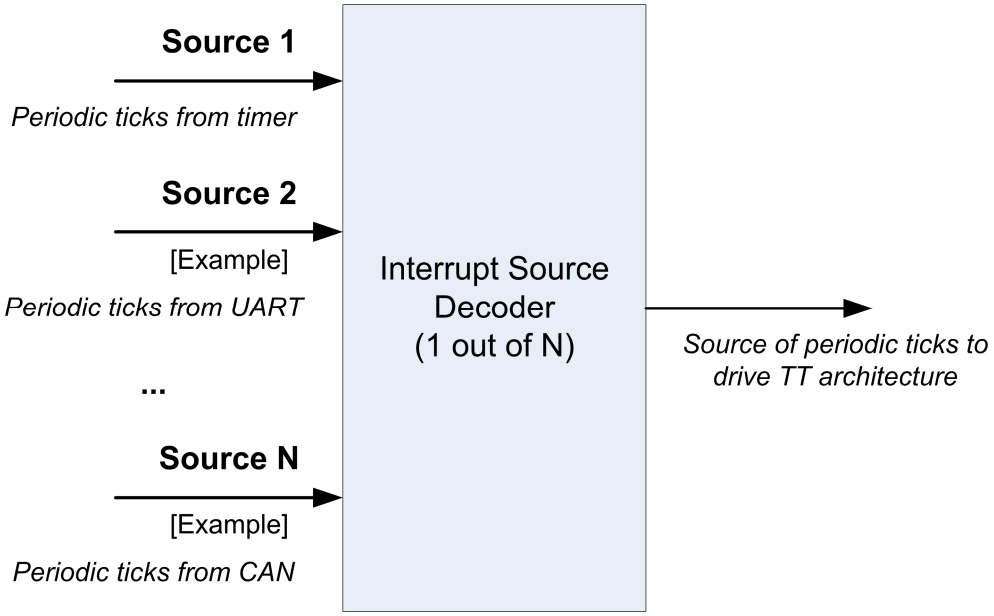


Figure 23: Interrupt source selector

For single processor designs, the ticks will generally be derived from an on-chip timer. However, in multiprocessor designs, the ticks may be derived from the arrival of messages from a suitable communication bus (e.g. CAN bus, UART). As a failsafe option, it may be necessary for the system to be able to switch interrupt sources on-the-fly, to for instance a

backup CAN bus. The hardware must then ensure that only a single source of interrupts is active at any one time during the operation of the system.

Interrupt sources which are not used to generate periodic system ticks can be monitored by polling (under the control of the system scheduler) at an appropriate rate.

4.7.1 Detailed Description

By restricting the number of interrupt sources (to 1) in a TT embedded system, the "one interrupt per microcontroller" design guideline can be met and thus the system behaviour can be made more predictable.

The modifications are implemented through the standard MIPS co-processor zero registers which are ordinarily used for configuring interrupts. The coprocessor unit is part of the internal processor core and has among others, the following registers derived from MIPS (Table 3).

Register Number	Register Name
R12	Status
R13	Cause
R14	Exception Program Counter (EPC)
R15	Processor ID (PRId)
R16	Config

Table 3: PH co-processor zero registers[4]

The coprocessor is setup to be similar to the MIPS convention in that the registers have the same name and that they perform similar operations to those found in a MIPS processor. However the structure and operation of these registers are slightly different. For instance, the status and cause registers which deal with interrupts are modified so that only the interrupt source number is held in the interrupt mask register rather than using

[4] The EPC and PRId registers are the same as in MIPS. However the PRId register contains version numbers pertaining to the PH processor. The Config register has a low-power bit (only) which is used to place the processor in idle/sleep mode.

individual bits to represent each source. Also the status register includes a global interrupt enable bit which allows the single interrupt source to interrupt the processor (Figure 24).

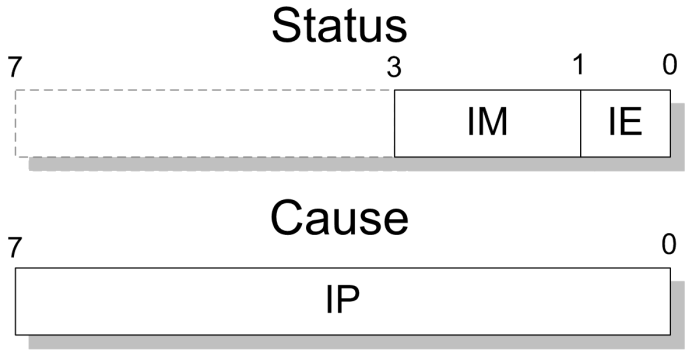


Figure 24: Co-processor status and cause registers

To enforce the one interrupt rule the hardware is made using a simple 3 to 8bit decoder (for 8 interrupt sources) which can never have more than one output bit enabled at any time (Figure 25). The IM (Interrupt Mask) register holds the binary number of the enabled interrupt source. Since there are 8 possible interrupt sources the IM register is only 3 bits wide in the current implementation.

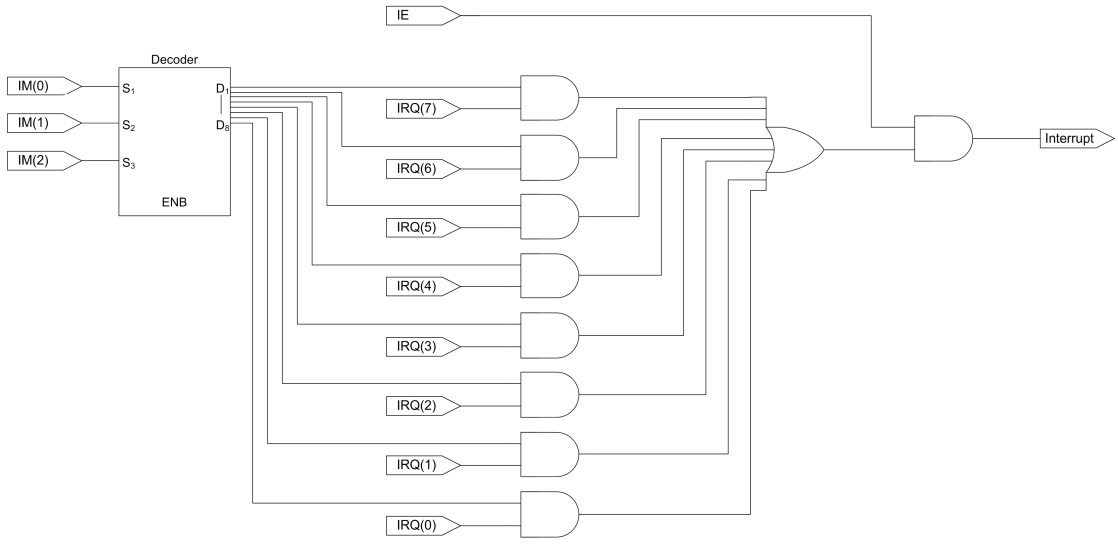


Figure 25: Interrupt source selector

The hardware takes the binary number from the IM register and runs it through a 3 to 8 bit decoder. The incoming interrupt sources are AND together with the decoder output, allowing only the relevant bit to pass through. The final output is then AND with the IE (interrupt enable) bit before being allowed to interrupt the processor core.

In addition to the above circuit, the interrupt sources will also be flagged in the cause register where each bit will represent an interrupt source. Polling applications can then monitor these bits and reset the individual flags by writing a 1 to the relevant bit.

Whilst the implementation is rather simplistic, it is very effective in preventing the possibility of enabling more than one interrupt source.

4.8 Results

In this section results were obtained to show the interrupt latency jitter and the interrupt overhead time for two versions of the PH core. By way of comparison, the first core is setup to represent a COTS microprocessor, which in this scenario will abort the current instruction on interrupt and enter into the interrupt handler immediately. The second – multi-pipelined – core implements the architecture described throughout Section 4.6.2.5, which on interrupt, will pause the current pipeline state and return directly to that state after the handler code has completed. For ease of reference, the first core will be known as the 'PH' core whilst the second will be referred to as the 'PH-Predictable' core.

The aim of the results will be to demonstrate that it is easier for a programmer to predict how long a particular piece of code will take to run on the PH-Predictable core. More specifically this will include the scenario when that code is interrupted by a simple interrupt handler. To achieve this objective, a test bed was setup as follows.

4.8.1 Test Case 1 Strategy

Each core will boot up and run its initialisation code, once done, the core will execute a set of instructions repeatedly in a loop. An example of this can be seen in Listing 1. Within this loop there is a single cycle 'NOP' instruction as well as a multiply and divide instruction as these are the longest running instructions on the core. On the both cores the multiply and divide instructions can consume 33 CPU clock cycles per calculation.

```
int main ()
   {
   // Set up IO
   LED_Init();
   SEG_Init();

   while (1) // Super Loop
      {
         // Set output pin to low
         GPO_pin = 0;

         // Multiply 2 * 3
         asm("li $2,2\n"
             "li $3,3\n"
             "multu $2,$3"
               :::"$2","$3");

         // Divide 6 / 2
         asm("li $2,6\n"
             "li $3,2\n"
             "divu $2,$3"
               :::"$2","$3");

         // No operation
         asm("nop");
      }

   // Should never reach here ...
   return 0;
   }
```

Listing 1: Multi-cycle instructions under test for test 1

The first test will be to measure the time between an interrupt signal being triggered and the core entering and executing the interrupt handler code.

To achieve this objective, an interrupt mechanism had to be devised to ensure that different instructions in the loop would get interrupted, rather than the same instruction consistently. Therefore, the period between consecutive interrupts should observe a random like behaviour. To keep the implementation simple, interrupts would be generated by an external source which would be driven by a timer whose timeout values will be based on the first 4000 digits of PI. On each timeout, the next digit in the table will be loaded into the timer for the next interrupt.

```
// Change timer match register to a number
// based on the next digit of PI
TMR0TMAT = (1000-5) + digitsOfPi[pIndex];
```

Listing 2: Setting of the variable timer timeout values

The timeout values were based around a 1 millisecond interval as interrupting too frequently would make it hard to record measurements and running slower would provide no benefit and consume unnecessary time.

At first glance it may seem rather ineffective to vary each timer timeout value between 995 and 1004 microseconds, however, the timing effect is accumulative. For instance, after 50 timeouts the time could vary as much as 49.75 milliseconds to 50.2 milliseconds which could amount to a difference of up to 11,250 CPU clock cycles at 25MHz. It is recognised that this mechanism may not necessarily provide truly random behaviour; however, it provides enough variation to ensure that various instructions in the loop will be interrupted.

Having created a means to generate interrupts at different points in time, the recorded measurements must be taken from when the external interrupt source goes high and until the interrupt handler on the core has begun executing (see Figure 17). This could be achieved by using an external hardware counter which counts at a rate of a 100MHz. This is equivalent to 4 clock cycles for each a CPU clock cycle with a core frequency of 25MHz. The counter will be started as soon as the external interrupt source is active and stopped when a GPIO signal received from the target core becomes active. This GPIO signal is set high immediately at the beginning of the interrupt handler code (Listing 3).

```
handler:
      # Set output pin to high
      li    $26, 0x00030008
      li    $27, 1
      sw    $27, 0($26)

      # Read CP0 EPC return address Reg
      mfc0  $26, $14

      # Enable Interrupts
      li    $27, 0x0F
      mtc0  $27, $12

      # Return from handler
      j     $26
      nop
```

Listing 3: Interrupt handler code for test 1

4.8.2 PH Core – Test Case 1

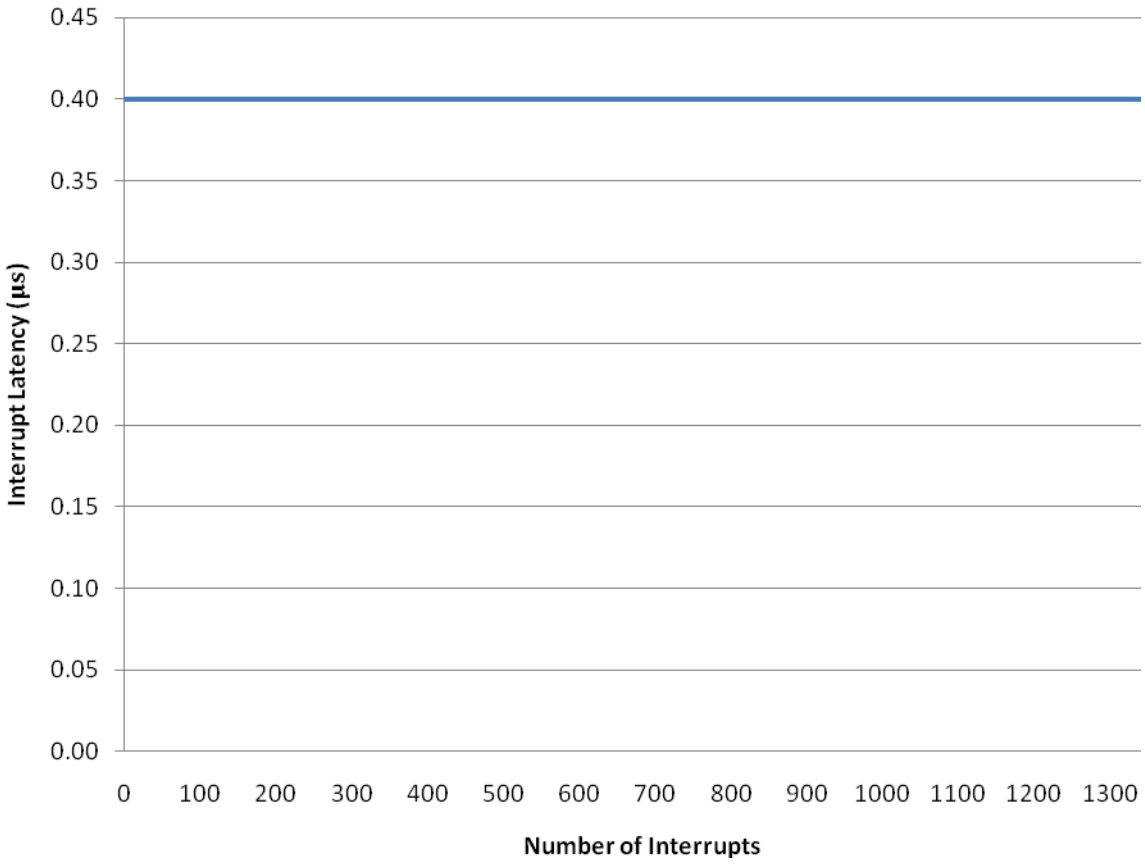


Figure 26: The PH processors interrupt latency over a mixture of MULT, DIV and NOP instructions

When recording the interrupt latency for a number of interrupts, the standard PH core was observed to have a fixed latency of 0.4µs (Figure 26). This amounts to 10 CPU cycles and can be broken down as shown in Table 4.

Event	Instructions	CPU Cycle Count
Internal Response Time		1
Pipeline Flush		3
Interrupt Vector	`j handler` `nop`	2

Interrupt Handler to write monitoring pin	`lui $26, 0x03`
	`ori $26, 0x08`
	`li $27, 1`
	`sw $27, 0($26)`
		4

	Total	10

Table 4: Measured interrupt latency time breakdown

For greater clarification, the latency breakdown can be explained as follows. The time from the external interrupt signal going high and the system co-processor 0 latching the interrupt state is one clock cycle. For reference, on MIPS based cores the co-processor 0 contains and deals with interrupt enable and status flags and thus the interrupt passes through the co-processor 0 before interrupting the main CPU core. When the CPU core is interrupted, the three instructions in the IF, ID and EX stages are flushed. Therefore, it takes 3 clock cycles before the first interrupt handler instruction reaches the end of the execution stage and begins to do some work. The first two instructions that are executed come from the interrupt vector address which contains a jump to the interrupt handler and its subsequent branch delay slot. At the beginning of the interrupt handler, the first four instructions deal with setting the GPIO pin high which is used to stop the external counter.

To ensure the explanation of the interrupt latency time was correct, the processor was simulated in Modelsim which is an industry standard tool for hardware for simulating hardware designs (Mentor Graphics 2010). The simulations were undertaken using a test bench file which loads the processor with the same binary code file as was used in the real world tests. Figure 27 shows the time taken from sensing an external interrupt signal going high to then setting a GPIO pin high.

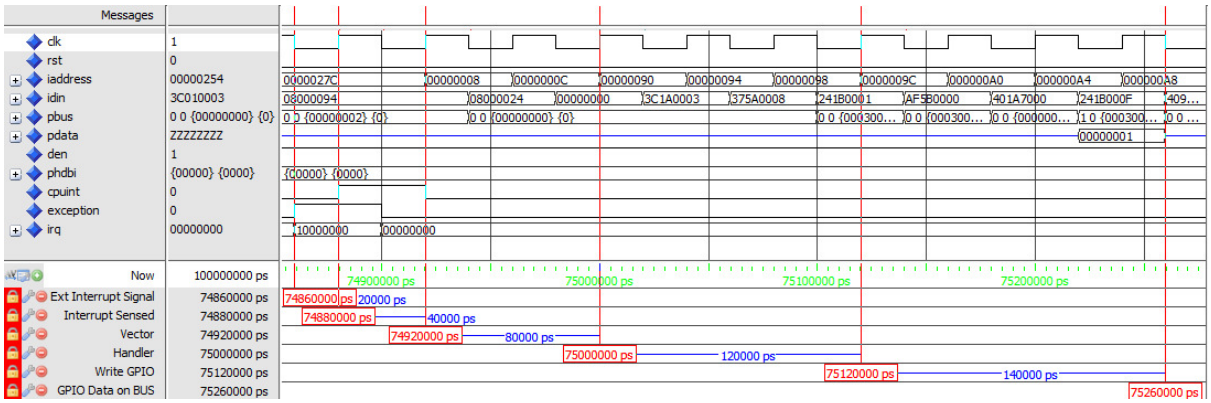


Figure 27: PH Core interrupt latency simulation[5]

From the first test it can be seen that the standard PH processor has a static zero jitter interrupt response time due to its ability to abort the currently running instructions. This already appears promising for the development of a predictable processor. However, the

[5] A larger version of this diagram is available in Appendix C, Figure 84.

next test will look at what the potential side effects of aborting the instruction might be when considering the overhead imposed on the task that has been interrupted.

4.8.3 Test Case 2 Strategy

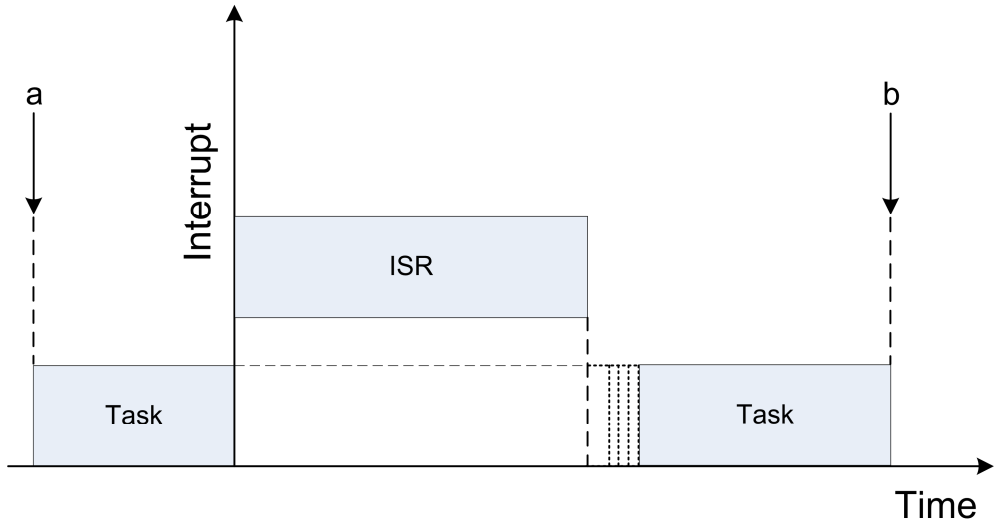


Figure 28: Timing measurements for test 2

The next test is similar to the first test apart from some minor changes. The most significant change is that the time measured by the external counter is now set to between the start of one iteration of the instruction loop and the end of that iteration (Figure 28). The purpose of this was to make the instruction loop represent a task that would be interrupted occasionally. The measurements would then record the execution time of that task. However, in order to make the main instruction loop execute for a duration large enough to get significant measurements, an internal loop was added to re-execute the set of assembly instructions (Listing 4).

```
        while (1) // Super Loop
              {
              // Set output pin 1 to high
              GPO_pin = 1;

              for(i=0; i<2048; i++)
                    {
                    // Multiply 2 * 3
                    asm("li $2,2\n"
                          "li $3,3\n"
                          "multu $2,$3"
                                :::"$2","$3");

                    // Divide 6 / 2
                    asm("li $2,6\n"
                          "li $3,2\n"
                          "divu $2,$3"
                                :::"$2","$3");

                    // No operation
                    asm("nop");
                    }

              // Set output pin 2 to high
              GPO_pin = 2;

}
```

Listing 4: Multi-cycle instructions under test for test 2

The job of the interrupt handler was then modified so that it simply re-enabled the interrupts and returned to the interrupted code. Since the handler doesn't do much, it wasn't necessary to save and restore registers as would normally be done in most interrupt handlers. This would also make understanding the interrupt overhead a bit easier.

```
handler:
      # Read CP0 EPC return address Reg
      mfc0  $26, $14

      # Enable Interrupts
      li    $27, 0x0F
      mtc0  $27, $12

      # Return from handler
      j     $26
      nop
```

Listing 5: Interrupt handler code for test 2

4.8.4 PH Core – Test Case 2

When running the system on hardware, the task execution time varied quite a bit (Figure 29).

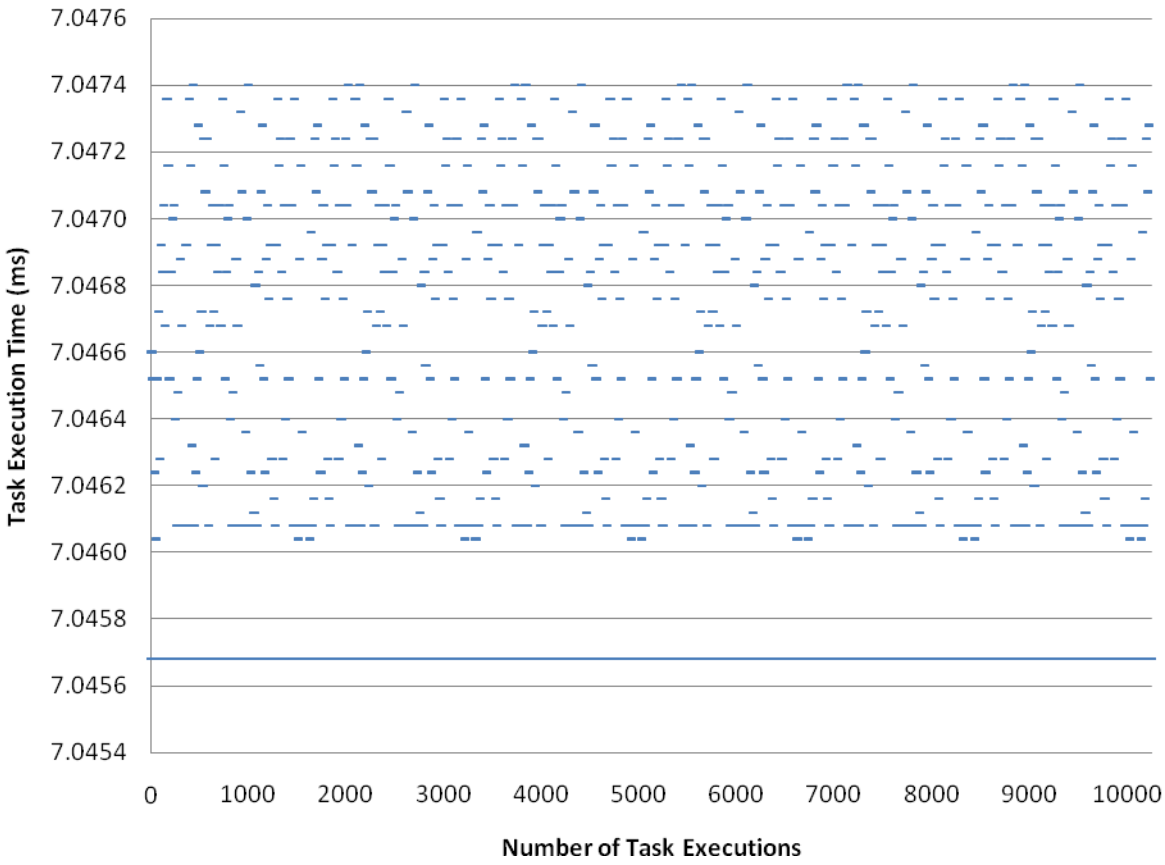


Figure 29: PH Core task execution time

Looking at Figure 29 a steady line can be seen at the lower end of the graph. This line represents the base task execution time when the task has not been interrupted. From this base line there is a gap of 10 CPU cycles which represents the interrupt handler execution time. Above the additional 10 CPU cycle interrupt handler time there are a number of varied measurements which peak upto a 43 CPU cycle time above the base uninterrupted task execution time (Table 5).

	Execution Time (ms)	CPU Cycles
Max	7.04741	176185
Min	7.04569	176142
Difference	0.00168	43

Table 5: PH Core task execution maximum and minimum times

It can therefore be seen that the standard PH core with its instruction abort mechanism provides a stable low jitter interrupt latency time. However, the jitter that would normally come from waiting for the current instruction to complete is replaced by jitter due to the need to re-execute any instructions that have been aborted. In effect, the instruction abort

design gives the interrupt handler a higher priority over the currently executing task by running immediately, however, the overhead imposed on the interrupted task is still variable. In some ways this design might lead to a misunderstanding of the interrupt overhead if not fully understood by the application designer who may be concerned about timing effects withing their system.

Event	Instructions	CPU Cycle Count
Pipeline Flush		**3**
Interrupt Vector	`j handler` `nop`	**2**

Interrupt Handler	`mfc0 $26,$14`
	
	`# Enable Interrupts`
	`li $27,0x0F`
	`mtc0 $27,$12`
	
	`# Return from handler`
	`j $26`
	`nop`
		5

Max Instruction cycle time	`MUL or DIV`	**33**
	Total	**43**

Table 6: PH Core task execution overhead breakdown

Table 6 gives a breakdown of the overhead that was observed in the measurements in Figure 29. However, to give more justification to where these results may have come from, the next few figures were obtained from Modelsim simulations of the PH core using the same code that was used in the tests.

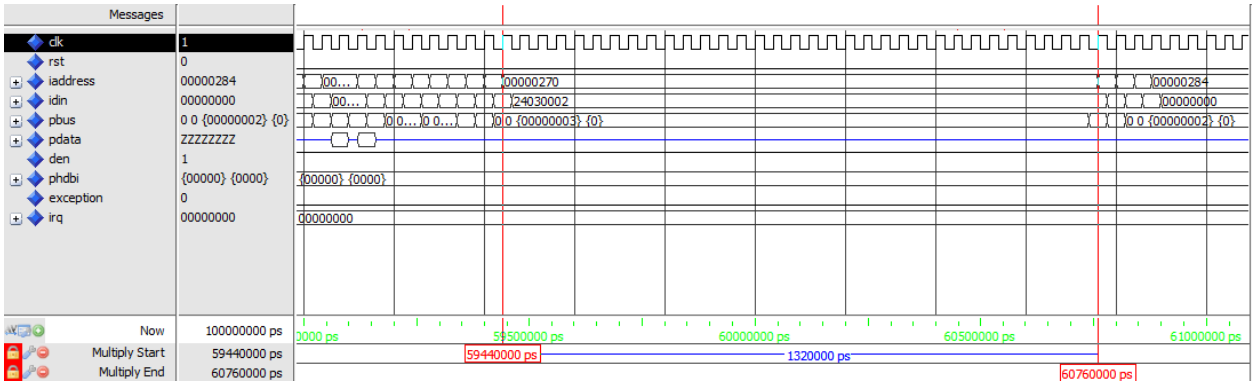


Figure 30: Multiply instruction (33 CPU clock cycles)[6]

Figure 30 demonstrates the execution time for a multiply instruction consumes 33 CPU cycles. This execution time is the same regardless of the size of the operands which in

[6] A larger version of this diagram is available in Appendix C, Figure 85.

other processors can lead to variable computational times. The PH core is also different to some standard MIPS cores in that multiply and divide instructions do not execute in parallel to normal integer instructions, but rather execute sequentially.

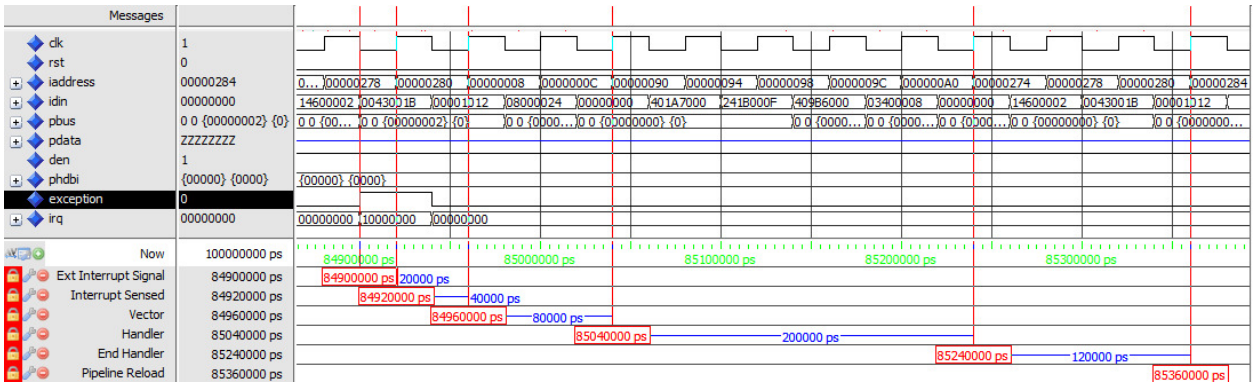


Figure 31: Interrupt Overhead (10 CPU clock cycles)[7]

Figure 31 shows the overhead of the interrupt mechanism and the interrupt handler code. It is worth noting that the delay of sensing the interrupt signal and latching it in the co-processor 0 does not affect the interrupt overhead since the interrupt is sensed in parallel to the pipeline. Therefore the interrupt overhead time starts from when the vector is executed.

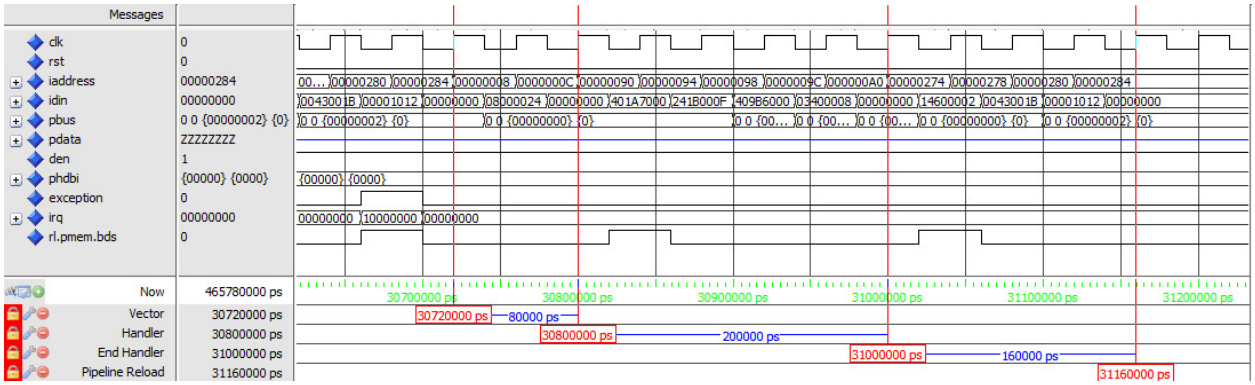


Figure 32: Interrupt on a Branch Delay Slot instruction (11 CPU clock cycles)[8]

Although not represented in the measurements, there is a further variation to the interrupt overhead which can come from interrupting on a branch delay slot. If such a condition should occur, then the previous branch instruction must be executed again in order to re-

[7] A larger version of this diagram is available in Appendix C, Figure 86.

[8] A larger version of this diagram is available in Appendix C, Figure 87.

evaluate if the branch should have been taken or not. This would result in the overhead time consuming a futher CPU cycle as can be shown in Figure 32.

In summary, the modelsim figures show that an interrupt overhead time of 10 CPU cycles could be expected with the potential need to re-execute a 33 CPU cycle multiply or divide instruction which would consume upto 43 CPU cycles. The worst case scenario would be if the multiply or divide instruction occured in a branch delay slot where a maximum overhead of 44 CPU cycles would be observed. If the interrupt handler code is not included, then the processor could be said to have an interrupt overhead of 5 to 38 CPU cycles.

In many applications, a varation as small as this may not seem a big concern. However, since this thesis is concerned with making a predictable processor down to the instruction level, then the next set of results show how the PH-Predictable core differs from the standard PH core.

4.8.5 PH-Predictable Core - Test Case 1

As in the standard PH Core, the first test was to measure the interrupt latency between an external interrupt signal going high and the interrupt handler responding by setting a GPIO pin high. As in the standard PH Core, the PH-Predictable core had no jitter in the interrupt latency time (Figure 33). Furthermore, the duration of the interrupt latency was also exactly the same. It may therefore appear that the PH-Predicatable core offers no real speed increase over the standard PH Core. However, for a normal interrupt handler in the standard PH Core many of the registers would have to be saved on the stack. Alternatively, the PH-Predictable core has two register files and therefore avoids the need to execute these additional instructions to save and restore the interrupt context.

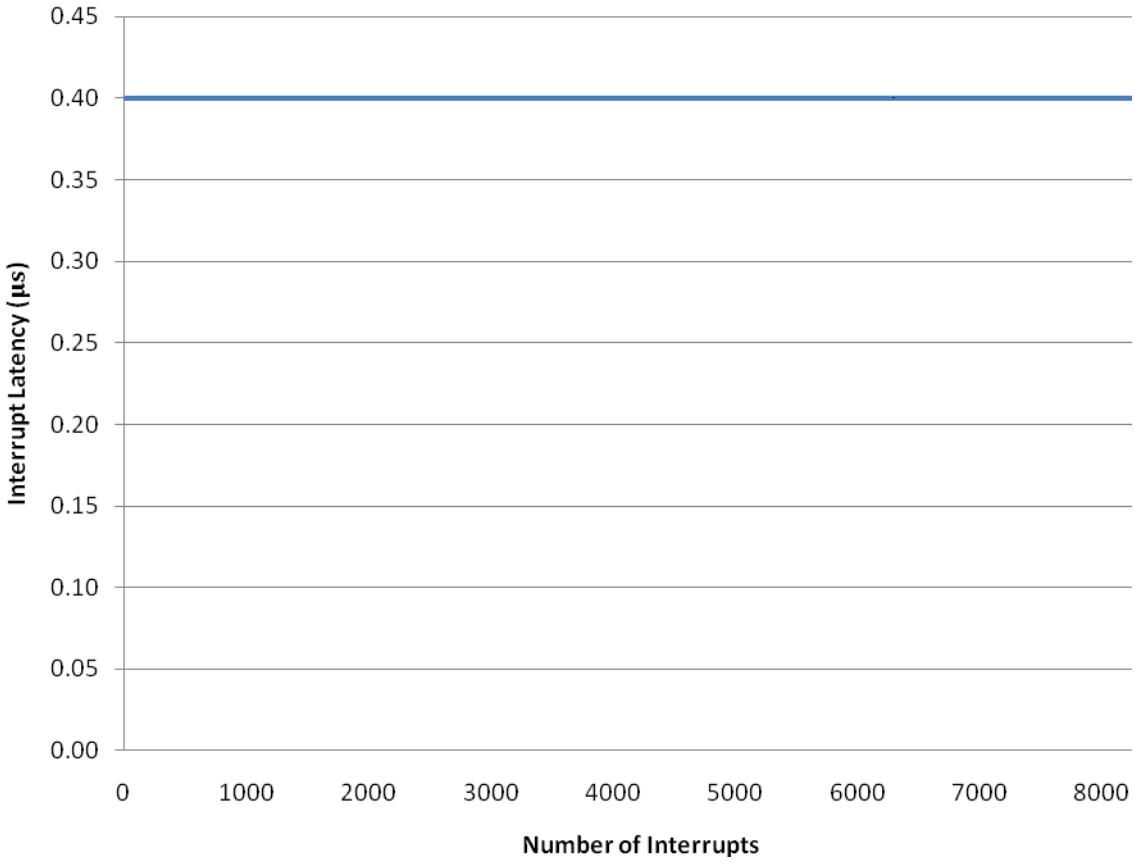


Figure 33: PH-Predictable core interrupt latency

The interrupt latency time can be broken down to the same actions as in the standard PH core. First there is a single clock cycle delay as the interrupt signal is latched by the co-processor 0. Although the pipeline is not flushed, 3 clock cycles are required to load the interrupt handler instructions to the EX stage where they begin to do some work. The interrupt vector code then consumes 2 CPU cycles followed by 4 CPU cycles for writing to the GPIO peripheral.

Event	Instructions	CPU Cycle Count
Internal Response Time		**1**
Load Pipeline		**3**
Interrupt Vector	`j handler` `nop`	**2**

Interrupt Handler to write monitoring pin	`lui $26, 0x0003`
	`ori $26, $26, 0x0008`
	`li $27, 1`
	`sw $27, 0($26)`
		4

	Total	**10**

Table 7: PH-Predictable interrupt latency time breakdown

Figure 34 shows the simulation of an interrupt on the PH-Predictable core. It can be seen that the value '1' which is written to the GPIO port is present on the peripheral bus (pdata) 10 clock cycles after the interrupt occured.

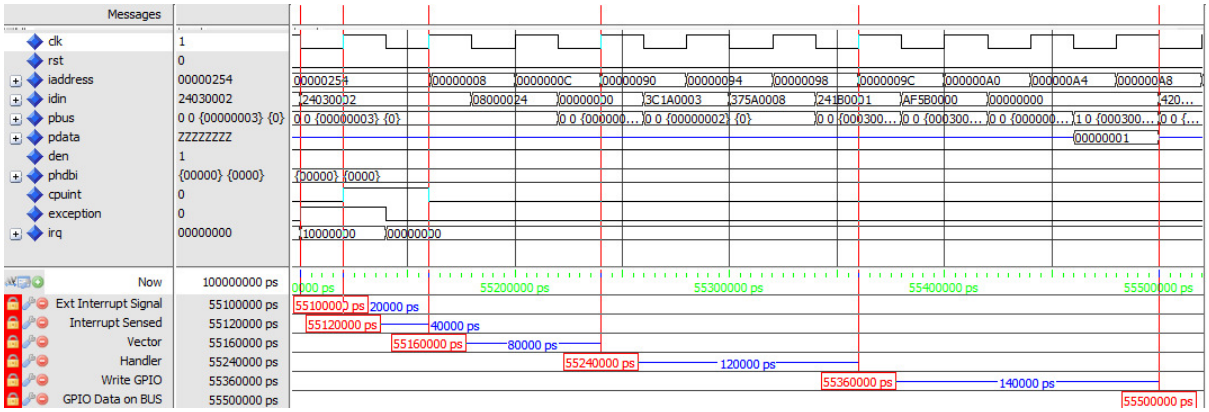


Figure 34: PH-Predictable core interrupt latency simulation[9]

Having shown that both the standard and the predictable core have zero jitter latency, the main difference should come in the next test where the hardware interrupt overhead on the predictable core should be static and not contain jitter as was seen on the standard PH core.

4.8.6 PH-Predictable Core - Test Case 2

The second test undertaken on the PH-Predictable core was similar to the test completed on the standard core, apart from the interrupt handler was modified since there was no need to re-enable interrupts and get the return address, instead a new instruction was added. The purpose of the 'retint' instruction was to signify to the processor that it should switch pipelines from the interrupt pipeline to the task pipeline. To add a bit more weight to the interrupt handler, a few 'NOP' instructions were added (Listing 6).

```
handler:
        nop
        nop

        # Return from handler
        retint
        nop
```

Listing 6: PH-Predictable core interrupt handler

[9] A larger version of this diagram is available in Appendix C, Figure 88.

When measuring the task execution time there was a base line which represented the task duration without being interrupted. The upper line represented the execution time when the task was interrupted. From the results it can be seen that when an interrupt occurred, the overhead was a constant 6 CPU cycles longer (Figure 35).

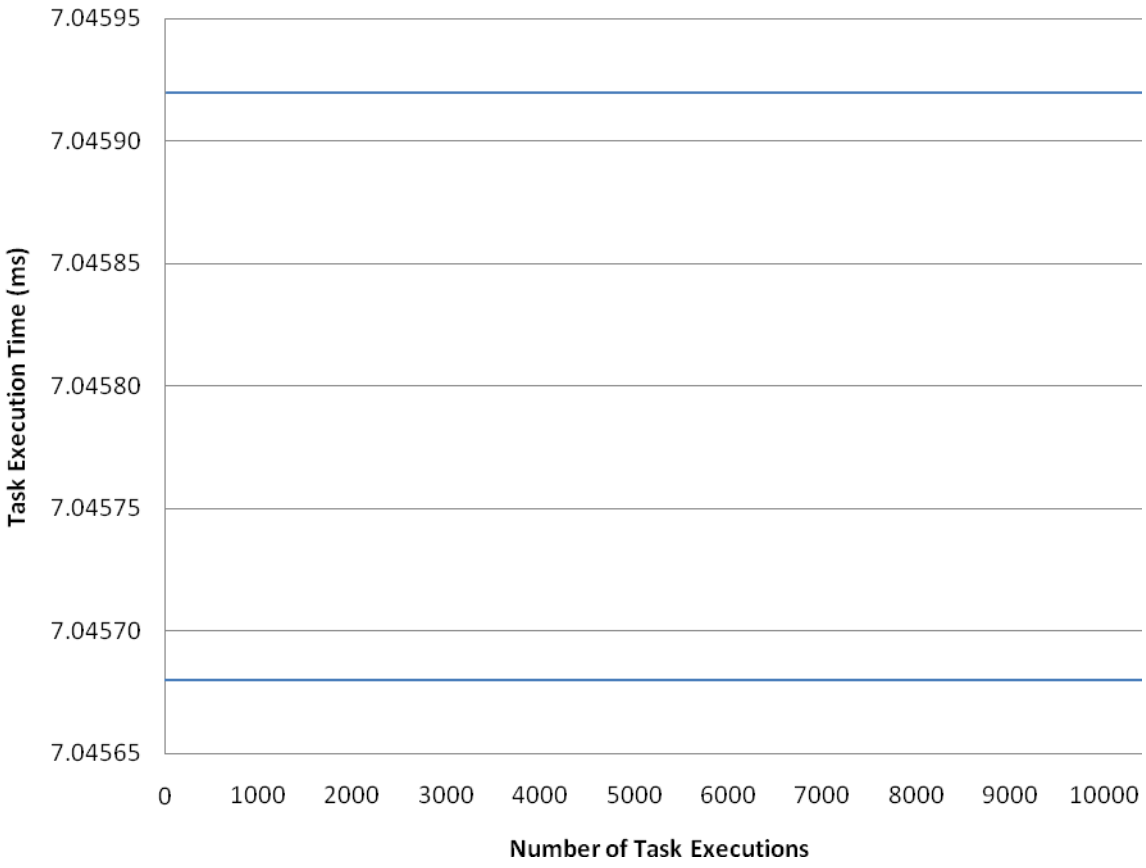


Figure 35: PH-Predictable task execution time

The exact measured values are shown in (Table 8).

	Execution Time (ms)	CPU Cycles
Max	7.04592	176148
Min	7.04568	176142
Difference	0.00024	6

Table 8: PH-Predictable core task execution maximum and minimum times

Justifying the 6 CPU clock cycle overhead time can be done by simply adding up the interrupt vector and interrupt handler instructions (Table 9).

Event	Instructions	CPU Cycle Count
Interrupt Vector	`j handler` `nop`	**2**
Dummy Interrupt Handler Code	`nop` `nop`	**2**
End of Handler Instruction	`retint` `nop`	**2**
	Total	**6**

Table 9: PH-Predictable core task execution overhead breakdown

Also of interest is that the interrupt overhead was static even when interrupting a multiply and divide instruction. This can be seen in Figure 36 where a multiply instruction has executed for 8 clock cycles when it is then interrupted by an interrupt handler whose code consumes a further 6 cycles. On return from the interrupt handler the multiply instruction completes the rest of the instruction by taking only a further 25 cycles. As a result the multiply instruction did not need to be re-executed and produced the correct result on completion.

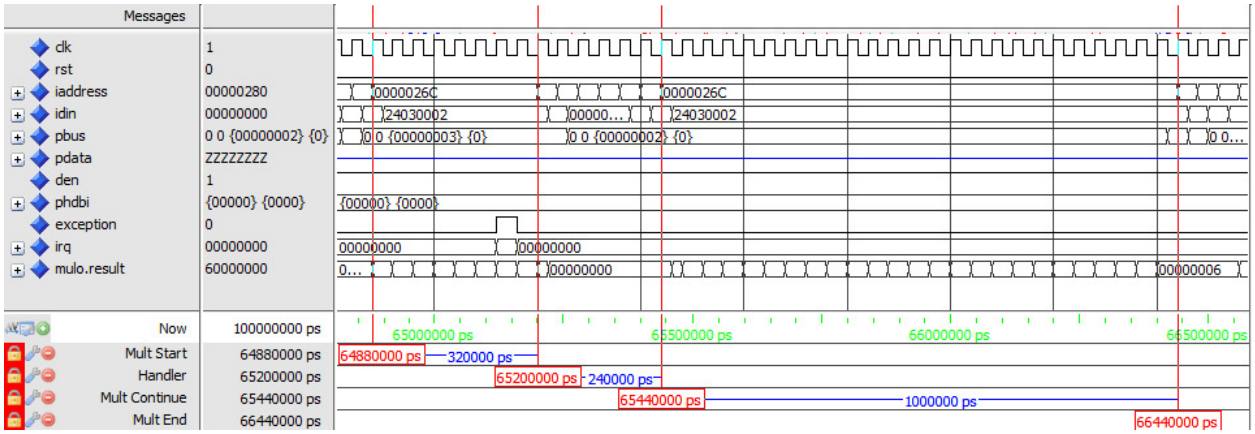


Figure 36: Multiply instruction paused as the interrupt handler is executed[10]

On the standard PH core, interrupting a multiply instruction resulted in a variable interrupt overhead which would vary by as much as 33 CPU cycles, or 34 CPU cycles if combined with a branch delay slot. The PH-Predictable core displayed predictable interrupt behaviour regardless of which instruction was interrupted.

[10] A larger version of this diagram is available in Appendix C, Figure 89.

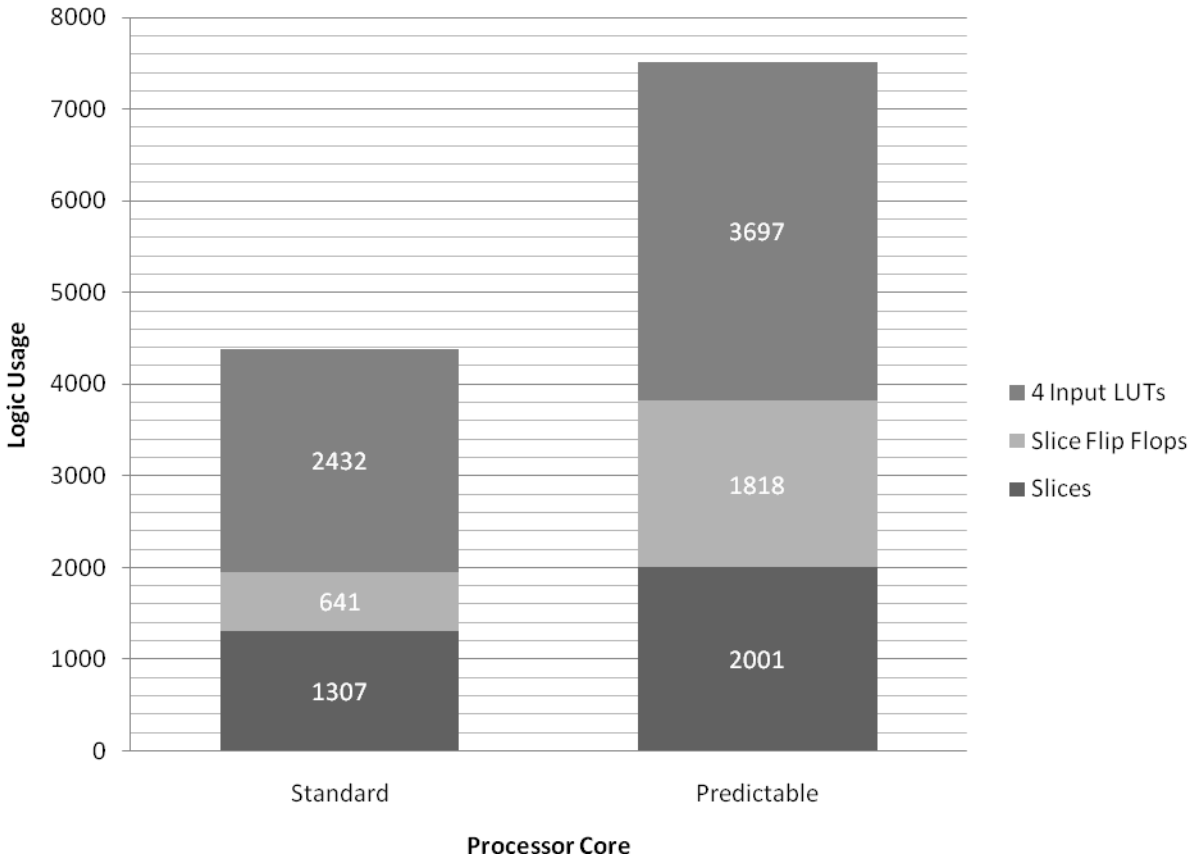


Figure 37: Comparisons of logic utilisation for the predictable multi-pipelined core

Making the processor predictable did come at a cost and resulted in the multi-pipelined design being 1.72 times larger than the standard processor core (Figure 37). The extra hardware utilisation is primarily from the need to duplicate the pipeline registers.

4.9 Discussion

This chapter has described a design for a predictable processor by making decisions to build on some existing architectural features and avoiding others which would lead to unpredictability. A choice was made to base the design around a MIPS pipelined processor. A solution to the problems of dealing with interrupt mechanisms, which are a requirement in most real-time operating systems, was provided. It was shown that the interrupt mechanisms in pipelined processors present a number of complications, especially when maintaining precise exceptions, dealing with branch delay slots and multi-cycle instructions. In Section 4.6, a number of design choices were considered however only one design using

a multi-pipelined solution solved all the problems in a practical way which would not impinge on performance.

To support time triggered architectures, the interrupt system on the processor was modified to conform to the 'one interrupt per microcontroller' rule. The new processor design was then compared against a similar processor design without the aforementioned features. The results demonstrated that even when comparing against a processor which uses an instruction abort mechanism, the predictable core produced no jitter in handling interrupts. The predictable core also imposed a static overhead over the code that it had interrupted. Furthermore, the interrupt overhead could be calculated by simply counting the instruction cycle times of the interrupt vector and interrupt handler code. This provides a more predictable and simple way of determining execution times when using an interrupt mechanism which would otherwise be non-obvious in systems using an instruction abort mechanism.

4.10 Conclusion

In Chapter 3, the causes for unpredictability in processor hardware were discussed and this led to the design and implementation of the predictable processor architecture described throughout this chapter. The unique multi-cycle instruction capable processor design was specifically aimed towards time-triggered architectures and included support for the one interrupt per microcontroller rule. The result was a highly predictable hardware platform with fixed and small interrupt overheads.

This chapter has demonstrated that a processor design with reasonable performance and very high predictability can be achieved. However, there remains a potential for the predictability to be lost by the implementation of the system code. This code can contain various control paths and loops. The variations in the control flow can then result in large differences in execution times and hence reduce temporal predictability. The next chapter looks at trying to solve this problem.

Chapter 5 TTC Hardware Scheduler

This chapter aims to address the issue of ensuring that the implementation of the TTC scheduler provides both predictable and easy to understand CPU overheads.

5.1 Introduction

When using a predictable processor there still remains a potential for a variance in execution times to be generated by the software implementation. The source of this unpredictability can be due to variations of program flow throughout the code. These can arise due to loops and conditional statements. For instance, on a predictable processor it may be possible to know the exact time when a task begins execution. However, it may not necessarily be possible to determine which paths and loops the code will take and thus when the task will actually finish. Consequently the overall CPU loads become unknown and the timing variation can then have knock-on effects on subsequent tasks.

The difficulty of predicting code execution times is not a new problem. One approach outline by (Puschner and Burns 2002) are techniques such as code balancing and single path programming. These may help to increase the predictability of code execution times. However, this method can result in a significant performance hit and thus it may not always be efficient or even practical to implement all code in this manner. An alternate solution can be to use bounded loops and obtain bounded WCETs rather than static execution times. Program control flow analysis and simulation can then be utilised to speculate when and how frequently different paths might be taken. However, this can result in a large variation between the best case and worst case execution times. A further problem exists if a piece of code is waiting on an external signal or event, and thus a relation between the code and external system properties must be factored in.

Often, embedded systems make use of a scheduler or RTOS in order to obtain synchronization and support multitasking. In general, these systems are built by combining three core components, a processor, RTOS and user tasks. The RTOS or scheduler may be built in-house or licensed from a third party and used in a variety of applications. Therefore, whilst it may not always be possible to control the way a programmer

implements their application code and tasks, it can be possible to control the implementation of the scheduler and RTOS on which the system is built.

Since a number of real-time applications do required some knowledge of system response times, many RTOS manufactures often aim to produce low task release jitter. This then at least gives a reference point on which to estimate task response times. Although on some systems this is generally applicable to higher priority tasks. It is then up to the application programmer to take a measured approach based on the particular system requirements on how they implement their individual tasks for predictability. Consequently, there is a limit to how predictable their task can be if not implemented on predictable hardware.

A similar problem is faced by RTOS designers where in order to achieve the minimal jitter and system overheads, the implementation of the software scheduler mechanism has to be undertaken carefully. This can prove to be complicated as the software implementation may not reflect the final output accurately, for instance due to compiler optimizations. As such, assembly code may often be used and as a side effect can result in the RTOS being less portable and have different timing properties across a wide number of COTS processors.

A key issue is that even if the final scheduler implementation can be made predictable, then a further problem exists if the application programmer does not have a clear of understanding of the scheduling mechanism and its complete system overheads. This may result in the tasks themselves being unpredictable. For instance, it may not be possible to know when and how much of the CPU time will be consumed by the scheduler and how much will be available for each task. Also the scheduler or RTOS loads may be related to the properties of the tasks in the system. As a result it is common practice in safety certifications to reassess the whole system even when a small part in a single task has been altered.

It is therefore desirable that the programmer and verification team can - without too much difficulty - understand and anticipate how the underlying scheduling technique, system architecture and system loads will be affected with each new program task. This in turn means that in addition to making the RTOS predictable it is generally desirable to keep the system implementation as simple as possible.

As described in Section 2.3, one of the simplest forms of multitasking scheduling architectures is a time-triggered co-operative approach (Pont 2001). However, when looking at scheduler implementation techniques, (Katcher, Arakawa et al. 1993) argues that there is a wide gap between scheduling theory and its implementation in operating system kernels running on specific hardware platforms. They also note that the implementation of a particular algorithm can introduce costs which must be taken into account when validating the timing correctness properties of a real-time system.

For instance, considering a simple time-triggered co-operative scheduling model as shown in Figure 4, time is split up into 1ms tick intervals in which one or more periodic tasks may execute as long as they return in a time less than the tick. However, the implementation becomes rather more complicated as the scheduling overhead and nature of the functionality for the particular microcontroller is introduced.

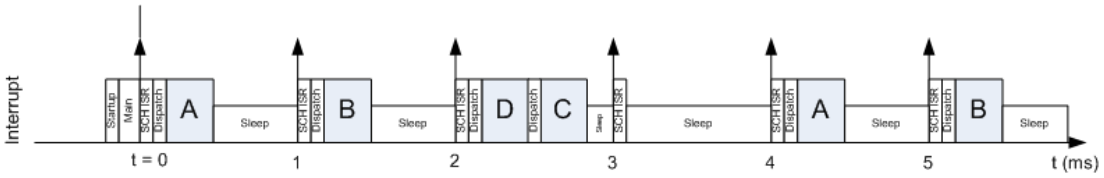


Figure 38: Detailed Time-Triggered Co-operative Schedule[11]

An example of this can be seen in Figure 38 which shows that the time left for the tasks to complete is no longer easy to determine and can largely depend on the conditions of the system. For instance, the number of tasks in the system may affect the time required for the scheduler to decide which task is to be executed next. Therefore taking into account all the overheads for what should in theory be a very simple scheduler, can actually become a rather complicated procedure. As a result, the aim is to attempt to achieve the execution as perceived and expected in the first diagram.

As previously mentioned, when implementing schedulers, some designers may use code streamlining by programming in assembly in an attempt to reduce the overhead impact and make calculating the system loads more deterministic. However, another approach can be to implement the scheduling system on hardware. This can have a number of benefits, including protecting the scheduler from being inadvertently modified and increasing system performance.

[11] A larger version of this diagram is available in Appendix C, Figure 90.

For instance, (Andrews, Peck et al. 2005) notes that: *"Specifically, developing an operating system with minimal jitter, deterministic behaviour, and fine scheduling granularity have by far been the most challenging goals for real-time operating systems designers. To achieve the best and tightest bounds on application program scheduling, migration of both the scheduler processing as well as the management of state information must be moved off of the CPU and into hardware components."*

Whilst the speed of silicon transistors appear to be no longer increasing as rapidly as it once use to, Moore's law on the increasing number of transistors is still being steadily applied. This then leads to the consideration of utilizing a custom hardware approach which is made more practical through the use of FPGA's. The intention is to make the system run in a simple and predictable way which can be easily understood by the application programmer. As a result, the scheduling overhead can be removed making it easier to determine CPU loads.

In scheduler implementations the following design features are desirable:

- Zero task jitter
- Constant and predictable system overhead
- Predictable schedule
- Efficient and simple design

5.2 Related Work

Before continuing, this section gives a brief overview of the work in literature on the development of hardware scheduling units.

A popular goal in literature is to increase system performance for pre-emptive kernels by implementing various system features into hardware units (Furunäs 2000; Lee, Ingström et al. 2003). Examples include, a System-on-Chip Dynamic Memory Management Unit (SoCDMMU) (Shalan and Mooney 2000), Multiprocessor synchronization support (Akgul and Mooney 2001), System-on-a-Chip Deadlock Detection Unit (SoCDDU) (Shiu, Tan et al. 2001), System-on-a-Chip Lock Cache (SoCLC) (Akgul and Mooney 2002), hardware support for priority inheritance (Akgul, Mooney et al. 2003), resource locking and message passing (Mooney and Blough 2002; Sun, Blough et al. 2002).

Many hardware scheduling units are based on pre-emptive scheduling algorithms as the overhead of context switching can be costly (Garcia, Vila et al. 1999; Saez, Vila et al. 1999; Kohout, Ganesh et al. 2003).

Some system include the support for a range of scheduling architectures such as round-robin, priority based, rate monotonic and earliest deadline first (Andrews, Niehaus et al. 2004; Andrews, Niehaus et al. 2004). Whilst others systems can modify the scheduling architecture in real-time (Kuacharoen, Shalan et al. 2003; W. Peck, J. Agron et al. 2004; Andrews, Peck et al. 2005).

Some features are scalable in the design phase (Lindh, Klevin et al. 1999) and lead to the developments in hardware and software workflows (Young and Wilde ; Niehaus and Andrews 2003; Issacson and Wilde 2004; Klingler and Wilde).

For many systems, the ability to perform context switching in hardware has been shown to have higher performance and improved determinism (Adomat, Furunäs et al. 1996; Stärner 1998). However the interest of the work in this thesis is on increasing the predictability of embedded systems.

A set of projects - based on pre-emptive schedulers - starting with FASTCHART showed that it was possible to get deterministic execution time through a CPU and real-time kernel. (Lindh and Stanischewski 1991; Lindh and Stanischewski 1991). However, in order to obtain this predictability the system had excluded the use of pipelines, caches, DMA and interrupts.

In order to improve performance, FASTHARD improved on FASTCHART by adding the support for interrupts. However the system was built as a purely hardware real-time kernel which would be interfaced with standard COTS processors (Lindh 1992; Lindh 1993). They showed that performance and determinism was better than software, however the use of a COTS processor meant that the predictability of the tasks was reduced.

The Real Time Unit 94 (RTU94) improved on FASTHARD by its capability to control multiprocessor systems (Furunäs, Stärner et al. 1995; Lindh, Furunäs et al. 1995; Adomat, Furunäs et al. 1996; Lindh, Stärner et al. 1998). The unit also includes semaphores, event flags and watchdogs. Tasks could either be fixed to a local processor or be dynamically scheduled on any processor. The RTU implements a priority based pre-emptive scheduler

for each CPU, with a local queue for each processor and a global queue for global tasks which can dynamically balance the load in the system. The emphasis of this work moved from predictability towards performance which was greatly improved.

The Hard Real-Time Compact Kernel (HARETICK) project presented a solution for executing co-operative tasks in a high priority mode (Micea, Cretu et al. 2005). The hardware scheduling unit was interfaced to a COTS processor where they stated that the WCET must be overestimated slightly to encounter for unpredictability. This resulted in a decrease in operating efficiency. Therefore, their solution was to include soft real-time tasks which would consume any leftover slack time. These tasks would operate in a low priority mode and be pre-empted when a hard real-time task is released. A problem with this solution is that the pre-emption on an unpredictable processor would incur variable overhead. Also the verification process will be more complicated when mixing hard and soft real-time tasks on the same system.

Through the literature it has been shown that most work has been focused on increasing the performance and determinism of hardware based pre-emptive systems. The HARETICK solution was found to be the closest to the work in this thesis. However, the focus of the work here is towards the support of the TTC scheduler architecture to be built around a predictable processor in order to meet the objectives of reduced system overhead complexity and increased temporal predictability which can be predicted down the instruction cycle level. It is thought that this architecture is novel and has not been undertaken before[12].

5.3 TTC Software Scheduler Implementation

Before describing a hardware scheduling solution, this section will described the operation of the time-triggered co-operative software mechanism as presented by (Pont 2001).

The time-triggered co-operative (TTC) scheduling system operates by splitting up the available processor time into "Tick" intervals, often with a typical duration of 1ms. Most tasks in the system will illustrate periodically: for example, every 3 ms (see Figure 39). In a

[12] A Patent application for the TTC hardware scheduler has been filed. Please see the list of publications.

TTC design, each task must finish and pass control back to the scheduler before the next task is run.

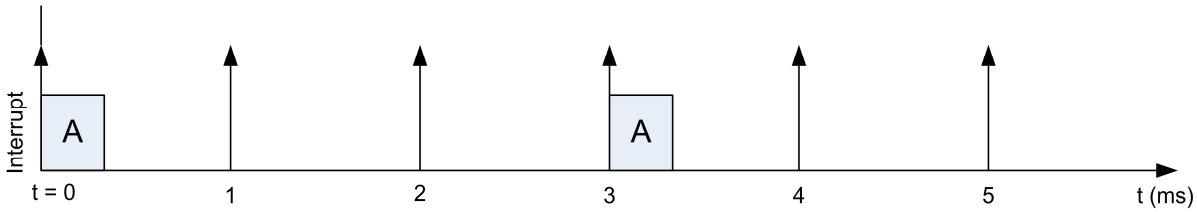


Figure 39: Periodic co-operative scheduling

To obtain this behaviour, a periodic timer linked to an interrupt service routine is required, plus (usually) a separate "dispatch" function e.g. see (Pont 2001). To achieve this, a conventional timer often has a set of count registers configured to generate the periodic interrupts. In the example shown in Figure 40, a prescaler register is used to bring down the high onboard clock frequency to a reasonable value for the timer registers. The prescaler count register is decremented with every pulse from the onboard clock source. On an underflow, a tick signal is set high upon which the prescaler reload value is then placed back into the count register and the process continues. The timer registers operate in a similar manner, but are instead driven by the tick clock from the prescaler. On a timer count register underflow an interrupt pin is set high where a processor then loads the default interrupt vector.

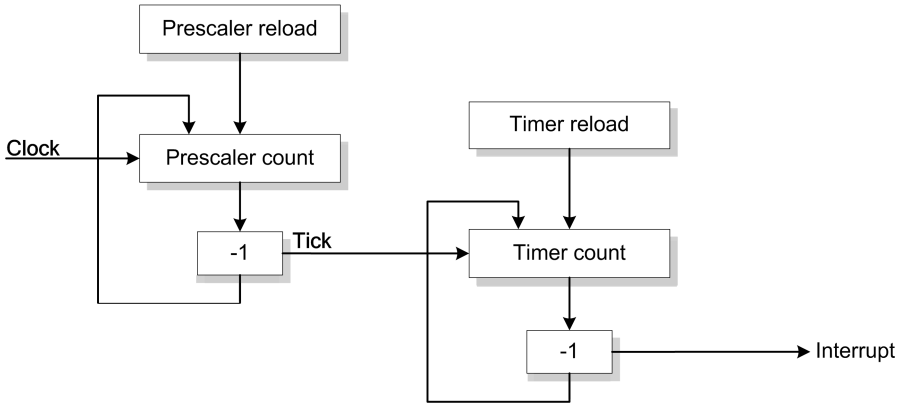


Figure 40: Generic timer operation

5.4 TTC Hardware Scheduler

Like the software scheduler the TTC hardware scheduler contains two major components, Update and Dispatch connected together by a task FIFO. Update feeds the FIFO with new task ID's when they are scheduled to execute. The dispatch unit executes these tasks until the FIFO becomes empty. Figure 41 shows an overview of the internal structure of the hardware scheduler.

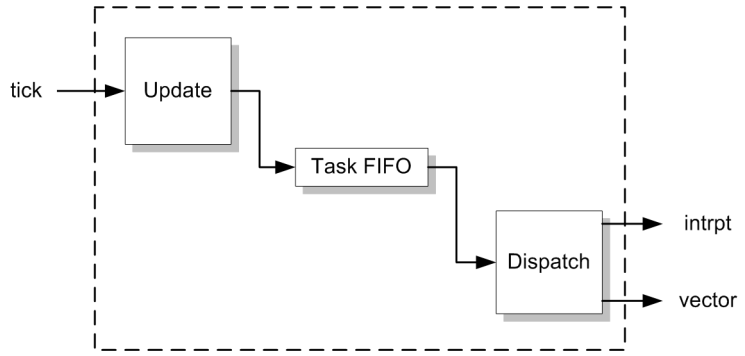


Figure 41: Hardware scheduler functional overview

Driven by a timer, the Update component waits until a new tick has been signalled where it then decrements each tasks delay value as long the respective task enable bit is set. When the tasks delay value underflows, the delay value is reloaded with the period and the task id is inserted into the FIFO queue. In one view, the update unit can be considered as a timer with multiple count and reload registers which contains one per task. When a task is due for execution it is added to the task FIFO queue.

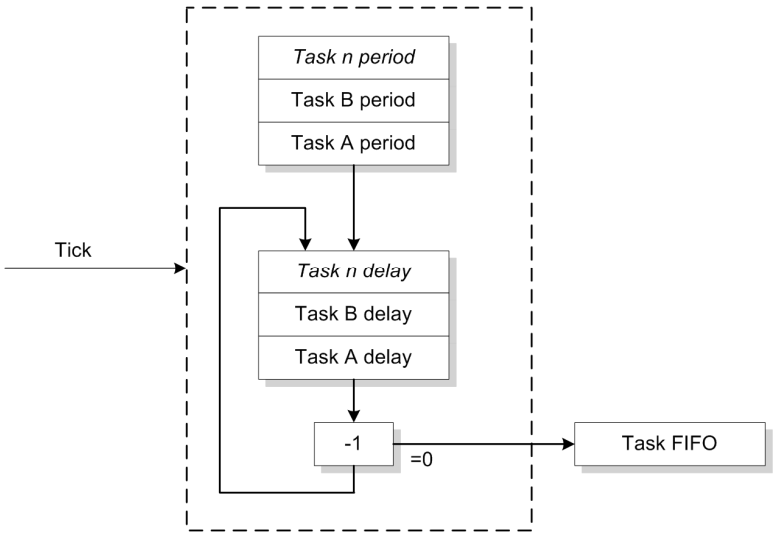


Figure 42: Hardware scheduler Update functional overview

The dispatch component waits until the FIFO queue contains tasks to be executed. If the queue is not empty then dispatch executes the first task. The dispatch unit then waits until current task has completed where it then continues by executing any remaining tasks.

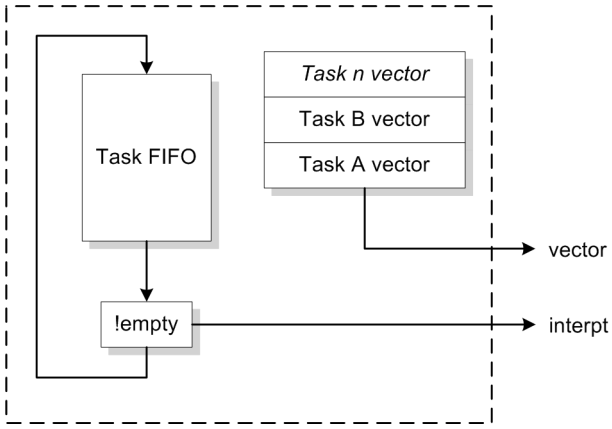


Figure 43: Hardware scheduler Dispatch functional overview

5.5 The Processor Interface

The first problem with the hardware scheduler mechanism is to find a way to instruct the processor to execute a task from a specific location in memory. The solution bears resemblance to an interrupt mechanism, since in a similar manner an interrupt directs the processor to execute a portion of code from a vector address. The main difference here is that the vector address for the task to be executed comes externally from the hardware scheduler.

Using this technique there must also be a means by which the hardware scheduler is notified of the tasks completion so that it may instruct the processor to execute the next task. This is achieved by implementing an 'endtask' instruction which is to be placed at the end of each task.

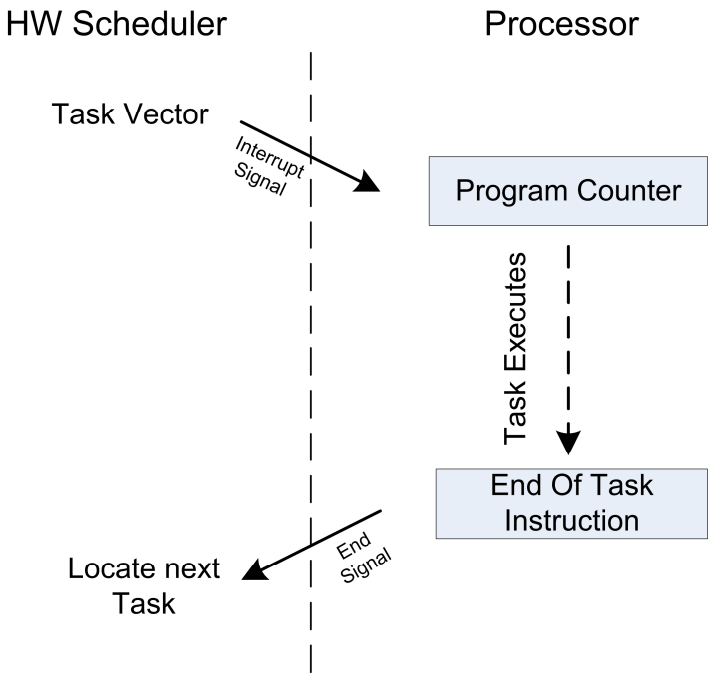


Figure 44: TTC Hardware scheduler interface overview

After the power up initialisation routine the processor is only required to execute tasks as instructed by the hardware scheduler (Figure 44). As a result, there is no need for the tasks to save and restore registers or even return back to any code. Instead, when the end task instruction is executed, the processor sends an end task signal to the hardware scheduler core - which assuming no further tasks are to be executed - places the processor into a suitable low power sleep mode. The processor then remains in the low power sleep mode until eventually signalled to execute another task.

One benefit to this system is that since the core instructs tasks to execute directly, there is no real software interrupt handling mechanism needed apart from exception handlers to deal with erroneous conditions. As a result, the predictable multiple pipeline method as presented in Chapter 4 for the software scheduler is no longer required. Consequently this simplifies the processor design.

5.6 Overhead Reduction

In the simplest implementation, when the 'endtask' instruction reaches the instruction decode pipeline stage the processor sends the end task signal to the hardware scheduler core (Figure 45).

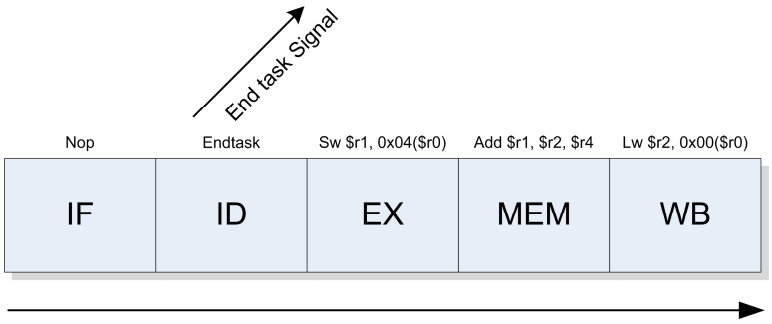


Figure 45: Initial solution

However, there are two problems with this method. The first problem is that the instruction immediately following the 'endtask' instruction is loaded into the pipeline and therefore there exists at least one CPU clock cycle overhead before the next task might be loaded into the pipeline (Figure 46). Secondly, the instructions in execution and instruction fetch stages could potentially raise an exception which would not get serviced until the MEM stage. This would result in an exception being raised after the end task signal has already indicated that the task has finished.

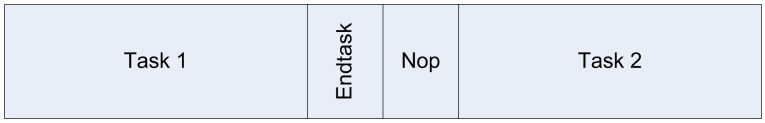


Figure 46: Example solution 1 overhead

A quick fix to the problem might be to move the end task signalling to the execution stage at which point the previous instructions can no longer generate exceptions (Figure 47).

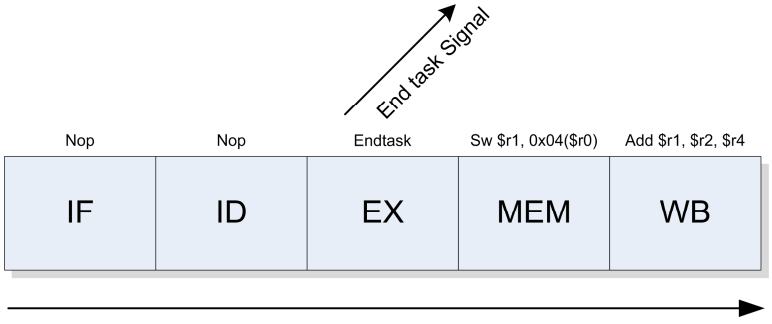


Figure 47: Alternate solution

However, this puts the constraint on the system that an 'endtask' instruction must be padded by two no-operation instructions since they do not generate exceptions in the pipeline. Consequently this then increases the overhead to 2 CPU clock cycles between tasks (Figure 48).

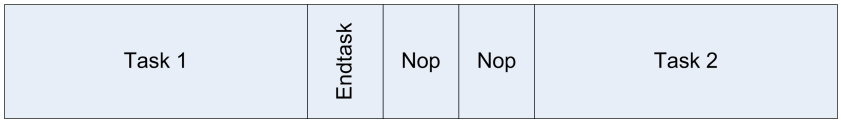


Figure 48: Example solution 2 overhead

Since the NOP instructions do not provide any useful work, it would be advantageous to attempt to get tasks to run back to back without any additional overhead. Starting each task with an interrupt type signal would also flush the first 3 pipeline stages and consequently waste CPU time. Instead, in order to meet the demands of removing the between task overheads, a solution to run tasks back to back with correctly ordered exceptions and end task signals is required.

A modification can be made to the hardware scheduler to output the task vector for next task whilst the current task is still executing. A small amount of logic in the instruction fetch stage could detect an end task instruction and redirect the program counter from fetching the next instruction to fetching instructions for the next task (Figure 49).

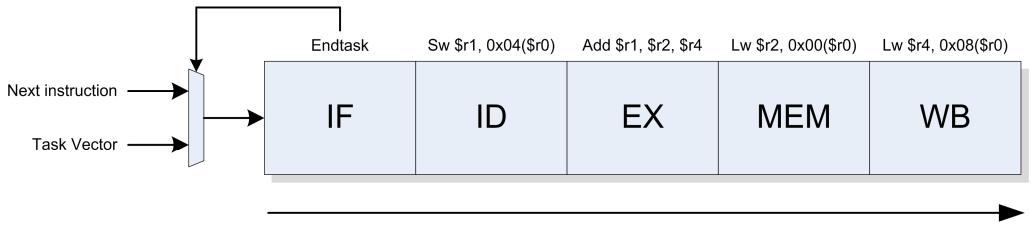


Figure 49: Redirecting instruction flow when 'endtask' instruction is detected

When the 'endtask' instruction propagates down to the execution stage where the previous task can no longer generate an exception, the hardware scheduler core can then be notified of the end of task where it would either output the task vector for any following tasks or send the processor to a low power sleep mode (Figure 50).

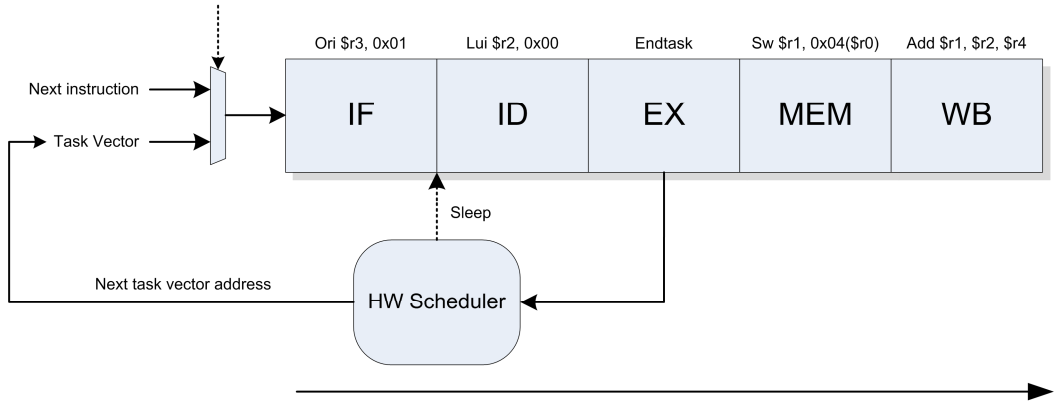


Figure 50: Signalling the end of task to the hardware scheduler

In the scenario where there are no more tasks to be executed and the processor is sent to sleep, the instructions in the pipeline stages IF and ID would be the first two instructions of task 1. Since task 1 may not be due to run next and that the sleep mode is deactivated by an interrupt signal from the hardware scheduler, then the first two instructions of task 1 will be flushed out of the pipeline.

The sequence of events starting at the beginning of a tick interval is as follows. When the tick begins and assuming that there are tasks to be executed, the hardware scheduler sends an interrupt signal along with the task vector of the task to be executed. This interrupt signal flushes the first three stages of the pipeline to remove any potentially unwanted instructions. The first instruction of the task to be executed takes 3 CPU clock cycles before it reaches the end of the execution stage and begins to change the system state (Figure 51). If there are tasks that follow the currently executing task then at the point when the 'endtask' instruction is loaded, the first instructions of the next task are preloaded into the pipeline. When the 'endtask' instruction reaches the execution stage, the hardware scheduler allows the new task to execute. If there are no further tasks to be executed and the now currently executing task reaches its 'endtask' instruction, the hardware scheduler puts the processor to sleep.

Figure 51: Overheads when using the hardware scheduler

Essentially, each tick starts with a three CPU clock cycle delay and each task must be terminated by an 'endtask' instruction. However, the hardware scheduler unit does require

some time to compute the task schedule and therefore requires some clock cycles. Although, this doesn't affect the CPU load as the schedule is calculated in parallel to the processor. What happens is there is an offset between the timer signal going into the hardware scheduler and the first task being executed on the processor. This isn't visible to the processor, just that the timing on the processor is offset slightly from the timer. However, this offset is kept constant so that the first task in a tick always executes the same time at the beginning of each tick interval.

5.7 Results

To demonstrate that the hardware scheduler and processor are predictable as intended, two test cases were devised. The first test case shows the offset time by measuring the duration from the timer tick being raised to the point where hardware scheduler actually executes the first task. The second test case shows the back to back execution of tasks by measuring the time between one task finishing and the next task beginning.

5.7.1 Test Case 1

To measure the offset time, a second FPGA was loaded with a measuring core which will be used to generate periodic tick signals. These tick signals are fed into the hardware scheduler and processor core (Figure 52). When directed by the hardware scheduler, the first few instructions of the first task will set a GPIO pin high. The measuring core will then measure the time between the tick signal going high and this GPIO pin going high. This interval is recorded by a hardware counter which is clocked at 100MHz.

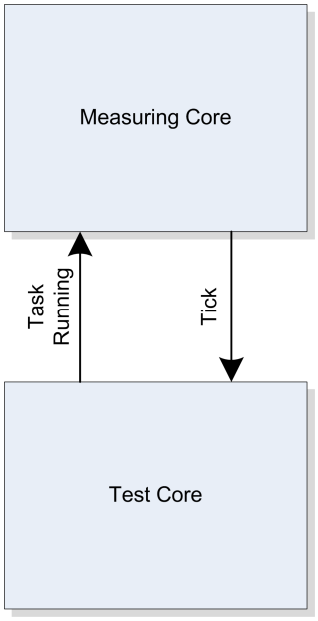


Figure 52: Setup for test case 1

For this test the tick period is set to 10ms to allow time for the counter values to be sent to a host computer for storage. A simple flashing LED task will be used for this test and is called by an assembly wrapper (Listing 7). The first few instructions of the wrapper set the GPIO pin high before calling the task. Once the task returns, the GPIO pin is set low again and the 'endtask' instruction is executed.

```
first_task:
      # Set GPIO pin1 high
      lui $26, 0x3
      ori $26, $26, 0x8
      li  $27, 1
      sw  $27, 0($26)
      # Call the LED Task
      jal LED_Update
      nop
      # Set GPIO pin1 low
      lui $26, 0x3
      ori $26, $26, 0x8
      sw  $0, 0($26)
      # End of task
      endtask
```

Listing 7: Assembly wrapper for the first task in test case 1

When measuring the test case it can be seen through the scatter diagram in Figure 53 that the offset time is constant throughout each run.

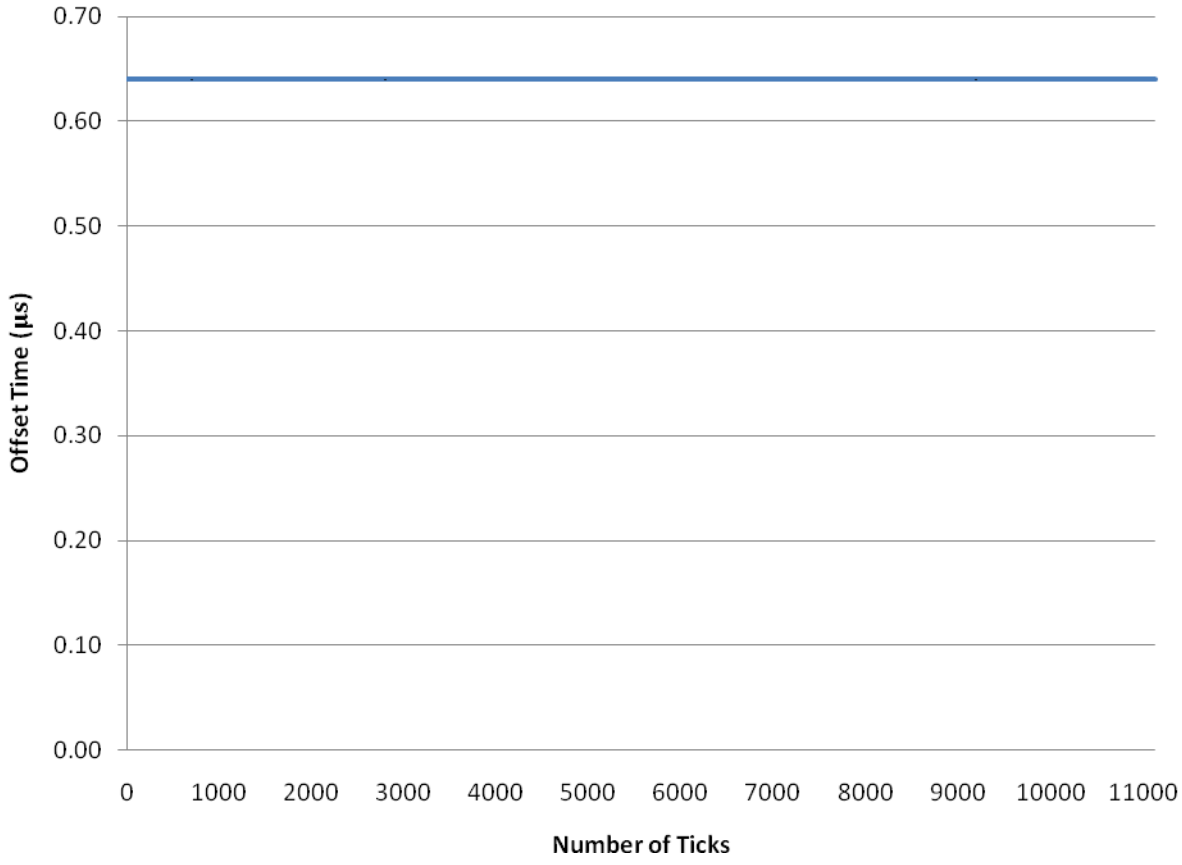


Figure 53: Measured offset time between tick and first task reacting

From Table 10 it can be seen that there is a 16 clock cycle offset from when the tick signal is fed into the hardware scheduler and the first task executing the instructions required to set the GPIO pin high.

	Offset Time (µs)	CPU Cycles
Measured	0.640	16

Table 10: Detailed measured offset time between tick and first task reacting

Table 11 shows a breakdown of where these 16 clock cycles come from. For this test the hardware scheduler component was synthesised for up to 8 tasks. By design, the dispatch part of the hardware scheduler will not execute the first task until 8 cycles has passed (Figure 54). This delay is based on the number of tasks the unit can handle and ensures the update part of the hardware scheduler has enough time to update each of the task count values. It also ensures that the first task, whether it is Task 1 or Task 8, will always start after a fix period after the tick signal has been received.

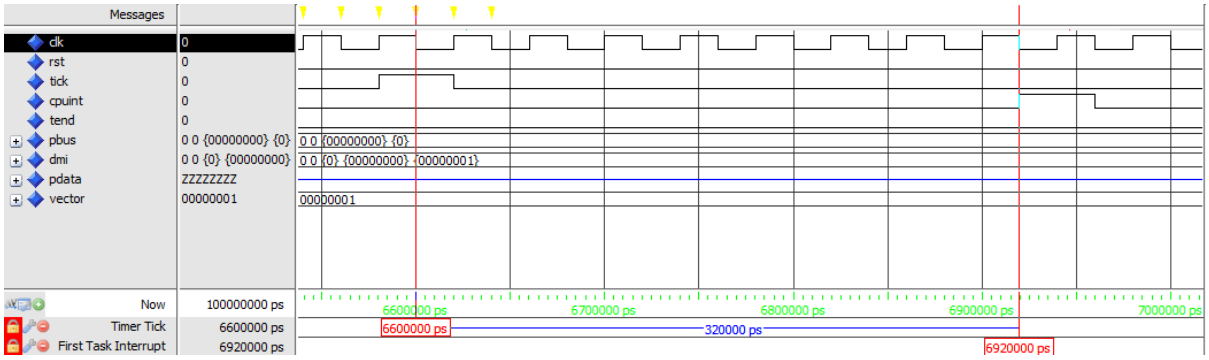


Figure 54: Hardware scheduler offset delay[13]

Following the 8 clock cycle delay, there is a single cycle consumed by the processor sensing and responding to the execute task signal. The processor then loads the pipeline with the tasks instructions which takes 3 clock cycles until the first instruction reaches the end of the execution stage. Four instructions must then be executed to set the GPIO pin high so that the measurement can take place (Table 11).

Event	Instructions	CPU Cycle Count
Hardware Scheduler offset		**8**
Internal Response Time		**1**
Load Pipeline		**3**

Set GPIO Pin High	`lui $26, 0x3`
	`ori $26, $26, 0x8`
	`li $27, 1`
	`sw $27, 0($26)`
		4

	Total	**16**

Table 11: Breakdown of the measured offset results

From these measurements it can be deduced that there is a hardware offset time of 9 clock cycles plus a 3 clock cycle overhead before the first task is executing. The 9 clock cycle offset time means that the processor is offset from the tick signal by 9 cycles, however, this may not appear as CPU overhead as the task in the previous tick can run right up until the end of these 9 clock cycles.

5.7.2 Test Case 2

This test aims to demonstrate the overhead between one task completing and the next task starting. Two tasks are therefore required which are called by means of an assembly

[13] A larger version of this diagram is available in Appendix C, Figure 91.

wrapper. The first task is a simple flashing LED task which before completion will set a GPIO pin high and execute the end task instruction (Listing 8).

```
led_task:
        # Call the LED Task
        jal LED_Update
        nop
        # Set GPIO pin1 high
        lui $26, 0x3
        ori $26, $26, 0x8
        li  $27, 1
        sw  $27, 0($26)
        # End of task
        endtask
```

Listing 8: Assembly wrapper for the first task in test case 2

The second task is a seven segment task which sets a different GPIO pin high before calling the main task function and executing the end task instruction (Listing 9).

```
seg_task:
        # Set GPIO pin2 high
        lui $26, 0x3
        ori $26, $26, 0x8
        li  $27, 2
        sw  $27, 0($26)
        # Call the SEG Task
        jal SEG_Update
        nop
        # End of task
        endtask
```

Listing 9: Assembly wrapper for the second task in test case 2

The test will use a similar setup as described in the previous chapter with the main difference being that the counter will measure the time between the first and second GPIO pin going high (Figure 55).

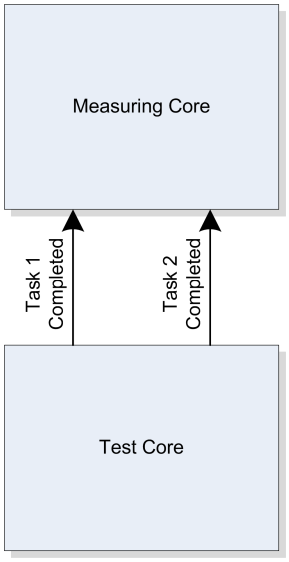


Figure 55: Setup for test case 2

In Figure 56 it can be seen that the time between the tasks is constant and does not vary.

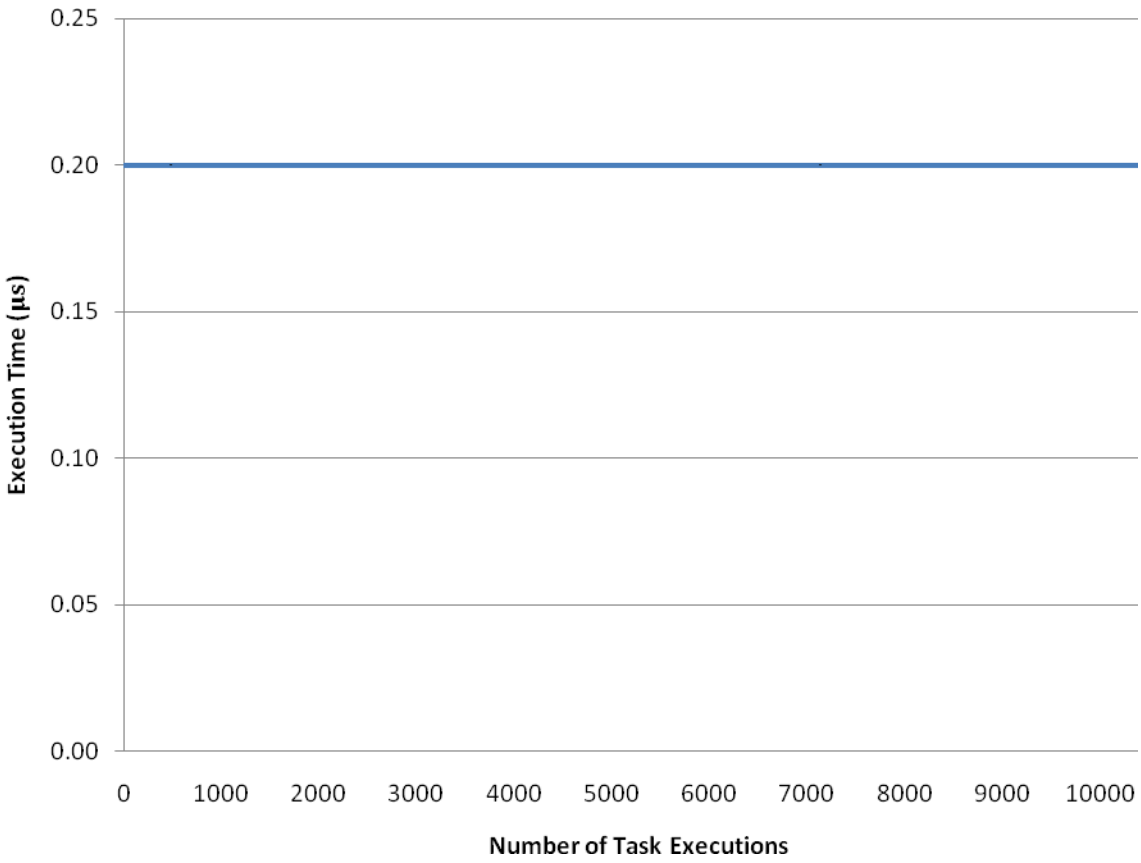


Figure 56: Measured time between first and second task

From Table 12 it can be seen that there is a 5 clock cycle duration between the first GPIO pin going high and the second GPIO pin going high.

	Execution Time (µs)	CPU Cycles
Measured	0.200	5

Table 12: Detailed measured time between first and second task

The breakdown of the 5 clock cycle duration is broken down in Table 13 where 4 of the measured clock cycles come from setting the GPIO pin high and only 1 clock cycle from executing the 'endtask' instruction.

Event	Instructions	CPU Cycle Count
Task 1: End of Task	`endtask`	**1**

Task 2: Set GPIO Pin 2 High	`lui $26, 0x3`
	`ori $26, $26, 0x8`
	`li $27, 2`
	`sw $27, 0($26)`
		4

	Total	**5**

Table 13: Breakdown of the measured between task time

Figure 57 shows the whole process from when the tick signal enters the hardware scheduler component to when the GPIO pin is set high demonstrating the 5 clock cycles as observed in the measurements.

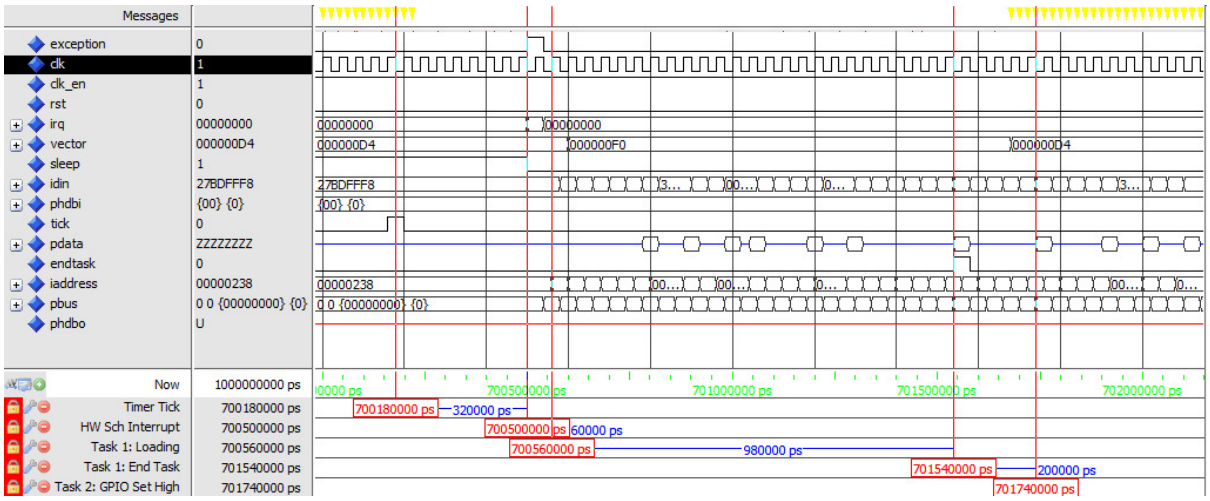


Figure 57: Simulated between task overhead[14]

[14] A larger version of this diagram is available in Appendix C, Figure 92.

In summary, the execution of tasks showed a 9 cycle offset from when the tick signal is fed into the hardware scheduler and an execute task signal is received by the processor. There was then a 3 cycle overhead to load the first task into the pipeline. It was then shown that there is a single cycle overhead between tasks because of the need to execute an 'endtask' instruction. In addition, it was observed that there was no jitter or variance to the measurements.

5.8 Analysis between hardware and software

When comparing the hardware scheduler core against the standard PH and predictable cores as introduced in Section 4.8.1, it can be seen that the overheads to the first executing task are significantly reduced (Figure 58). The reason why the predictable core has less overhead than the standard PH core is because the interrupt handler does not need to save and restore registers. It is worth point out that the loads measured for the software schedulers are based on the time to execute Task 1 at the beginning of the tick. If time was measured between the interrupt and the Task 2 as the first task in the tick, then the overhead would be slightly larger due to the software mechanism needing to iterate once more around a loop. This problem is not faced in the hardware scheduler which will always have fixed load regardless of which task is executed first in the tick.

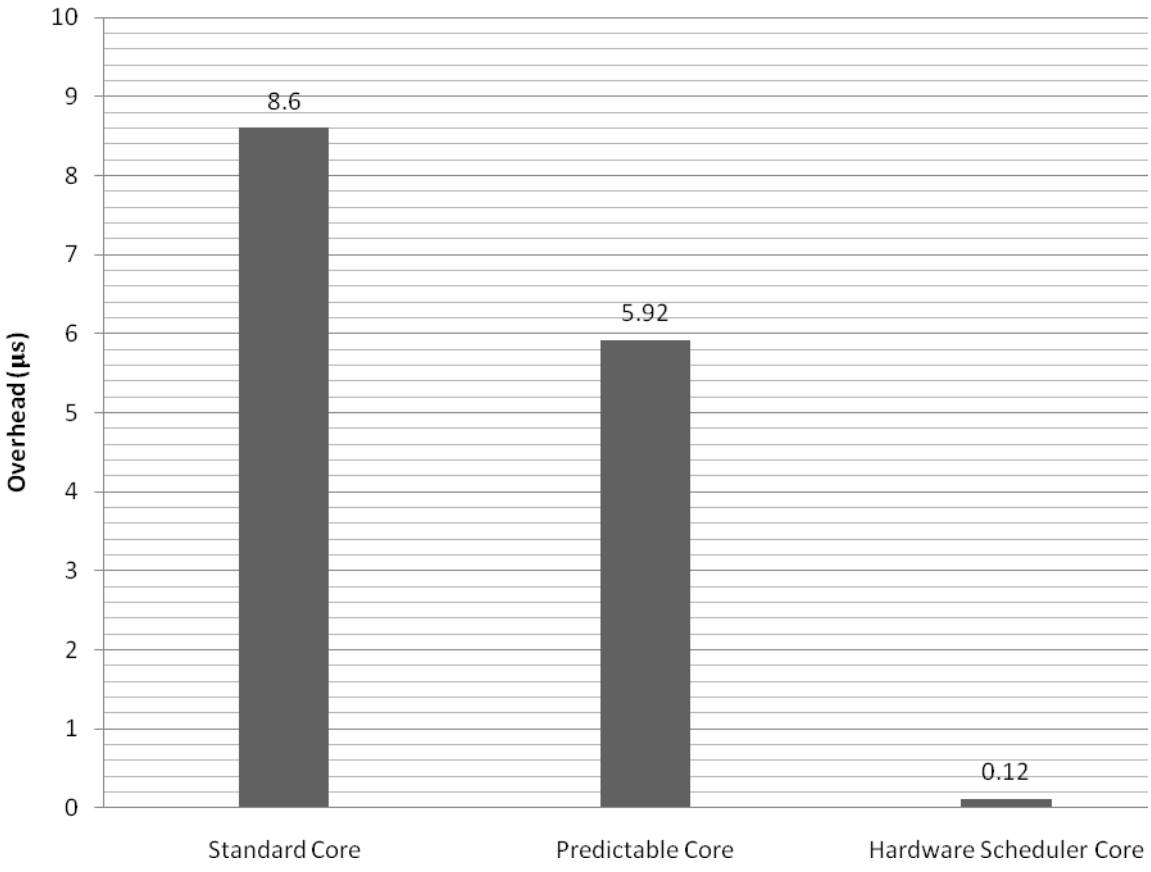


Figure 58: Software scheduler loads on standard and predictable cores

The overheads between tasks is also variable in the software mechanisms as can be seen in Figure 59. However, the hardware scheduler has a fixed one cycle overhead between tasks which is due to the need to execute an end task instruction.

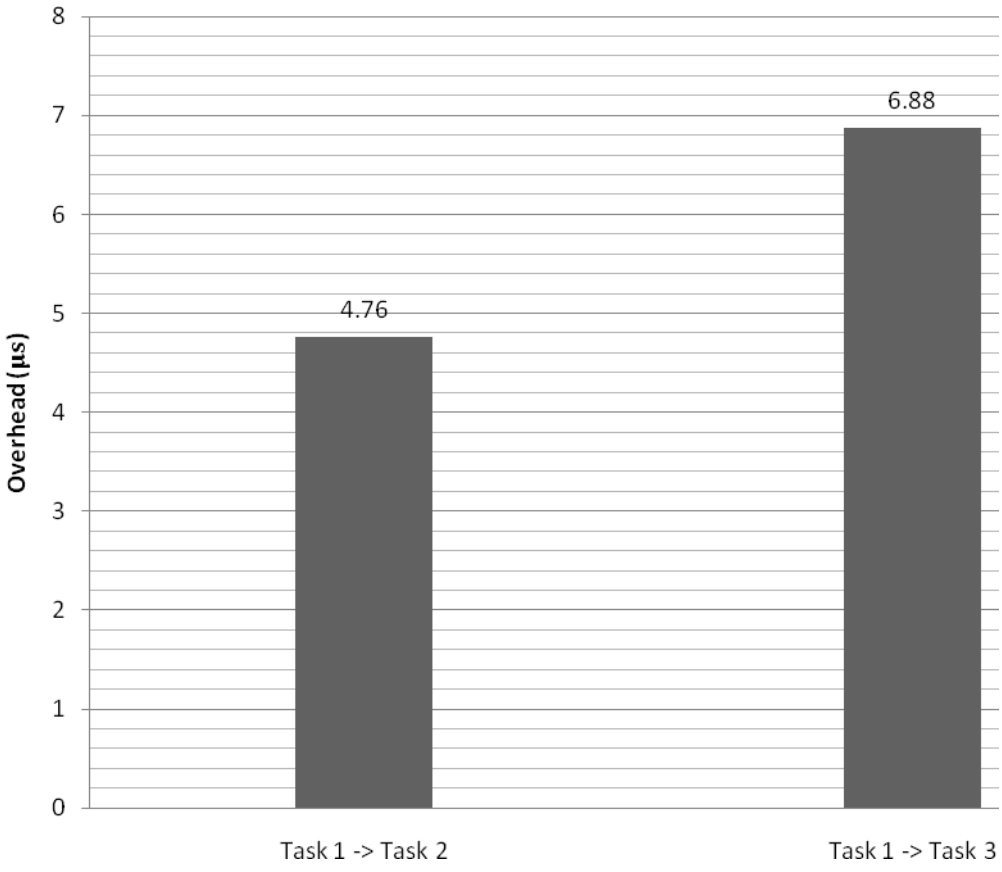


Figure 59: Variable software scheduler overhead between tasks

When comparing the code and data sizes between the software and hardware schedulers, it can be seen that the hardware mechanism consumes much less memory (Figure 60). The memory that is consumed for the hardware scheduler are for helper functions to load the hardware scheduler registers when the system is first started.

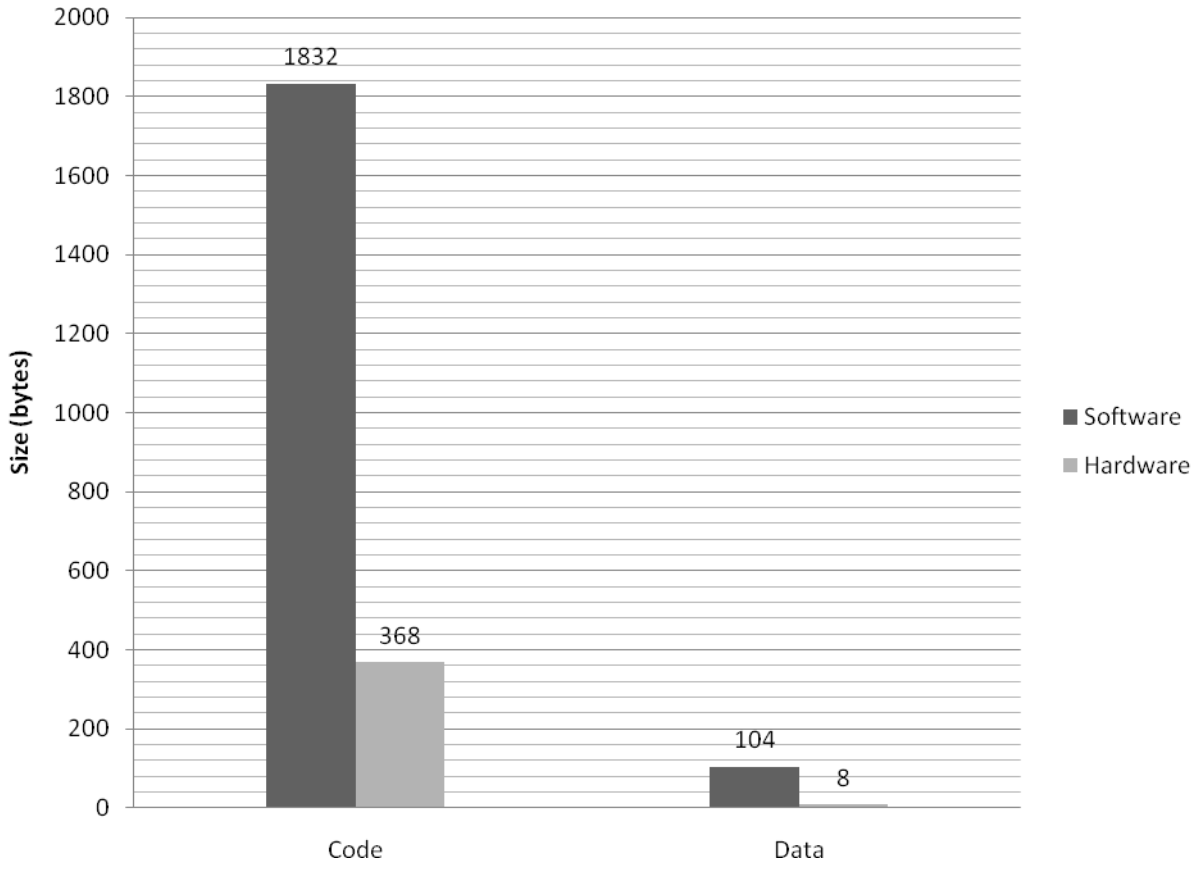


Figure 60: Hardware and software scheduler code and data sizes

Some of the other benefits from using a hardware scheduler are not only is the predictability better, but also the hardware provides protection from the user code and software bugs whilst enhancing system performance.

Due to the complexity of understanding the software scheduler loads, this chapter looked at moving the scheduling mechanism from software to hardware. A large benefit of this was that the processor core no longer required the need for an interrupt handler as the tasks executions are directed from hardware. As a result, the multi-pipelined core presented in Section 4.6.2.5 for the support of predictable interrupts was no longer necessary. This resulted in the processor core including the hardware TTC scheduler and support for up to 8 tasks being 13.5% smaller than the multi-pipelined core (see Figure 61). However, the new processor core was still 51.2% larger than the standard unpredictable processor core.

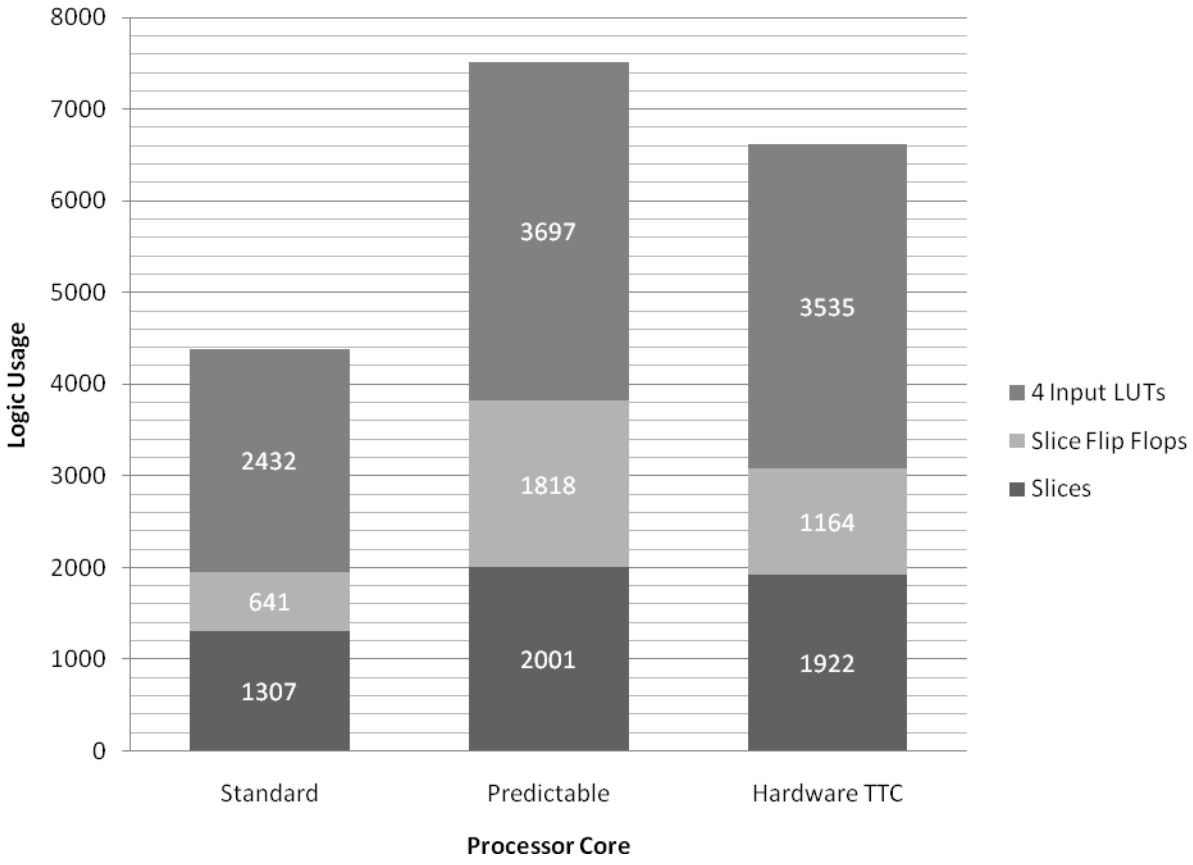


Figure 61: Comparisons of logic utilisation for the HW TTC core

Even though the logic consumption was larger, the design ensured that the scheduler loads were now much smaller and fixed. These loads were fixed with a 3 clock cycle overhead for the first task in a tick interval and had a between task overhead of 1 CPU cycle due to the inclusion of an 'endtask' instruction.

5.9 Discussion

This chapter has dealt with the issue of unpredictable scheduler loads on the processor by adopting a hardware solution. By comparison, the use of a scheduler implemented in software resulted in variable execution times before the first task is executed in a tick interval. Also the scheduler exhibited variable overheads when switching between tasks due to variations in the control paths. This was noticeable when comparing the load when switching between the first and second task to the load between the first and third task which had an increase of 44.5%.

The very nature of a scheduler being implemented in software can also give rise to timing which is a more complex to understand. For instance, compiler optimizations could alter the order and the amount of the generated code and thus lead to different scheduler loads.

The hardware scheduler mechanism described in this chapter produced loads that were both fixed in duration and easy to understand. These loads would not be affected by changes to the software. Furthermore the core size was smaller than the predictable core and the code and data sizes were reduced whilst the performance was increased.

The simplicity of the overheads in the hardware scheduler makes it easier to prevent mistakes being made in the calculations. By moving the scheduling mechanism from software to hardware the system offers better protection from malicious and accidental alterations to the system code. This means that changes to the functionality of the scheduler are restricted unless the hardware can be regenerated. However, verifying systems in hardware are considered to be easier than verifying software systems, thus hardware designs can more suitable for safety critical type systems.

5.10 Conclusion

This chapter has presented a novel design for a hardware TTC scheduler which has minimised the scheduling overhead and increased the predictability of embedded systems. This design is unique because it provides very small overheads for a pipelined processor based on the TTC scheduling architecture, with just 3 CPU cycles before the execution of the first task in a tick interval and the inclusion of a single cycle instruction at the end of each task.

Whilst a hardware TTC scheduler can provided a very predictable platform, there remains a key failure mode in TTC scheduling that has the potential to cause the system to become completely unresponsive. The next chapter will discuss this problem and provide a solution for this failure mode.

Chapter 6 Software Task Guardian

This chapter describes the design of a software task guardian mechanism to protect against the problems of task overruns in both TTC and TTH scheduling architectures.

6.1 Introduction

In Chapter 2 it was discussed that the predictability for real-time systems not only relies on functional requirements but also on the guarantees that the system will meet its deadlines. Due to this importance for temporal predictability and a common need to execute more than one task in a system, a time-triggered co-operative scheduling solution was chosen.

Whilst a time-triggered scheduler can have many benefits when it comes to temporal predictability, there exists one major failure mode which has the potential to greatly impair the system performance: this failure mode relates to the problem of task overruns.

In the event of a task overrun a problem may not even be detected (let alone resolved). This may have a serious impact on the system behaviour. For example, as Buttazzo has noted: *"[Co-operative] scheduling is fragile during overload situations, since a task exceeding its predicted execution time could generate (if not aborted) a domino effect on the subsequent tasks"* (Buttazzo 2005).

A minimal task overrun may only generate a tick offset error where the scheduler sequencing has been shifted by a small amount, however the remaining schedule recovers and may not be apparent to the user. A task overrun that is longer but still returns to the scheduler, often results in the system appearing very slow and sluggish. The worst case scenario is where a task overrun does not return and causes the system to hang indefinitely.

In summary, for a co-operatively scheduled system, any task that overruns has the potential to bring the whole system down to its knees. In a similar situation, a pre-emptive system may still be able to execute the higher priority tasks whilst the lower priority tasks remain blocked. As a result this can make a TTC system a less attractive solution, especially when considering safety critical systems. This chapter therefore aims to address the problem of task overruns by means of a flexible framework which provides a variety of mechanisms for dealing with overruns.

Before continuing, this chapter not only provides a task guardian solution for TTC architectures, but also includes the mechanisms to support a TTH scheduler which is described in Section 2.4. This is to allow the system to be used in a wider range of applications such as data acquisition systems which may require tasks that have short and long execution times to be scheduled within the same system.

6.2 Related work

There has been a considerable amount of work on the scheduling of systems in which there is an overload situation (Caccamo, Buttazzo et al. 2002; Cervin, Henriksson et al. 2003; Buttazzo 2005).

When considering the use of backup tasks which are called when a task overruns, some interesting work has previously been carried out in this area building on Jane Liu's publications on imprecise computations (Liu, Lin et al. 1987). This work involves running a simple version of a task first: this will be followed by a more complete version if there is time. One key difference between Liu's work and the approach presented here is that a static task schedule is used and backup tasks will only be employed if the main task fails.

Another related publication – with a focus on the Ada Ravenscar Kernel (Puente and Zamorano 2003) - outlines a technique for using a timer ISR routine to run a recovery task in the event of an overrun in a co-operative scheduler. Few details are provided but the approach appears to be similar to the original TTC Task Guardian design (Hughes and Pont 2004).

In relation to the Time-Triggered communication Protocol (TTP), a time-triggered network architecture has been designed for the supervision and protection of network communications (Bauer, Kopetz et al. 2003). The idea is to detect and recover from network errors in distributed systems. The work here differs because the intention is to detect and recovery from task errors and not just on the network communications. However, the failure of tasks may themselves lead to network errors and thus the proposed solution here may help to prevent against some of these errors.

In existing system designs, the main alternative to the techniques described here involve the use of a simple watchdog timer (Douglass 1997). For example, previous studies have described a set of design patterns which allow watchdog timers to be used in conjunction

with TTC embedded designs (Pont and Ong 2003). Such techniques can be useful in TTC designs, but cannot provide precise timing behaviour during recovery due to the long restart times to boot the system back into a working state.

6.2.1 Normal operation

In order to understand the effects of task overruns in the software TTC and TTH scheduling implementations the normal operations of the scheduler is described in this section.

During normal operation of systems using the TTC/TTH scheduler implementations, the first function to be run (after the startup code) is `main()`. Function `main()` then calls `Dispatch()` which in turn launches the co-operative task(s) currently scheduled to execute: it will be assumed in this discussion that a (co-operative) task - `C_Task()` - may be called. Once any co-operative tasks have completed their execution, `Dispatch()` calls `Sleep()`, placing the processor into a suitable "idle" mode. A timer-based interrupt occurs every millisecond (in typical implementations) which either wakes the processor up from the idle state or pre-empts a long co-operative task. In either case, the ISR `Update()` is invoked, by means of an assembly-language "wrapper" (in the version of this system used here). For TTH systems, `Update()` then directly calls the pre-emptive task (here it will be assumed that this is `P_Task()`). Once the pre-emptive task is complete, `Update()` increments a tick counter. The function calls then "unwind" back to `main()`, and `Dispatch()` is called again. The cycle thereby continues.

Please note that, in most designs, it would generally be expected that the pre-emptive task would occupy no more than approximately 10% of the tick interval.

6.2.2 Co-operative task overrun

As previously mentioned, the hybrid scheduler differs from the co-operative scheduler in that the assumption that co-operative tasks will complete within a tick interval is relaxed: that is, co-operative tasks are permitted to have a duration greater than a tick interval. However if there is more than one co-operative task scheduled to execute at any instant of time then the current task must finish before any other (co-operative) task can run (Figure 62).

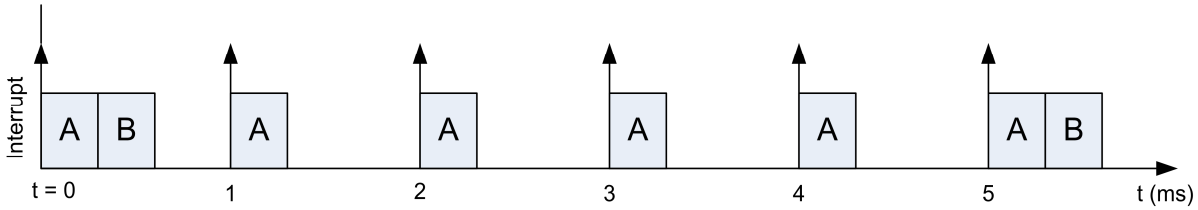


Figure 62: Typical co-operative schedule under normal conditions

When a co-operative task overrun occurs, then - instead of `Sleep()` being interrupted by the ISR - the overrunning task is interrupted. The pre-emptive task will still run at every tick, but all other co-operative tasks will be blocked (Figure 63).

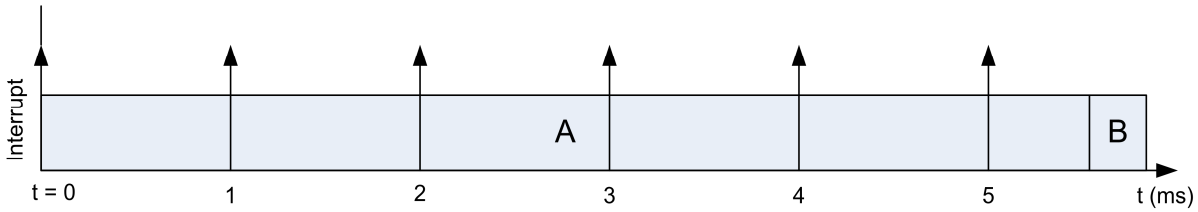


Figure 63: Overrunning of task A causes task B to be released late

6.2.3 Pre-emptive task overrun

If a pre-emptive task overrun occurs, then the co-operative tasks lose processing time within the tick interval(s). As a consequence, the co-operative tasks will have their execution delayed until the pre-emptive task returns (if ever) (Figure 64).

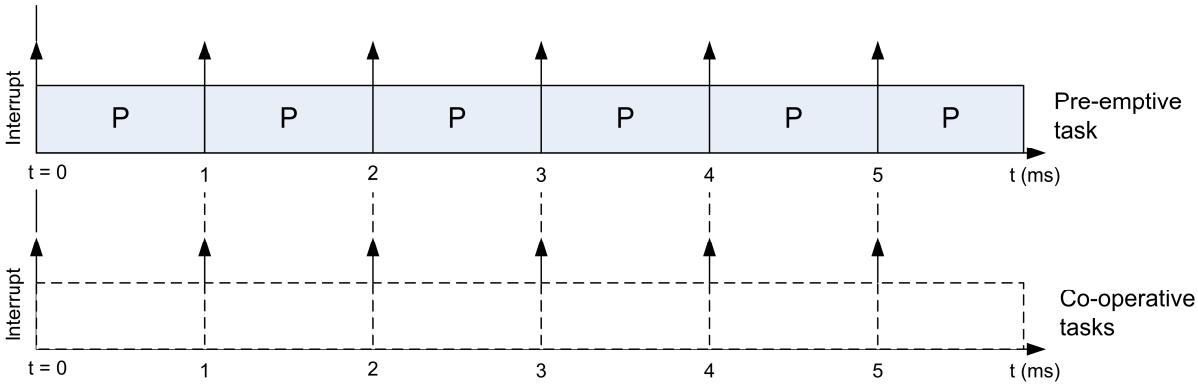


Figure 64: Overrunning pre-emptive task causes co-operative tasks to be blocked

6.3 Adding Task Guardians

In the TTH architecture, two Task Guardians are required: one for the co-operative tasks and one for the pre-emptive task. Ways in which these Guardians can be implemented are discussed in this section.

6.3.1 Co-operative task overruns

The Task Guardian for co-operative tasks is described first.

6.3.1.1 Providing WCET information

In a TTC scheduler, a task overrun can be detected comparatively easily, since - if a task is still executing when the next Tick occurs - the task has overrun. In a TTH scheduler, the situation is more complicated because one or more of the (co-operative) tasks may have been designed to execute for longer than a Tick interval: some additional information is therefore required if an overrun situation is to be detected.

The implementation presented in this chapter will assume that the user will provide the required information by indicating the expected WCET of each task (in μs) as it is added to the scheduler. Tasks will then be allowed to execute up to its expected WCET, where if found to exceed this time, the task is then shutdown. This WCET information is then checked by the Task Guardian implemented through a function called `SCH_Check_Tasks_OR()`.

To perform these checks, two timer match registers are employed to generate two interrupts on the timer IRQ line. Both interrupts are periodic and have the same period (equal to the Tick Interval: 1 ms in the studies discussed in this chapter). The Pre-Tick Interrupt occurs just before the Tick Interrupt: the interval between these events is the Pre-Tick Offset (and is set to 50 μs in the studies discussed in this chapter).

The Pre-Tick Interrupt is used to execute the `SCH_Check_Tasks_OR()` function and the Pre-Tick Offset period is set to a value such that the checks in the `SCH_Check_Tasks_OR()` function can be completed before the main Tick Interrupt occurs. The purpose of the pre-tick interrupt is to avoid the jitter and variable overhead that would be caused by `SCH_Check_Tasks_OR()` function. This concept is a similar technique to the planned pre-emption mechanism presented by (Maaita and Pont 2005).

The interrupt behaviour is implemented by means of an "IRQ wrapper" (in assembly language): this code is used to save (and restore) registers and call the appropriate function when a timer ISR is invoked (Figure 65).

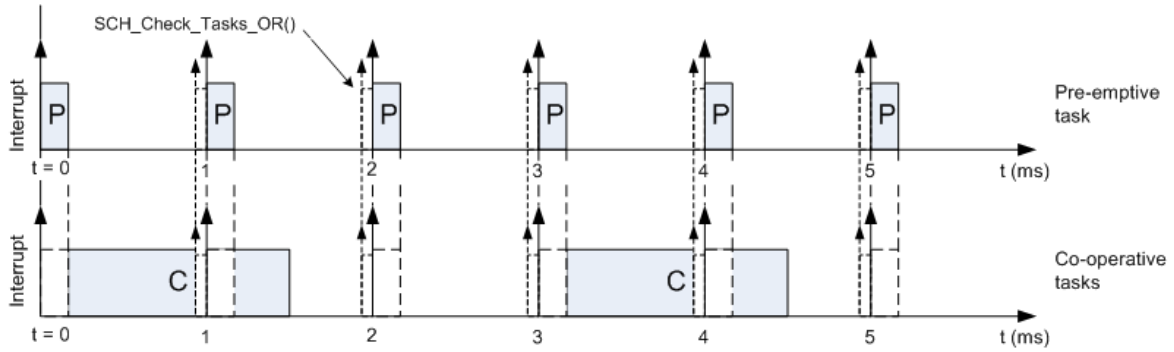


Figure 65: Task scheduling diagram with co-operative task guardian.

6.3.1.2 Desired behaviour

In the co-operative Task Guardian, the aim is to shut down any (co-operative) task found to have exceeded its predetermined WCET when the IRQ Wrapper is invoked. However if – as the result of a task overrun – some tasks in the tick interval have not been executed, these tasks need to be allowed time to execute. To allow this, the schedule is paused (for one Tick Interval), to allow any "blocked" tasks to execute. This then creates a predictable recovery time before the normal schedule continues.

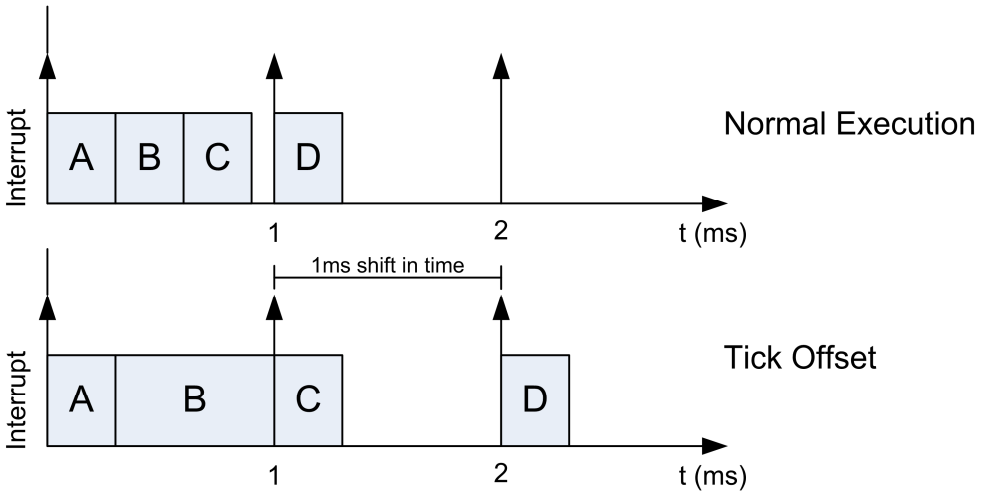


Figure 66: Paused tick offset to allow blocked tasks to execute before the system continues

6.3.1.3 Overview of the Task Guardian mechanism

The co-operative Task Guardian is implemented as follows:

- The IRQ Wrapper – when invoked by the "Pre-tick offset" interrupt – will call `SCH_Check_Task_OR()`
- `SCH_Check_Task_OR()` will check the WCETs of the tasks and (if necessary) flag that a task must be shut down.
- The IRQ Wrapper then places the processor into sleep mode.
- The tick interrupt then wakes the processor up from sleep mode and calls `Update()`
- For TTH implementations `Update()` then calls the pre-emptive task.
- Finally - based on the result flagged from the previous call to `SCH_Check_Task_OR()` - the IRQ Wrapper will either shut down the currently overrunning task or return normally from the interrupt.

The two main stages of the task guardian (detecting task overruns and returning from the IRQ wrapper), are considered in more detail in the following sub-sections.

6.3.1.4 Detecting task overruns (general mechanism)

In order for the `SCH_Check_Task_OR()` function to determine that an overrun has occurred and take appropriate action, a simple but reliable method is required to detect overruns. Modifying the code in `Dispatch()`, where the co-operative tasks are launched, enables this to be achieved. First the start and stop times of tasks are stored (before and after the task is called) so that the task duration can be recorded. A (single) variable (`Co_op_Task_Overrun_G`) is also used to indicate if a task is still running when the timer interrupt occurs and to identify the task concerned (see Listing 10). Note that the value 255 is used here as a reserved ID to indicate successful task completion (the value is arbitrary).

```
Co_op_Task_Overrun_G = Index;    // Store task ID
Start_Time = TMR0_TC();          // Get Start time of Task
EnableInt();                     // Enable timer interrupt
(*SCH_tasks_G[Index].pTask)();   // Run the task
DisableInt();                    // Disable timer interrupt
Stop_Time = TMR0_TC();           // Get Stop time of Task
Co_op_Task_Overrun_G = 255;      // Task completed
```

Listing 10: Overrun detection in Dispatch()

Due to the problem that an interrupt could occur anywhere in the scheduler code resulting in numerous systems states, there are three main areas in which an interrupt has been allowed to occur: (i) while a co-operative task is still executing; (ii) whilst `Dispatch()` is "between tasks"; and (iii) during sleep mode, when all the tasks in the tick have completed. Interrupts are – therefore - only enabled in pre-determined areas of the code for these specific conditions (before and after the task is called, before and after sleep and once in a suitable place in dispatch between task calls).

By detecting which of the three conditions the scheduler was in when the interrupt occurred, it becomes possible to take appropriate action. This is discussed in the following sub-sections.

6.3.1.5 Detecting task overruns 1 (interrupt an executing task)

In the condition where a task is still executing at the time when an interrupt occurred, the currently executing task is checked to see if it has exceeded its WCET, and if so, appropriate action is taken (Figure 67). To ensure this condition was not the result of a previous error the recently executed tasks are then checked to see if they had exceeded their WCETs. Finally another check is done to see if there are any tasks that have missed their execution deadlines as a result of this condition: if so, the co-operative tick is paused (for 1 Tick Interval) to allow the unexecuted tasks to run. After this "pause", the normal schedule continues. Under certain conditions an optional error report may be generated which is sent to a buffer which can be transmitted over a communication medium such as a UART.

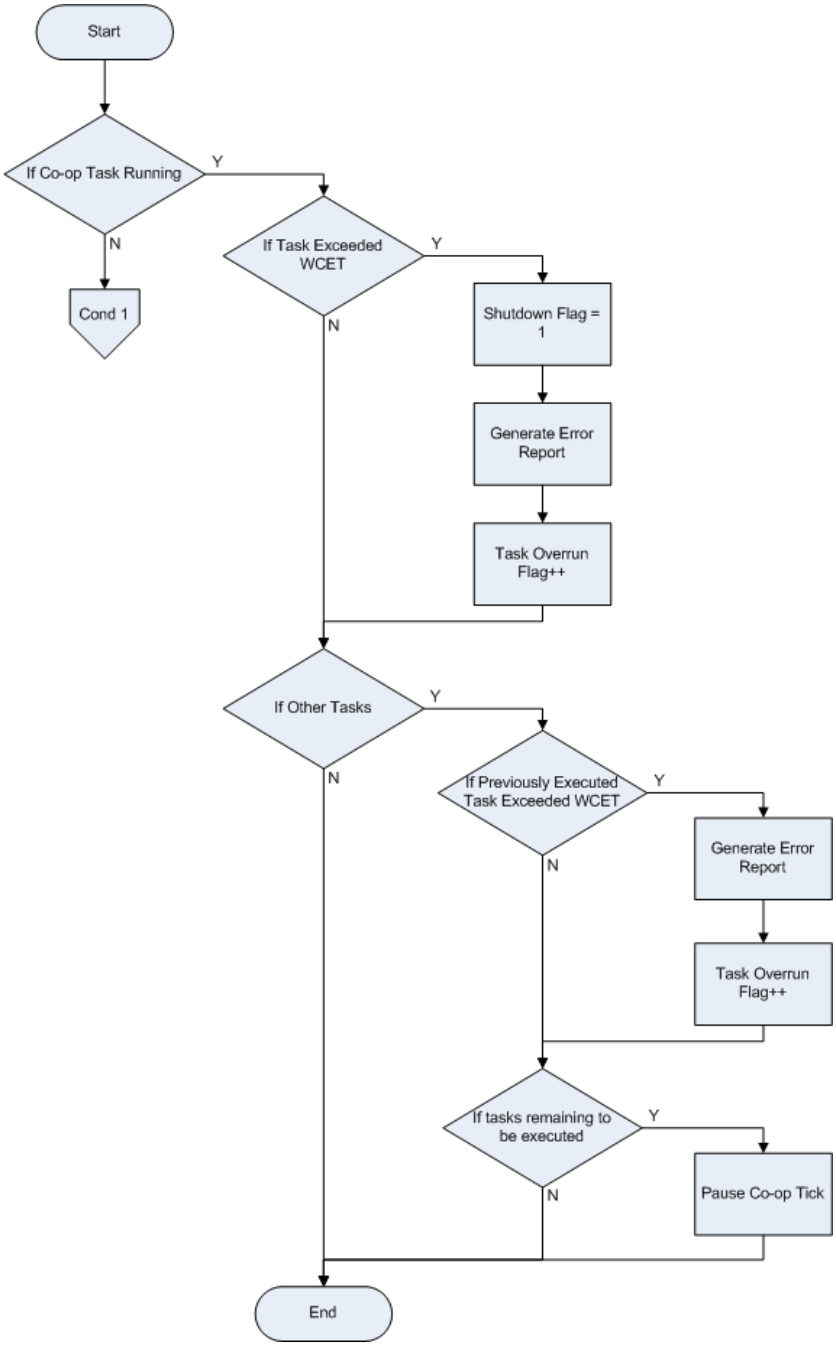


Figure 67: Flowchart of SCH_Check_Task_OR() when interrupting an executing task

6.3.1.6 Detecting task overruns 2 (Interrupt the scheduler in sleep mode)

When interrupting the scheduler in sleep mode, the system does a check on the previously executed tasks to identify if any tasks may have exceeded their WCETs when they ran (Figure 68).

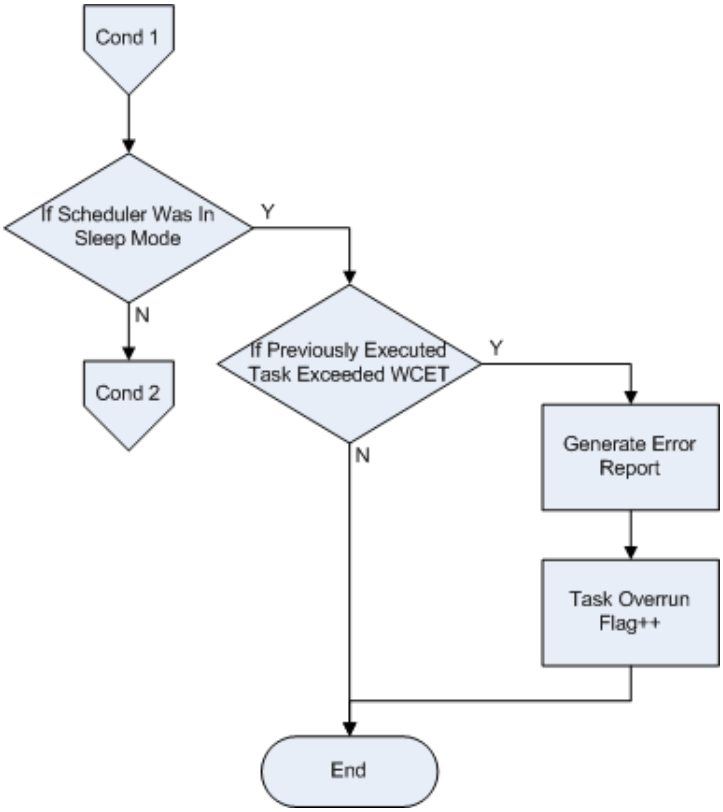


Figure 68: Flowchart of SCH_Check_Task_OR() when interrupting sleep mode

6.3.1.7 Detecting task overruns 3 (Interrupt the scheduler between tasks)

If a timer interrupt occurs when the dispatcher is "between tasks" this may mean that a particular task has overrun slightly, causing delayed execution of later tasks in this interval. If a "between tasks" event is detected, all previously-executed tasks are checked for overruns and the schedule is examined to see if there are unexecuted tasks remaining to be run. In the latter situation, the scheduler is paused (again for 1 Tick Interval) in order to allow the outstanding tasks to execute. The normal schedule then continues (Figure 69).

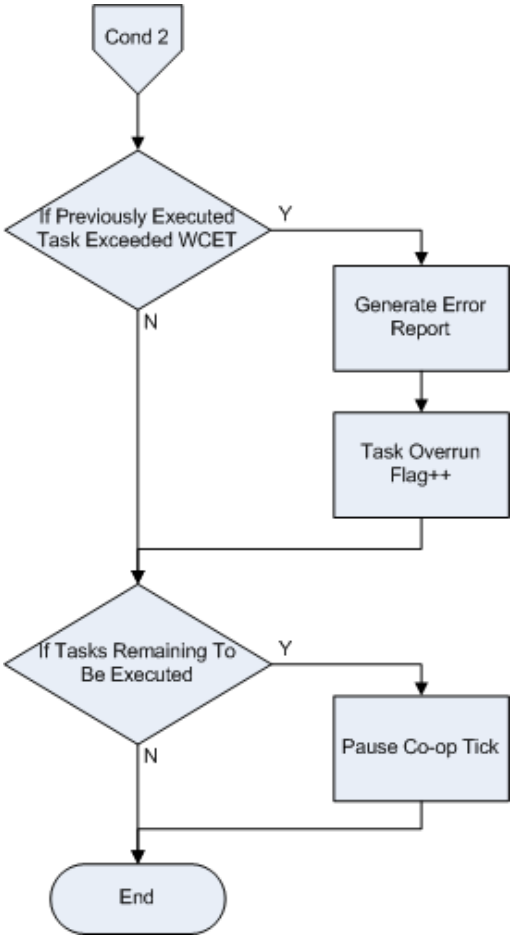


Figure 69: Flowchart of SCH_Check_Task_OR() when interrupting scheduler between tasks

6.3.1.8 Returning from the IRQ Wrapper

Once the `SCH_Check_Task_OR()` function has completed and the system has been awakened from sleep mode (by the timer interrupt), `Update()` and the pre-emptive task are then executed. Following this, `Update()` then returns to the IRQ wrapper where the decision to shut down a task is made, based on the value of the `Shutdown_Task` variable (which was set in the `SCH_Check_Task_OR()` function).

If the decision is made to shut down the task, then the IRQ Wrapper must alter its return address so that - instead of returning to the overrunning task - it returns to `Dispatch()` (Listing 11). This can be achieved by loading the address of a code label (pointing to the appropriate place in the `Dispatch()` function), into a general purpose register (such as 'r26'). Then, to perform a return, the program counter ('pc') register is loaded with the contents of 'r26' through a jump instruction.

```
# Load the dispatch return vector
la $26, Disp_Return_Ptr
lw $26, 0($26)
nop

# Return from handler
jr $26

nop
```

Listing 11: Return address known by use of code label in Dispatch().

Since the code returns to `Dispatch()` directly, there is no way of knowing which registers have been altered or been placed on the stack by the overrunning task. The `Dispatch()` function therefore contains two assembly macros which save the processor state before and after the task calls, allowing the `Dispatch()` function to continue regardless of any register changes.

6.3.2 Pre-emptive task overruns

With pre-emptive tasks, it is clear that – if the task is still executing at the time of the next Tick - it has overrun. In practice, detecting and handling such overruns requires care because pre-emptive tasks execute from a timer ISR: if the task overruns, subsequent interrupt requests (from this source) can be blocked. To resolve this, a second timer is used: this is set to an overflow time which is (i) slightly longer than the WCET of the pre-emptive task, and (ii) slightly shorter than the tick interval.

Overflow of this timer is linked to a separate interrupt which must be allowed to interrupt the current timer ISR.

Overall, the aim is to provide an effective TG mechanism without increasing the levels of task jitter under normal operating conditions. Using the methods described below, this is achieved by ensuring that there is always a fixed amount of tick available for the co-operative tasks.

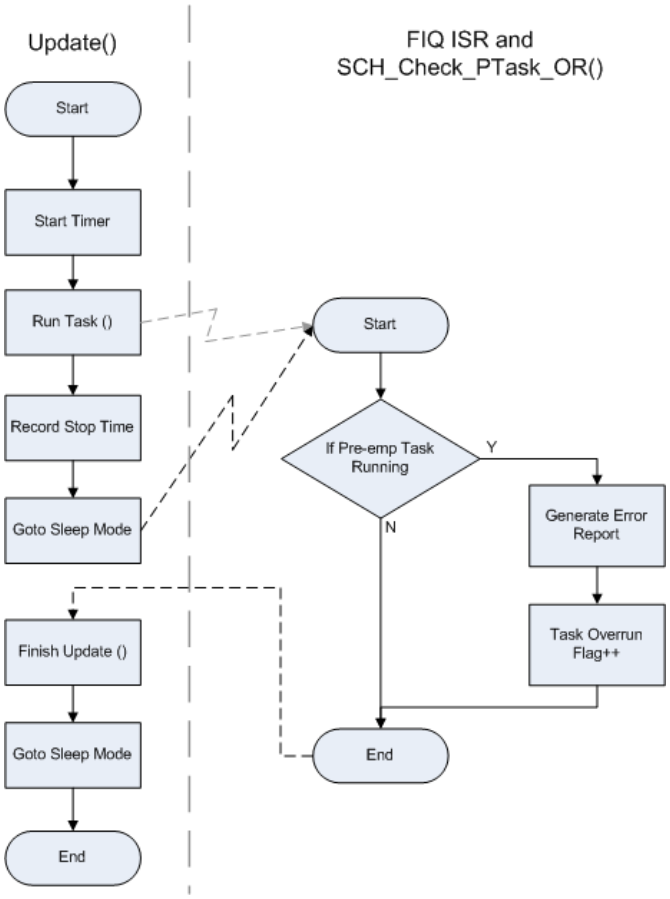


Figure 70: Flowchart of pre-emptive TG

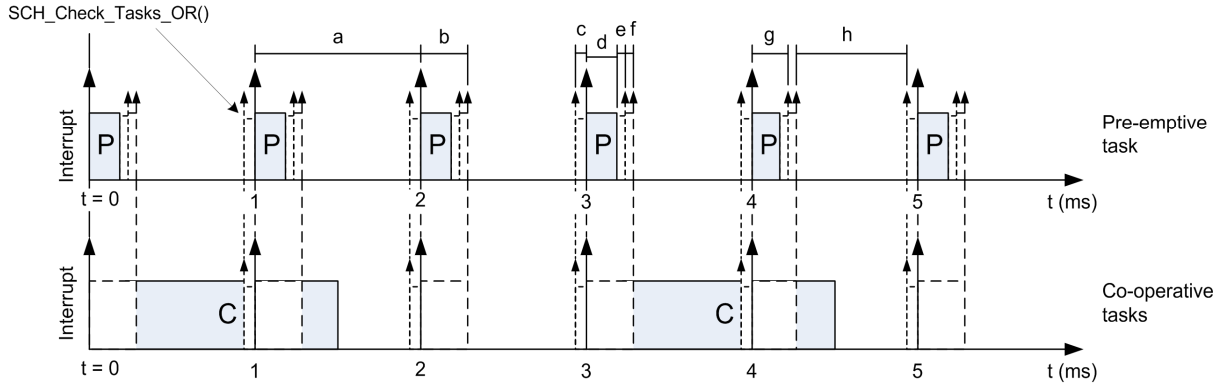
The second 'pre-emptive' timer is started before a pre-emptive task is called. Once the timer match value is reached, the IRQ Wrapper is invoked which in this instance calls `SCH_Check_PTask_OR()`. If the (pre-emptive) task is found to be executing then its overrun flag is updated and an error report is created (Figure 70). If however the (pre-emptive) task returned before the timer finished, then the processor is placed in sleep mode for the remainder of the timer duration. Either way, the `SCH_Check_PTask_OR()` is called and the ISR shuts down the code it has interrupted. Control is then passed back to `Update()`.

It is worth noting that, just before `Update()` finishes, the processor is once again placed into sleep mode. This encapsulates the pre-emptive task and pre-emptive task guardian into a fixed pre-emptive time frame which reduces the likelihood of jitter in the timing of the first co-operative task (Figure 71).

The remaining techniques used to create the pre-emptive Task Guardian are the same as those described for the co-operative version.

6.3.3 Overview of Task Guardian timing

For ease of reference, a summary of the timing measurements associated with the task guardian is given in (Figure 71). The large arrows represent the tick interrupts and the smaller dotted arrows represent the pre-tick and pre-emptive interrupts to create a fixed time frame which encapsulates the pre-emptive task. This process means that the co-operative task is interrupted by a fixed amount of time which is predictable and could be factored into timing analysis.



a. Tick interval
b. Pre-emptive time frame
c. Pre-tick offset
d. Actual execution time for a pre-emptive task
e. Pre-emptive task sandwich delay
f. Jitter compensation delay
g. Maximum allowed execution time for a pre-emptive task
h. Slack time (available for co-operative tasks)

Figure 71: Task scheduling diagram showing low jitter solution

6.4 Evaluating the basic TG mechanism

In this section results are presented from a study which was intended to assess the speed and response of the system in the event of a task overrun.

6.4.1 Overview of the study

In this design, a Tick interval of 1ms was used and a fixed period of 20 μs was allocated in each Tick interval for processing the pre-emptive task.

For test purposes, two tasks were created to generate task overruns at pre-determined times.

The co-operative task was set to alternate one of the LEDs on the development board every time it was called. This function will also count the number of times it was called, when a count value of 5 was reached, an infinite `while` loop is executed to generate a task overrun. Note that, under normal conditions, the execution time of the co-operative task was approximately 2.36 µs.

The pre-emptive task simply pauses for a short while to represent some load and a variable counts the number of times the task was called: after 500 calls, an infinite loop is executed (Listing 12). Please note that "`Function_A()`" is used to demonstrate that the Task Guardian can successfully shut down tasks with sub functions.

```
void Premp_Update(void)
{
      Function_A();
}

void Function_A()
{
      volatile int i;

      for (i = 0; i < 10; i++);

      if (++State == 500)
      {
            State = 0;
            while (1);
      }

}
```

Listing 12: Pre-emptive task will generate a task overrun every 500 times it is called

Please note that the code in these tests was compiled using the GCC 3.3.3 compiler without optimizations set.

6.4.2 Timing behaviour

Timing measurements were taken at key points in the systems execution by using a hardware CPU cycle counter and sending its values over a serial connection to the host. These values contain the cycle time from the beginning of the tick interrupt to the associated event, as can be seen in Table 14. One of the more prominent values is the time to the beginning of the first co-operative task as this reflects the overhead of the scheduler and any pre-emptive tasks before the CPU time is handed over to standard TTC tasks.

Event	Measured Time (μs)
Start of Pre-emptive Task	7.56
End of Pre-emptive Task	13.44
End of Update	61.80
Start of 1st Co-operative Task	74.52
End of 1st Co-operative Task	76.88

Table 14: Measured times during key event in the basic TG operation

In addition, values were recorded for the time between the first and second co-operative tasks as well as the overhead imposed by check task overrun function (Table 15).

Event	Measured Time (μs)
Time between 1st and 2nd Co-operative Task	14.20
Check Task Overrun Overhead	12.52

Table 15: Measured overheads during normal conditions of the basic TG

When testing the basic task guardian mechanism under different overrun conditions it can be seen that the time until the first co-operative tasks is unaltered (Table 16).

Condition	Scheduler Load (μs)
No Overrun	74.52
Pre-emptive Overrun	74.52
Co-operative Overrun	74.52
Both Overrun	74.52

Table 16: Scheduler loads during task overrun conditions

When examining the results shown in (Table 16), it should be noted that – in all cases – 20 μs of the load comes from the fixed available execution time for the pre-emptive task and a further generous 30 μs arises from the Jitter Compensation Delay. Even including this fixed (50 μs) load, the scheduler can shut down an overrunning pre-emptive and / or co-operative task within 74.52 μs from the beginning of the tick. More specifically, the shut down time is 24.52 μs greater than the fixed available pre-emptive execution time and has no effect on the jitter behaviour of the first executing co-operative task.

For almost all practical applications, this represents an extremely fast (and highly predictable) response time.

6.4.3 Implementation costs

The focus in this thesis is on applications which must have highly predictable patterns of behaviour and ideally low resource requirements. Both the memory and CPU requirements of the modified software scheduler are considered in this section.

Table 17 compares the memory requirements for the Base and Task Guardian schedulers. It can be seen that the implementation of the Task Guardians requires a significant amount of additional code for the main scheduler functions. However, even with this additional code, the total data memory requirement is 318 bytes, and the total code requirement is 6352 bytes.

Scheduler	Code Size (bytes)	Data Size (bytes)
Base	2216	108
Basic Task Guardian	6352	318

Table 17: Code and data memory requirements for the "Base" and "Task Guardian" TTH schedulers

It is also important to consider the CPU load imposed by adding the Task Guardians. To illustrate this, Table 18 shows the amount of the tick interval used to process the scheduler code in normal conditions (with empty tasks and no task overruns). As previously noted, the Task Guardian in this case study fixes 20μs of the tick interval to process the pre-emptive task and 30μs for a short pause at the end of `Update()`. Even taking this into account, the modified framework requires only a total of 74.52 μs of the available CPU time in a tick interval (under normal conditions).

Scheduler	Scheduler Load (μs)
Base	8.60
Basic Task Guardian	74.52

Table 18: Scheduler loads per tick interval

6.5 Adding support for backup tasks and "allowed overruns"

The ability to shut-down tasks that overrun and exceed their WCET (as described in Section 6.3), has the potential to improve the reliability of TTC and TTH designs. However, avoiding scheduler "jams" may not address the underlying problem: indeed, if a critical operation is rendered inoperative through the shutdown process, the system reliability may not be improved (at all).

There are a number of ways in which the functionality of the Task Guardian can be extended in order to address this problem:

- Support for "Backup tasks" can be provided (such tasks to be invoked in the event of the failure of a critical task).
- A task that is still running at the next tick interval can be shut down even if it has not exceeded its WCET but its continued execution would prevent another task from starting on time.

This section describes these features in more detail.

6.5.1 Backup Tasks

To support backup tasks, a second function pointer is required in the task array (for each task). This will hold the address to the backup task (if one exists). When – in the `SCH_Check_Task_OR()` function – a co-operative task overrun is detected, the backup-task address is checked. If a non-zero value is found, the original task address is replaced with the address of the backup. The scheduler then executes the backup task immediately (before running any other pending tasks).

To reduce the problems of a domino effect (caused by the insertion of a new task in what may be a full schedule), a period of grace (of 1 Tick Interval) is allowed for the backup task and the completion of tasks which may have been unable to execute in the same tick interval as the offending task.

Please note that, in the framework described here, the recovery time is known (1 Tick Interval, from the task overrun being detected to the return to the normal schedule). It is also known that the worst case overrun detection time is also 1 Tick Interval. Therefore in the case where a co-operative task has overrun by a small amount (measured in microseconds), it is only possible to react within the period of 1 Tick Interval.[15]

The pre-emptive backup task operates slightly differently. Recall that, if the pre-emptive task exceeds its pre-determined WCET value, it will be shut down and control will be

[15] This behavior is not ideal. However, the cost of supporting more rapid detection of overruns of the co-operative task would be considerable (it would be necessary to perform overrun checks – for example – ten times in every tick interval: this would impose a substantial CPU load).

passed back to `Update()`. At this point, to reduce jitter in the timing of the co-operative tasks, the scheduler goes to sleep mode for a short period (the Pre-emptive Delay). This results in a pre-emptive time frame which will always have a duration equal to the WCET of the pre-emptive task plus the Pre-emptive Delay (30 μs in the present system). If a pre-emptive backup task (labelled P_b in these discussions) is included in the schedule then - in the case where the pre-emptive task exceeds its WCET - the backup will be run immediately and will take up exactly two pre-emptive time frames. For example, if the system has a pre-emptive task and backup which each have WCET values of 50 μs, then the "double pre-emptive frames" slot would have a total duration of 160 μs (that is, WCET times of the two tasks + 2 x Jitter Compensation Delay). This worst-case figure should be taken into account when setting the schedule for the co-operative tasks.

6.5.2 Allowed Overruns

As has been discussed, incorporating backup tasks without causing some disruption to the schedule is a practical impossibility. One way around this problem is referred to here as an "Allowed Overrun" mechanism for co-operative tasks. What this means is that, if a co-operative task overruns, the designer may opt to allow it to carry on until either its WCET is reached or another (co-operative) task is due to execute. The aim is to allow tasks to continue to run, so long as it does not impact on other tasks. This might be particularly useful if (because of inaccurate WCET estimates) a task may overrun by a small amount.

For a task to be allowed to overrun, the scheduler must be able to determine when the next task is due to execute. The framework in the TTC and TTH schedulers determines the execution times by decrementing delay counters for each task in the `Dispatch()` function: when a delay of 0 is reached, the task is executed and the delay counter is reset. By keeping a running count of how many ticks a task has overran and comparing this with the delay values of the tasks in the `SCH_Check_Task_OR()` function, it is possible to determine when a task will next be executed.

6.6 Evaluating the extended TG mechanism

As with the basic Task-Guardian framework in Section 6.4, a study was conducted with the extended framework, described in Section 6.5, in order to assess the speed of response in the event that there are one or more task overruns.

6.6.1 Overview of the study

The study was conducted in the manner described in Section 6.4 using the same basic task set. Please note, however, that in this study, backup tasks were also employed.

The co-operative and pre-emptive tasks were set to run (with no offset) every 2 ms, as shown in Listing 13. Both tasks were scheduled to execute in the same time slot so that measurements could be taken for the worst-case situation (in which both tasks overrun simultaneously).

```
// Def: SCH_Add_Task(Task, BackupTask, Delay, Period, WCET, Premp);

// Pre-emptive task
SCH_Add_Task(Preemptive_Task, Preemptive_Task_Bkp, 0, 2, 20, 1);

// Co-operative task
SCH_Add_Task(Cooperative_Task, Cooperative_Task_Bkp, 0, 2, 100, 0);
```

Listing 13: Scheduling the tasks and backup tasks

6.6.2 Timing behaviour

When looking at the times of the key events in the extended task guardian it can be seen that the overhead is slightly increased to accommodate the extra functionality (Table 19).

Event	Measured Time (µs)
Start of Pre-emptive Task	7.08
End of Pre-emptive Task	14.08
End of Update	61.32
Start of 1st Co-operative Task	77.56
End of 1st Co-operative Task	79.16

Table 19: Measured times during key event in the extended TG operation

In addition, the overhead between co-operative tasks and the check task overrun function has also increased slightly (Table 20).

Event	Measured Time (µs)
Time between 1st and 2nd Co-operative Task	16.32
Check Task Overrun Overhead	14.92

Table 20: Measured overheads during normal conditions of the extended TG

Unlike the basic task guardian mechanism, the extended task guardian has some additional overheads when backup tasks are executed (Table 21). In the case of a pre-emptive backup task being executed, the additional overhead is predictable in that it requires exactly one extra pre-emptive time frame. When a backup co-operative task is run the additional

overhead is less obvious but is static and can be measured or calculated by static code analysis. In this study this overhead was measured as 3.04 μs.

Condition	Scheduler Load (μs)	Difference (μs)
No Overrun	77.56	0
Pre-emptive Overrun	77.56	0
Pre-emptive Overrun + Backup	127.56	+50
Co-operative Overrun	77.56	0
Co-operative Overrun + Backup	80.6	+3.04
Both Overrun	130.4	+53.04

Table 21: Scheduler load during task-overrun conditions

As the table makes clear, the additional features lead to a very small increase in the scheduler load (3.04 μs under normal operating conditions). The loads for the various overrun conditions are summarised in Table 22. Note that the Dispatch Delay load may vary if the first task in a tick interval is not the same.

Condition	Load Equations
Normal Conditions	Scheduler load = (Pre-emptive Time Frame) + Dispatch Delay
Pre-emptive Backup	Scheduler load = (2 × Pre-emptive Time Frame) + Dispatch Delay
Co-operative Backup	Scheduler load = (Pre-emptive Time Frame) + Co-operative Backup Overhead + Dispatch Delay
Both Backup	Scheduler load = (2 × Pre-emptive Time Frame) + Co-operative Backup Overhead + Dispatch Delay

Table 22: Scheduler overhead equations[16]

The user should use the load equations to predict the worst case scenario for a schedule and apply this with the WCET information to ensure that the scheduler is not overloaded under normal or task overrun conditions.

6.6.3 Implementation costs

As noted in Section 6.4.3, it is important to understand the implementation costs of the mechanisms required to implement backup tasks and allowed overruns.

16 Measured values for scheduler overhead equations: Dispatch Delay = 27.56μs; Co-operative Backup Overhead = 3.04μs

Table 23 shows the impact on the memory requirements from the addition of the backup function and allowed overrun features. Again there is a significant increase in code size, however there is only a small increase in data requirements.

Scheduler	Code Size (bytes)	Data Size (bytes)
Base	2216	108
Basic Task Guardian	6352	318
Extended Task Guardian	8648	366

Table 23: Code and Data memory requirements for the Base and Task Guardian Schedulers

Although there is additional code, the CPU requirements (in normal operation) are increased by only a very small amount (see Table 24).

Scheduler	Scheduler Load (µs)
Base	8.6
Basic Task Guardian	74.52
Extended Task Guardian	77.56

Table 24: CPU loads for the various TTH schedulers

6.7 Further applications of the TG approach

In this section TTC and TTH implementation of the techniques described in this chapter are compared.

6.7.1 A TTC implementation

The process to convert between the complete TTH framework (with support for backup tasks and allowed overruns) into a TTC Scheduler framework can be done simply through the use of a compiler '#define' directive. In the TTC version, there is no support for pre-emptive tasks (and, therefore, no Task Guardian for such tasks).

The memory requirements of the TTC and TTH designs are compared in Figure 72. The comparisons of the CPU loads are shown in Figure 73.

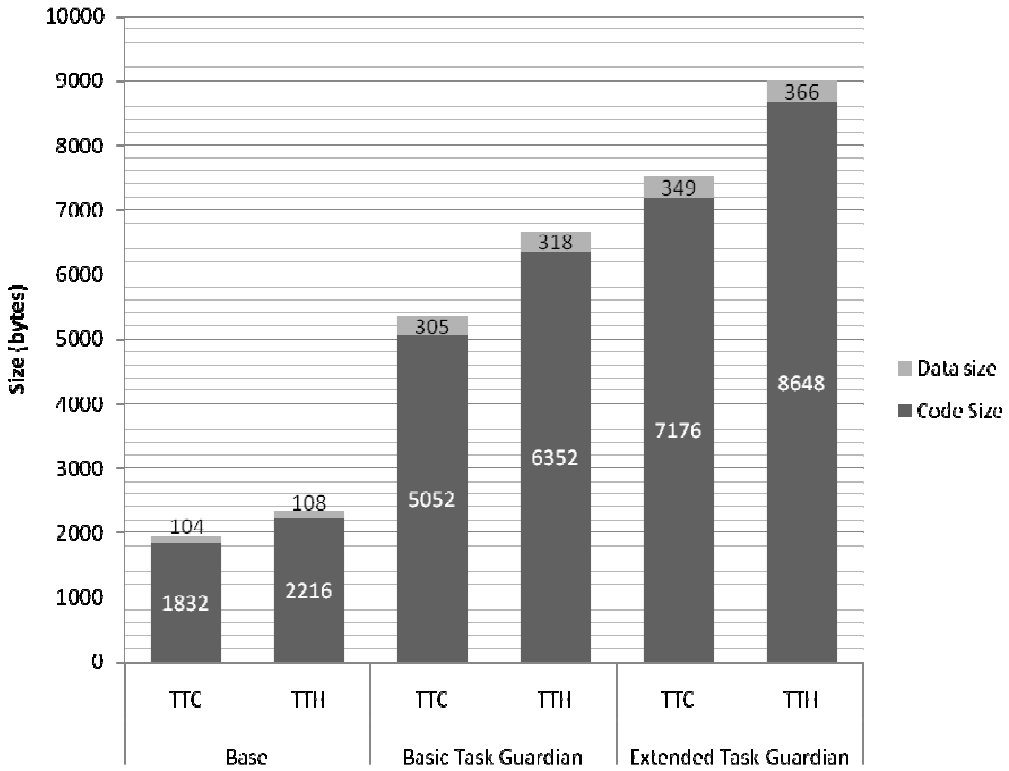


Figure 72: Comparison of Code and Data memory consumption for the software TTH and TTC schedulers

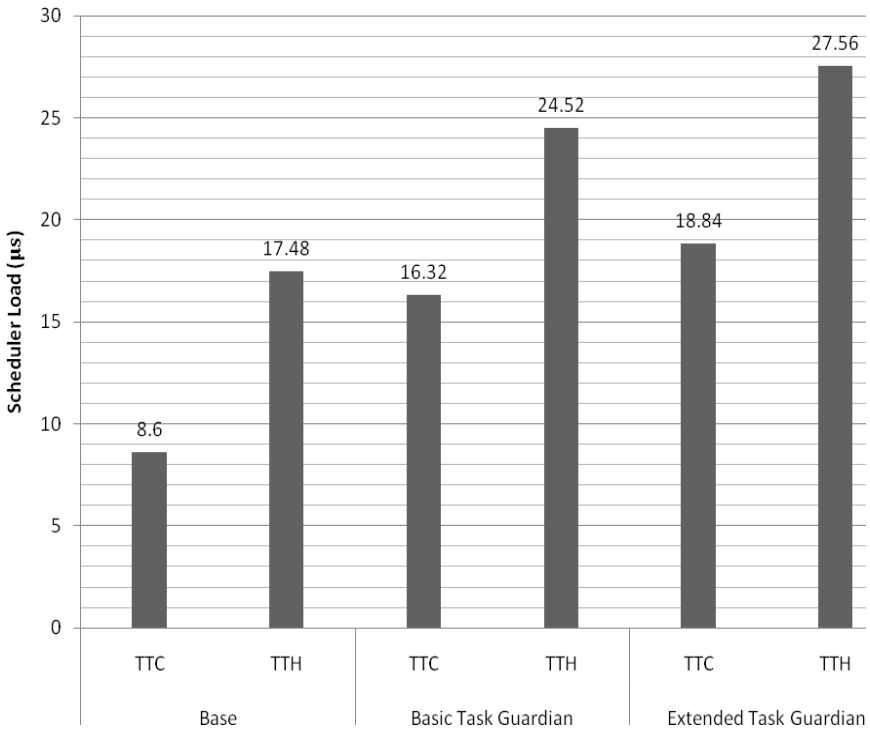


Figure 73: Comparison of CPU loads for the software TTH and TTC schedulers

6.7.2 Dealing with the underlying causes of task overruns

Task overruns do not only arise from inaccurate estimates of WCET. For example, electromagnetic interference can cause code and data corruption (Ong and Pont 2002) which could lead to task overruns. In many cases, hardware features on the processor may be employed to detect such errors and could make use of the TG mechanisms to handle them. A similar form of generic error-handling mechanism (for watchdog timers) has been described (Pont and Ong 2002).

The PH processor used in the examples in this chapter has four main "exception" mechanisms: Undefined Instruction, Arithmetic error, Instruction Abort and Data Abort.[17] The Task Guardian implementation described in this chapter can be set to deal with all four exceptions in the same manner that it deals with task overruns. On entering an exception the handler will record the return address of the instruction that generated the exception (which is printed along with the exception type by a generated error report). The handler then flags an exception has occurred and places the processor into sleep mode to await the next `SCH_Check_Task_OR()` interrupt: this function then checks the exception flag and shuts down the offending task. The TG treats the exception in the same manner as a normal overrun, and therefore backup tasks (if any) are executed. All other TG features remain available when this form of exception handling is employed.

6.8 Discussion

When a TTC scheduler is used the behaviour can be very predictable. However, in order to expand the range of applications that a TTC scheduler can be used, a "time-triggered hybrid" (TTH) scheduler can be employed as an alternative to a fully pre-emptive design.

Both systems can suffer due to the condition of task overruns. To address this problem, this chapter presented a Task Guardian mechanism which can be employed to deal with both co-operative and a single pre-emptive task. The task guardian provided a way to shutdown an overrunning task and execute - if requested - a backup task. The benefit of

[17] Data Abort errors occur when an attempt is made to load data from (or store data in) an invalid memory location. Instruction Abort errors occur when an attempt is made to fetch an instruction from an invalid memory location. An Undefined Instruction errors occur when an attempt is made to execute an invalid instruction. An Arithmetic error occurs when a signed value overflows.

the backup task was that the user could define a specific recovery mechanism for each task. The resulting framework has been shown to have predictable behaviour, even in the event of task overruns.

Inevitably, there are costs involved in applying these approaches. For example, knowledge of task execution times and timing characteristics of the scheduler are required in order to apply the equations presented in Table 22. In particular, for the basic TG mechanism, the developer needs to ensure that the pre-tick interval and the jitter compensation delay are sufficiently large for the guardian to execute. For the extended TG mechanism, the Pre-emptive Time Frame, Pre-emptive Backup Overhead, Dispatch Delay, and Co-operative Backup Overhead need to be defined. If these measurements are not accurate then the reliability of the system could be compromised.

Even though the task guardian provided a number of useful features, this came at a high price. For instance, the code size was 3.9 times larger than the software scheduler and the data size was approximately 3.36 times larger. This not only represented a large increase in memory requirements but also a significant amount of complexity. A timer with 2 match registers was also required in order to provide a low jitter solution by running a 'check task overrun' function in a separate interrupt handler before the main tick interrupt occurred. The result was that the scheduler overhead was 2.2 times larger without including the extra time required for the 'check task overrun' function. The between task overhead was also increased by 3.4 times than that of the software scheduler.

Whilst the software task guardian was large and complex, it was designed to achieve a good level of temporal predictability. However, it could be considered that because the system is large and complex the the effort required to verify the system meets its safety standards could render the use of a software task guardian to be impractical.

6.9 Conclusions

Throughout this thesis the goal has been to provide a way of constructing highly predictable embedded systems. This chapter has sought a software solution to the task overrun problem through the implementation of a task guardian. The software mechanism achieved this goal but has substantially increased the amount of code required to implement the system. This in turn increases system costs and complicates both the

development and maintenance processes for the system. In addition, the developer must correctly configure the software mechanism or they may actually reduce the system reliability rather than improve it. Furthermore, the implementation is tied tightly to the features of the microcontroller.

The work in this chapter differs from most previous approaches in that no attempt is made to create an on-line scheduling algorithm which can adapt to deal with an overload situation: instead, the aim is to follow a pre-determined (static) schedule as closely as possible and to either shut down (or replace) a task that does not meet its pre-determined WCET constraints. Providing this kind of mechanism is advantageous so that the error detection and recovery times can be done quickly and in a known predictable time.

Due to all the associated complexities of the software task guardian mechanism, the next chapter looks at providing a hardware solution which can be integrated with the hardware scheduler mechanism as described in Chapter 5.

Chapter 7 Hardware Task Guardian

This chapter aims to look at a hardware alternative to the software task guardian solution. The solution presented here will be integrated with the predictable processor and hardware scheduler designs as outlined in the previous chapters. A comparison will be then drawn between the hardware solution and the software mechanism presented in Chapter 6.

7.1 Introduction

One of the main goals of a task guardian component is to guarantee task execution time by ensuring each tasks allocated block of CPU time will always be available to it, especially if any other task attempts to exceed its allocated time. To minimise the effect of an overrunning task on subsequent tasks, involves the need for quick error detection. Thus preventing CPU overload and ensuring that the recovery mechanism is fast so that the tasks can collectively meet their deadlines. This highlights one of the problems with the software task guardian mechanism where a task overrun is only checked at the end of each tick interval. As such, a task may overrun by up to 1 millisecond before being detected. This may result in several slack stealing tasks being forced to run in the next tick interval in order to catch up with the desired schedule. In hardware, this issue is no longer a problem as the overrun detection mechanism can occur in parallel to the executing task. As such, this presents some interesting design choices to the way the hardware task guardian can be implemented.

With the ability of the hardware to see what tasks are due to run next and quickly respond to an overrun event and take the appropriate action, the need to shutdown the task immediately can be relaxed. For instance, the guaranteed processor time (GPT) might be set to the task's WCET. However, the design can have the additional flexibility in that it does not require precise information about the WCETs for the tasks that it is monitoring. This is because the task may be allowed to keep running (even if this execution takes longer than might have been predicted – i.e. longer that the guaranteed processor time) while no other task requires access to the CPU. Accordingly, this feature can help to simplify the system design process. In addition, it may help to ensure that a system operates in a reliable way and could also significantly reduce the opportunities for programming errors to affect the system.

While the advantages of allowing a task to exceed its guaranteed processing time when it will have no impact on other task can have many benefits, there does come a point or upper time limit where the user would prefer a recovery action to be taken. For example, if a task has a WCET of 100μs but the next task in the schedule does not execute until 500ms away, then the task could overrun for a substantial amount of time before anything is done about the problem. Therefore, the hardware mechanism can be provided with an additional variable, the maximum allowed overrun time (AOT). This means recovery will be guaranteed to happen between the GPT and AOT times.

Finally, the hardware mechanism will include the ability to execute a backup task after the current task has exceeded its GPT time. If no backup task is provided, then if specified the task can run up to its AOT.

7.2 Task Guardian Component

This section briefly describes the functionality of the core of the hardware task guardian mechanism.

7.2.1 Task information block

The task guardian registers are interfaced to the processor through the address and data bus to load the task information along with the TT hardware scheduler unit. The current layout of the task information block for the TT hardware scheduler unit contains the information detailed in Table 25.

Task Variable	Type	Description
Vector	Long	Task address vector
Delay	Short	Delay until task executes
Period	Short	Period between task executions
Enable	Bit	Enable task to be scheduled

Table 25: TT Hardware Scheduler unit task information block

To support the task guardian unit three additional variables must be added to the task information block. These include a guaranteed processor time 'GPT' variable, which

contains the maximum execution time that the specified task will be guaranteed in each task period. The second variable is an allowed overrun time 'AOT' up to which the specified task may run when its additional execution time (i.e. that over and above its GPT) does not coincide with any pending tasks[18]. The third additional variable is a backup task vector address 'bVector'. When this contains a non-zero value the backup task at the specified address will be executed in the event that the original task does not manage to complete within its GPT time limit. The backup task is then allowed to execute for the duration specified in the AOT. These variables are contained together with the TTC hardware scheduler to form the task information block detailed in Table 26.

Task Variable	Type	Description
Vector	Long	Task address vector
Delay	Short	Delay until task executes
Period	Short	Period between task executions
Enable	Bit	Enable task to be scheduled
GPT	Long	Guaranteed Processor execution Time (CPU cycles)
AOT	Long	Allowed Overrun execution Time (CPU cycles)
bVector	Long	Backup task address vector

Table 26: TT Hardware Scheduler + Task Guardian task information block

Whilst these three extra variables appear to the programmer as part of the hardware scheduler unit, they are actually contained within registers of the task guardian unit. Figure 74 illustrates the structure and some key input and output control signals involved in the operation of the hardware task guardian unit.

[18] Note, that the GPT and AOT are specified in CPU cycles as these are the types of values that can be obtained by static analysis and are not affected if features like dynamic frequency scaling are used. For a CPU clock frequency of 25MHz, the maximum duration that can be specified is 2.86 seconds.

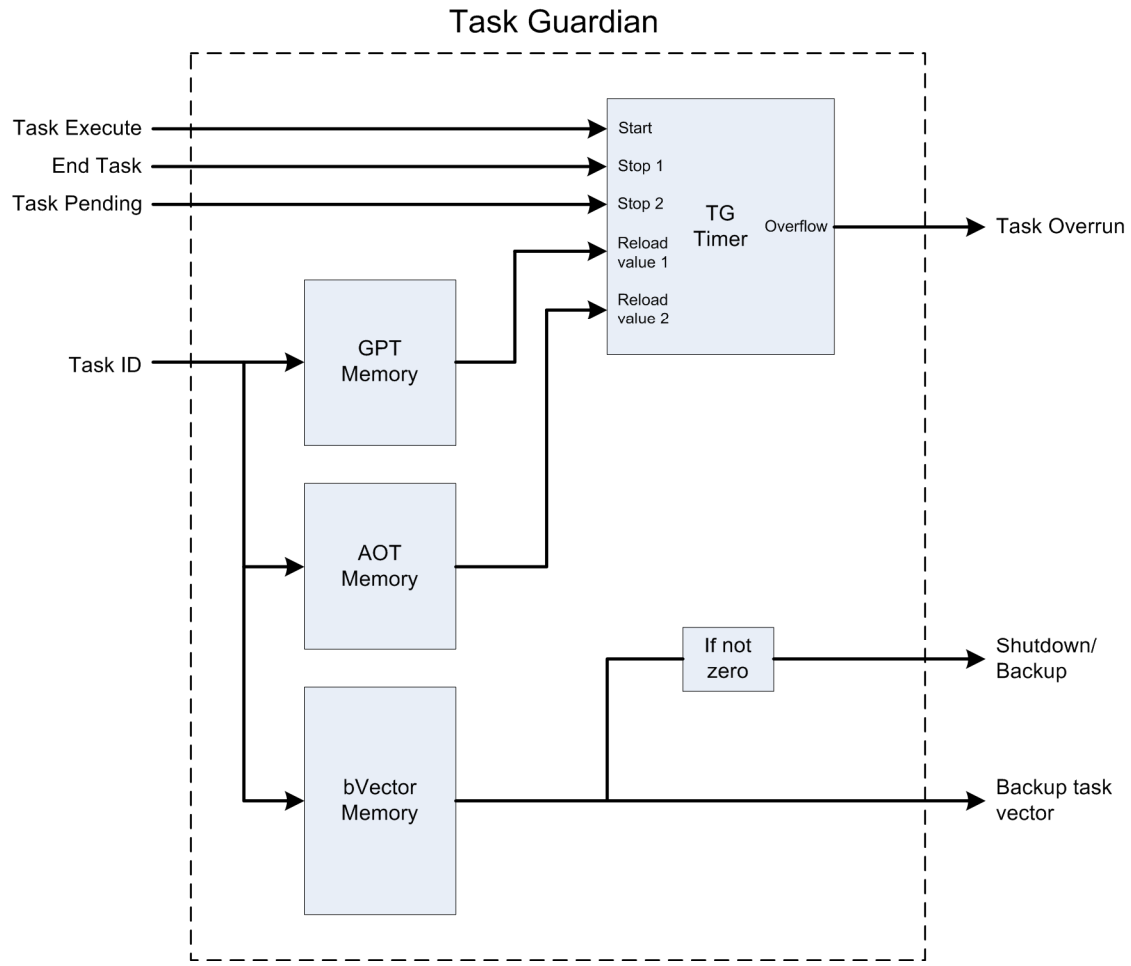


Figure 74: Schematic overview of the hardware task guardian unit

In order to fully appreciate how the task guardian works it is important to review the basic operation of the hardware scheduler as presented in Chapter 5. The hardware scheduler contains two major components referred to as 'Update' and 'Dispatch'. A timer generates tick intervals (typically 1 ms in duration) by sending signals to the Update component. On receipt of each signal, the Update component checks if there are any tasks due to run in the current tick interval. Upon detecting that a task is due to run, the task is added to a First In First Out (FIFO) queue. The role of the Dispatch component is to then execute each task as it comes out of the FIFO queue until the queue becomes empty. The tasks are executed by sending a task execute signal to the processor which causes its program counter (PC) to be loaded with the vector address of the current task. An example of a resulting schedule is shown in Figure 4 for tasks A, B, C and D.

To support the task guardian, the task IDs which are stored in the FIFO queue are also used to reference the current task's GPT, AOT and backup task vector. In addition, the task execute signal from the Dispatch component causes the task guardian timer to be loaded with the stored GPT and AOT values and the timer is then started.

If the AOT is set to zero, two conditions can occur - either the task completes before the GPT is reached or the GPT is reached and an error-recovery mechanism steps in.

When a task completes, an 'endtask' instruction is executed signifying to the Dispatch component that it can proceed to execute the next task in the FIFO queue. In the scenario when a task completes in time, the end task signal is also routed to the task guardian component where it causes the task guardian timer to be stopped.

Alternatively, in the scenario when the GPT has been reached, the task overrun signal indicates to the Dispatch component that the task should have ended. The next action then depends on the outcome of the 'shutdown / backup' signal. If this simply indicates that the task is to be shutdown then the Dispatch component stops the current task and continues to execute the next task in the FIFO list. However, if no tasks are due to run, the processor is put into sleep mode.

If the task guardian indicates that a backup task should be executed, the Dispatch component then sends another task execute signal to the processor, but this time loading the program counter with the backup task address.

If the AOT is not equal to zero then, in the event that the GPT has been exceeded, the task guardian timer will keep running until either the AOT is reached or a task pending signal is received. In either case the recovery mechanisms described above will operate unless the task ends before either event (i.e. before the AOT is reached or before a task pending signal is received).

Note that identifying pending tasks is achieved by reading the 'queue not empty' signal from the task FIFO in the hardware scheduler unit.

Figure 75 represents the above functionality of the task guardian, in terms of a flow diagram. The actions in the flow diagram execute every tick interval.

The task guardian component may also include a "task overrun register" for each task. This can be configured to maintain a value that indicates the number of times that the task has exceeded its GPT and/or AOT.

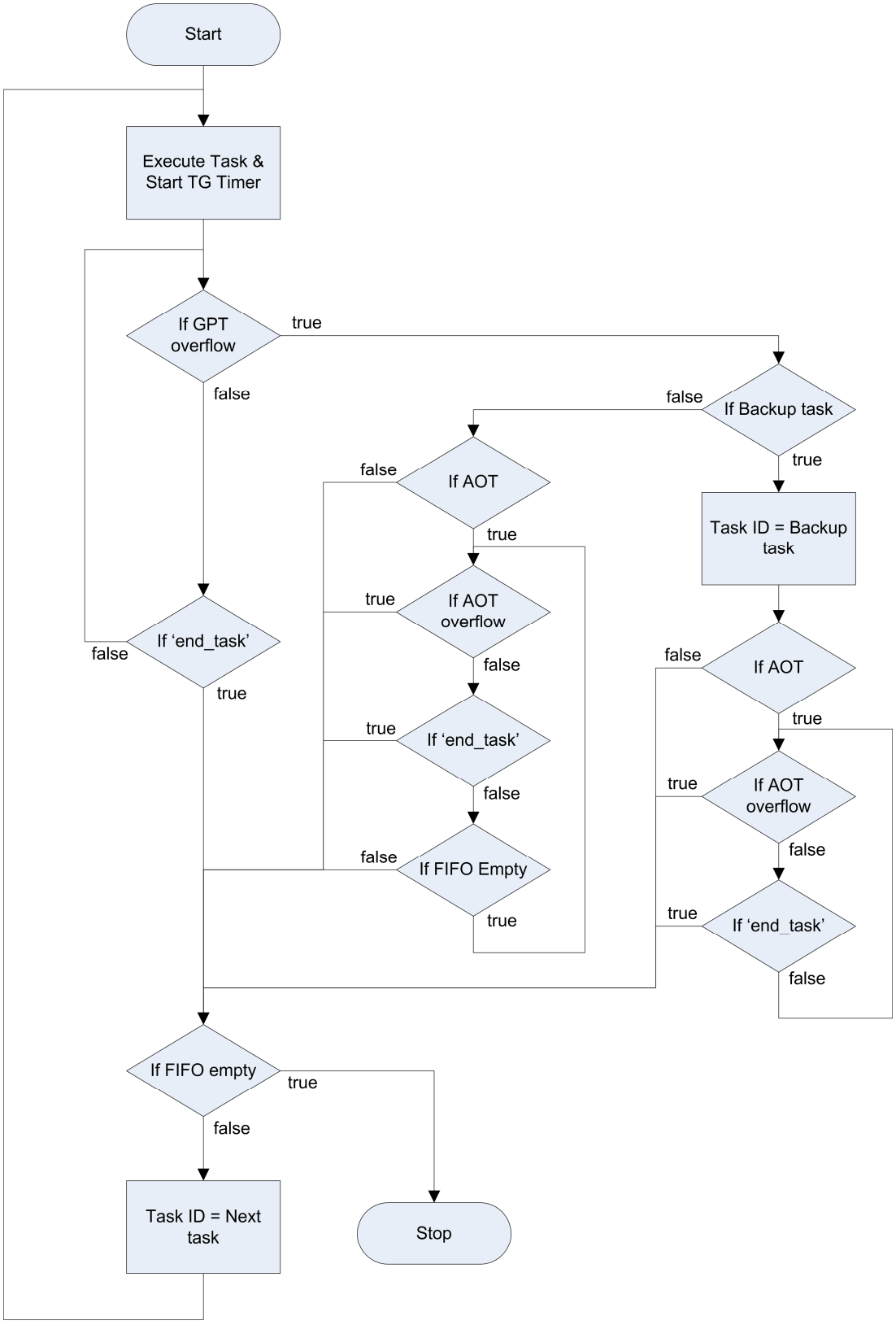


Figure 75: Flowchart of the hardware task guardian operation.

7.3 Results

The following section demonstrates the overhead that occurs when the hardware task guardian shuts down the current task or runs a backup task.

A flashing LED task was added to the hardware scheduler with its subsequent backup task along with the following values (Listing 14). The task was given a 100 CPU cycle guaranteed execution time and a 100 CPU cycle allowed overrun time, which in this case (because a backup task is provided) would be the maximum time the backup task is allowed to run before it too would be shutdown. The task was set to execute with a 1ms delay and a period of 50ms.

```
SCH_Add_Task(led_task, led_bk_task, 1, 50, 100, 100);
```

Listing 14: Hardware TG LED task parameters

Before the main LED task is run, an assembly wrapper was used to set a GPIO pin high (for measuring purposes), load the stack pointer with the base address and execute the 'endtask' instruction once the task completed.

```
led_task:
      # Set GPIO pin1 high
      lui $26, 0x3
      ori $26, $26, 0x8
      li  $27, 1
      sw  $27, 0($26)
      # Set stack pointer register
      la $29, 0x00011FF0
      # Call the LED Task
      jal LED_Update
      nop
      # End of task
      endtask
```

Listing 15: Hardware TG LED task assembly wrapper

The LED_Update function that is called by the assembly wrapper will alternate the LED pin and on every other execution it will execute a 'while 1' loop.

```
void LED_Update(void)
{
	LED_pin = LED_state;
	LED_state = ~LED_state;

	if (++overrun_state == 2)
	{
		overrun_state = 0;
		while (1);
	}
}
```

Listing 16: Hardware TG LED task

When the LED_Update task executes the while 1 loop, the hardware task guardian will shutdown the overrunning LED task and execute its backup task. The backup task also has an assembly wrapper which differs from the normal task wrapper in that it sets a different GPIO pin which is used for measurements.

```
led_bk_task:
	# Set GPIO pin2 high
	lui $26, 0x3
	ori $26, $26, 0x8
	li  $27, 2
	sw  $27, 0($26)
	# Set stack pointer register
	la $29, 0x00011FF0
	# Call the LED Backup Task
	jal LED_Backup
	nop
	# End of task
	endtask
```

Listing 17: Hardware TG LED backup task assembly wrapper

In addition to the LED tasks there is also a seven segment task. The purpose of the seven segment task is that when the LED task does not overrun a GPIO pin is set high so that a measurement can be taken to show the difference between normal execution and when the LED task overruns.

```
seg_task:
      # Set GPIO pin2 high
      lui $26, 0x3
      ori $26, $26, 0x8
      li  $27, 2
      sw  $27, 0($26)
      # Set stack pointer register
      la $29, 0x00011FF0
      # Call the SEG Task
      jal SEG_Update
      nop
      # End of task
      endtask
```

Listing 18: Hardware TG seven segment task assembly wrapper

When comparing the time between the beginning of the LED task and beginning of the next task (the Seven Segment task) or backup task, the following was observed (Figure 76).

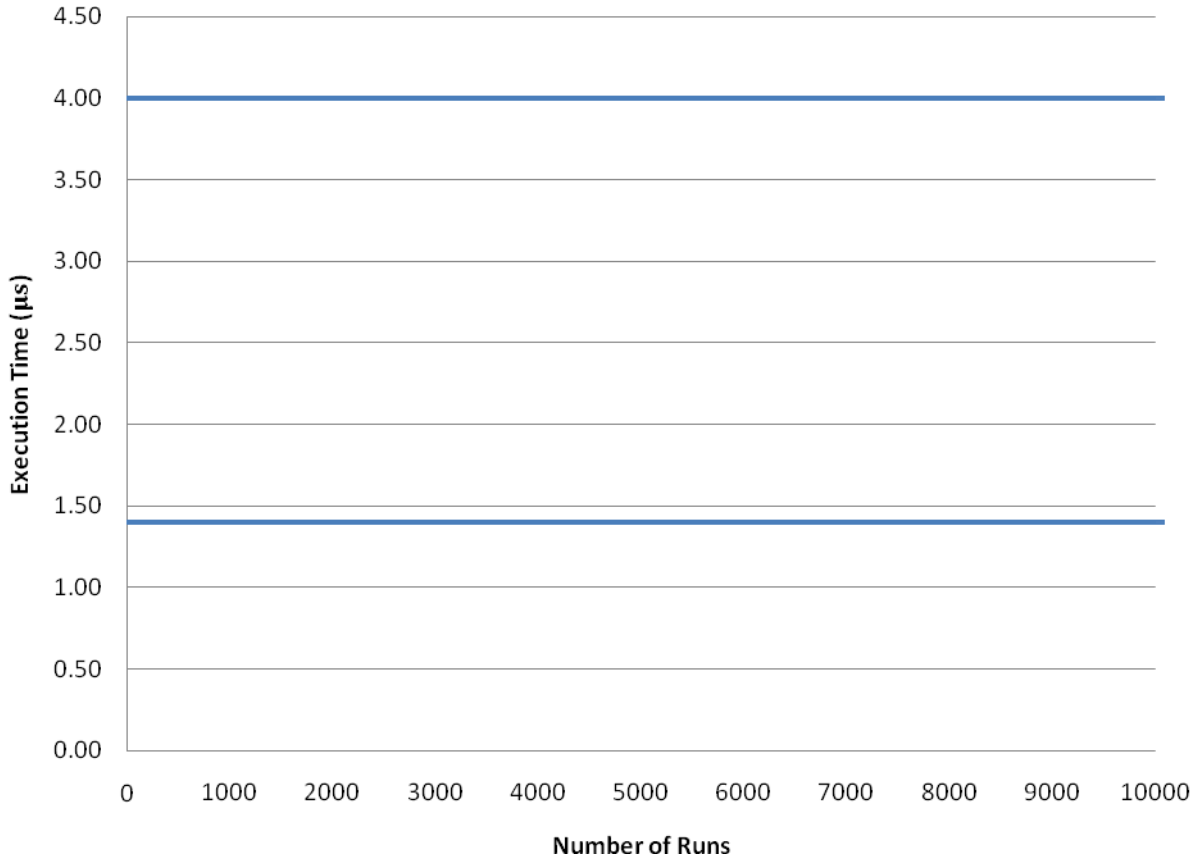


Figure 76: Execution of times of the LED task

It can be seen that when an overrun occurs there is precisely 100 CPU clock cycles between the start of the LED task and its backup task (Table 27). Whilst correct, this result is slightly offset from the actual time the LED task starts and the backup task runs. This is

due to the 4 instructions required before the GPIO pin is set in both instances. However, what is not directly apparent is the overhead required to shutdown the task and force the next one to begin.

	Execution Time (µs)	CPU Cycles
Normal	1.40	35
Overrun	4.00	100

Table 27: Detailed execution of times of the LED task

As described in Chapter 5, the first task to execute in a tick has a 3 clock cycle delay whilst its instructions are loaded into the pipeline and reach the end of the execution stage. An interrupt type signal is used to direct the program counter to the vector address of the task. This also has the effect of flushing out the contents of the first three pipeline stages. From then on, tasks are separated by a 1 clock cycle 'endtask' instruction until there are no more tasks to execute and the processor is put to sleep (Figure 77).

Figure 77: Hardware TTC scheduler overheads

In the scenario when Task 1 overruns and must be shutdown, the same interrupt signal can be used to flush out the unexecuted instructions and direct the processor to either the backup task (if one exists) or to execute the next task. Therefore, instead of a single 'endtask' instruction overhead as in normal execution, there is a 3 clock cycle delay whilst the pipeline is loaded with the next task. If the backup task completes within its allotted time, then normal execution continues. If however the backup task also overruns, then it too is shutdown and the same 3 clock cycle delay is required to force the next task to execute.

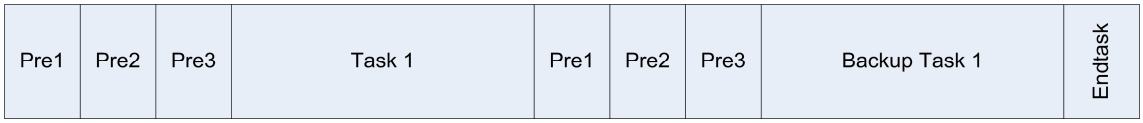


Figure 78: Hardware TG task shutdown overheads

As can be seen in Figure 78, both Task 1 and the backup task have a 3 clock cycle delay before they are executed. It is because of this identical delay that the measured overrun time in Table 27 has the same offset to give an exact value of 100 CPU cycles. This

measured value is identical to the GPT value entered in for the task parameters. To clarify how this is possible, the task guardian execution time counter starts as soon as the task is given control of the processor. Therefore in the case of Task 1, the counter is started when the processor begins to load Task 1 into the pipeline (e.g. at 'Pre1'). Similarly for the backup task the counter restarts at its 'Pre1'. However, assuming that previous task did not overrun and the current task is not the first task in the tick, as is the case for Task 2 in Figure 77, then the counter starts as soon as the first instruction is executed. This in essences means that the GPT and APT values should be set to 3 CPU cycles longer than the expected worst case times to cover the potential overhead to load each task into the pipeline.

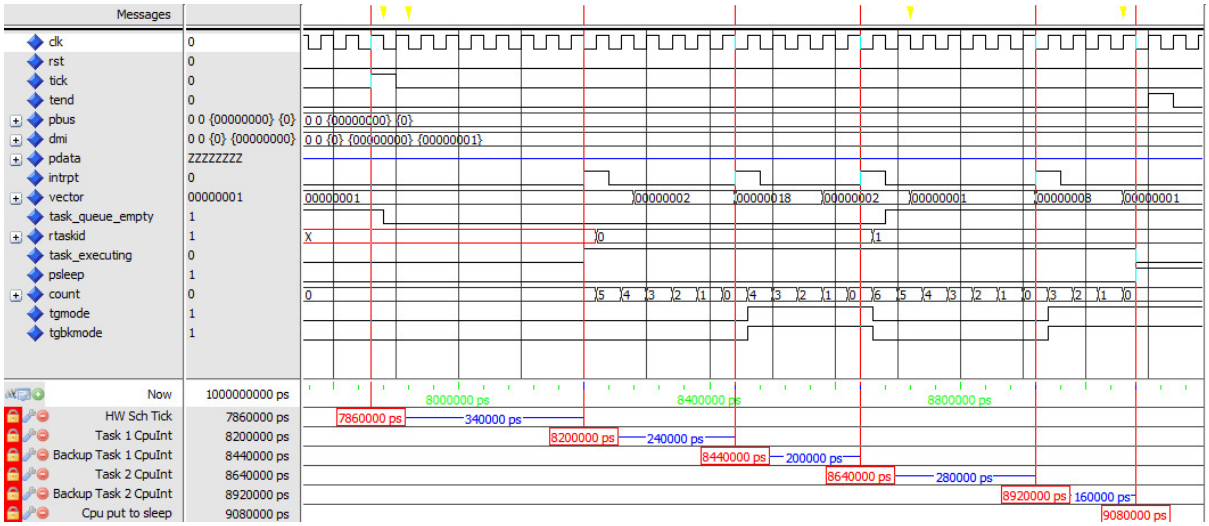


Figure 79: Modelsim simulation of the hardware task guardian unit in action[19]

To verify the task guardian component the unit was simulated in Modelsim. Figure 79 shows the scenario where two tasks and their respective backup tasks overrun. The end result is that the final backup task is shutdown and the processor is immediately sent to sleep.

7.4 Expansion of TG

The description of the task guardian so far has related mainly to a time guardian. However there are many other conditions in which it is suitable to employ the task guardian recovery

[19] A larger version of this diagram is available in Appendix C, Figure 93.

mechanism. For instance, when exceptions generated from processor detect errors such as mathematical and invalid instructions can be linked into the task guardian.

For instance it might be advantageous for a software engineer to include their own 'divide by zero' error handler as a back-up task for a specific task containing important mathematical routines. Features such as these can be provided and processed through the use of the TG unit.

One key aspect to ensuring a task is operating within its bounds is to check for invalid memory accesses and program flow errors. These could be detected through a memory guardian designed to interface with the task guardian unit in order to employ its error recovery mechanisms.

Overall, the various aspects of the task guardian mechanism allow for predictable behaviour in the event of task overruns without unnecessarily restricting processor time for task executions.

7.5 Comparisons of the hardware cores

When observing the FPGA logic utilisation it is apparent that the hardware task guardian mechanism consumes a fair bit more logic than the other cores (Figure 80). This may in part be due to the number of task variables which are synthesised into registers made out of FPGA logic. The cores containing hardware schedulers in Figure 80 were synthesised for 8 tasks, however for larger numbers of tasks it may be advantageous to store the task variables in FPGA block RAMs to conserve the logic resources.

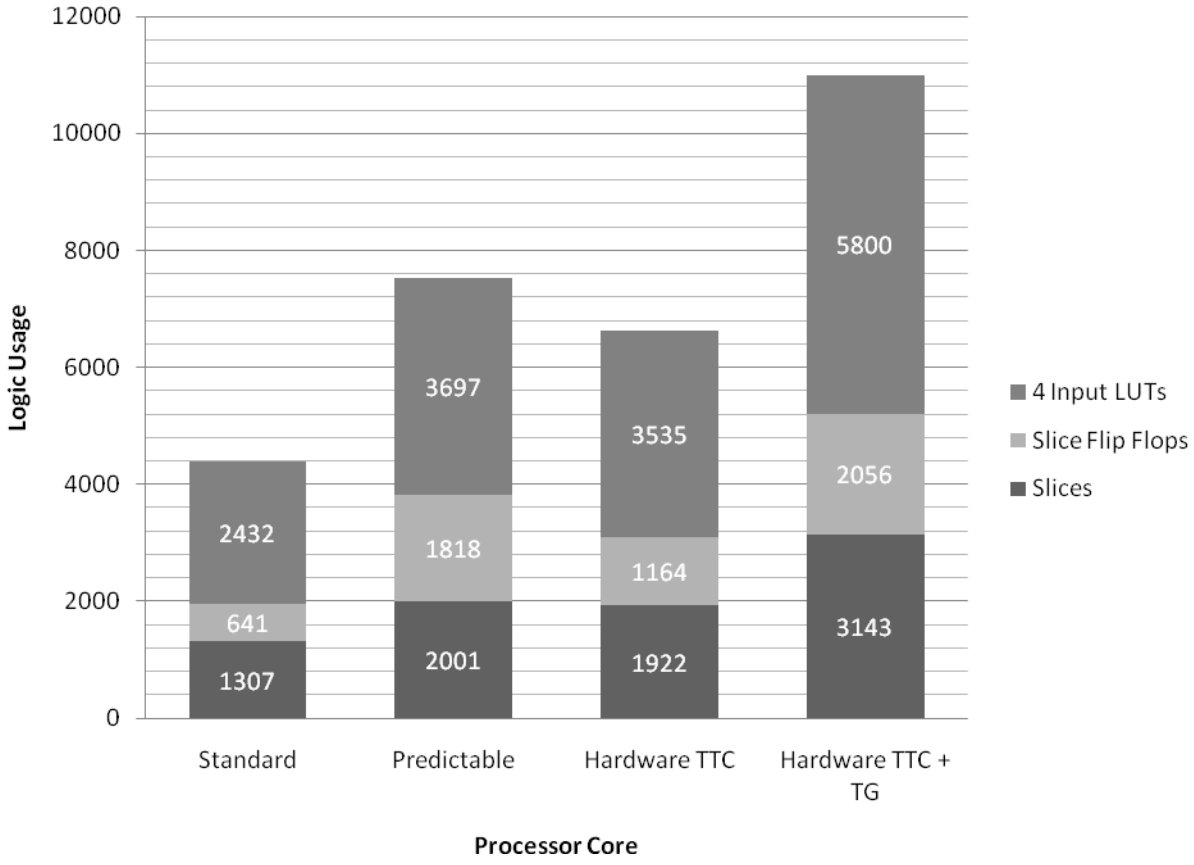


Figure 80: Xilinx Spartan 3 – 400 FPGA logic usage

Whilst the hardware logic consumption is increased quite substantially, the consequence is that the code and data sizes are dramtically reduced when using the hardware schedulers and task guardians (Figure 81).

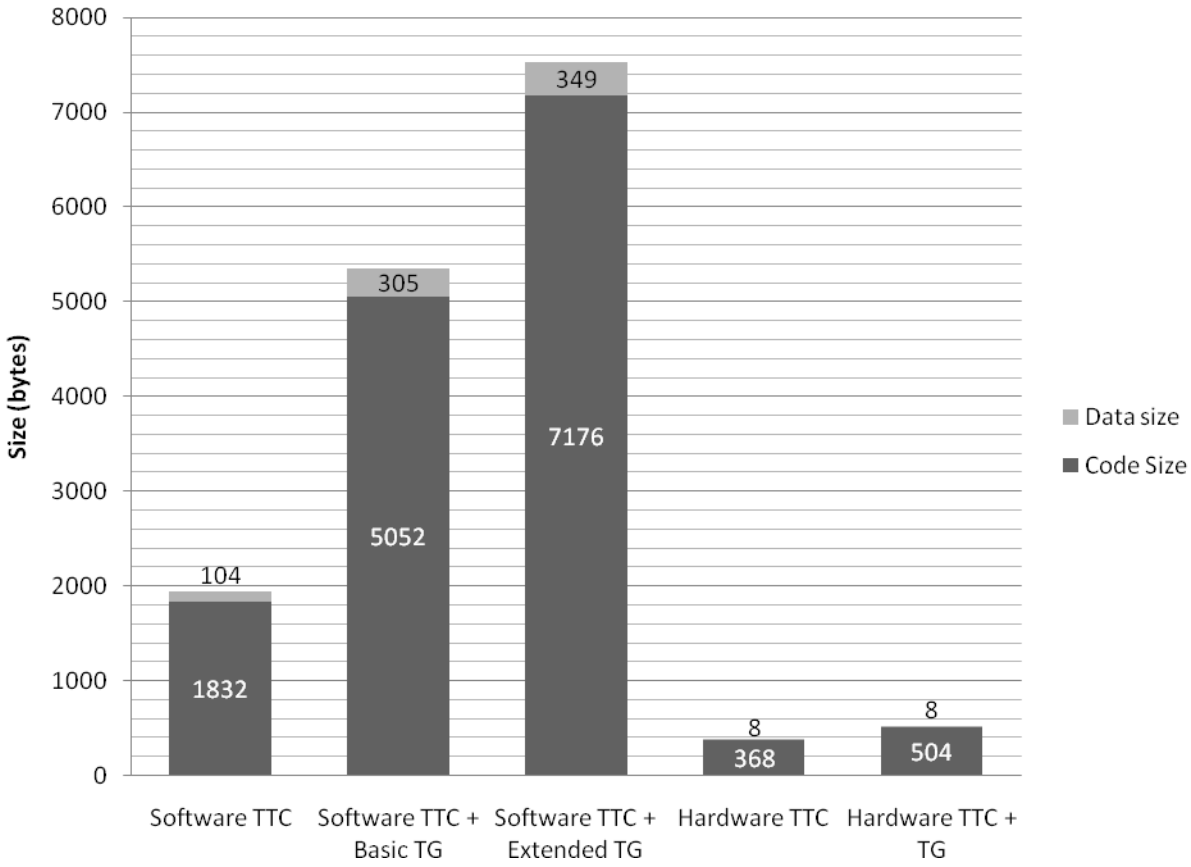


Figure 81: Code and data sizes of software and hardware systems

In addition, the hardware scheduler overheads for executing tasks are significantly reduced and the between task overheads are made static when compared to the software solutions (Figure 82).

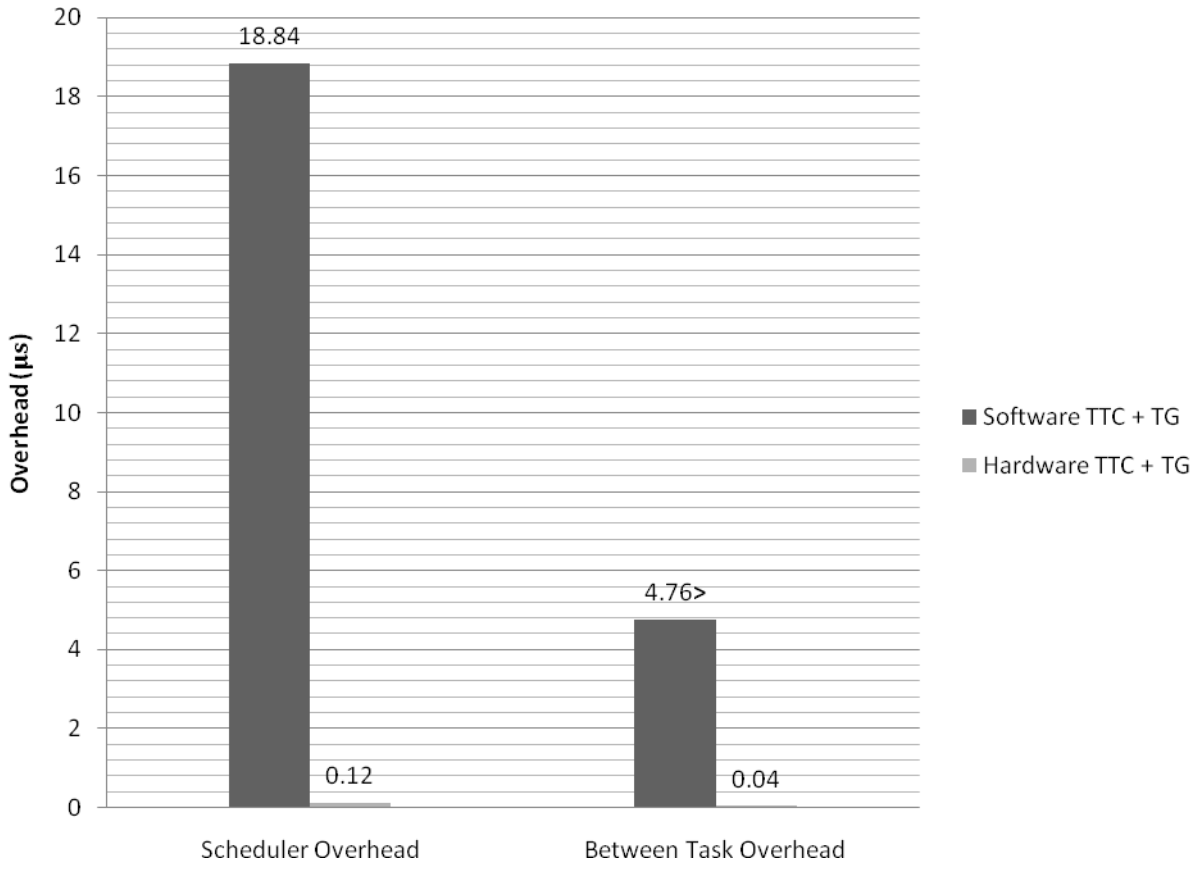


Figure 82: Extended software task guardian and hardware task guardian overheads

For the hardware task guardian the overheads due to a task overrun and executing an optional backup task are also significantly reduced (Figure 83). These values are static and not affected by code changes or optimisations. There is also a significant improvement in the responsiveness to detecting and taking action on a task overrun. By comparison, the software mechanism has to wait until just before the end of the tick interval before discovering if a task has overrun, whereas the hardware mechanism detects the error immediately because it runs in parallel to the processor.

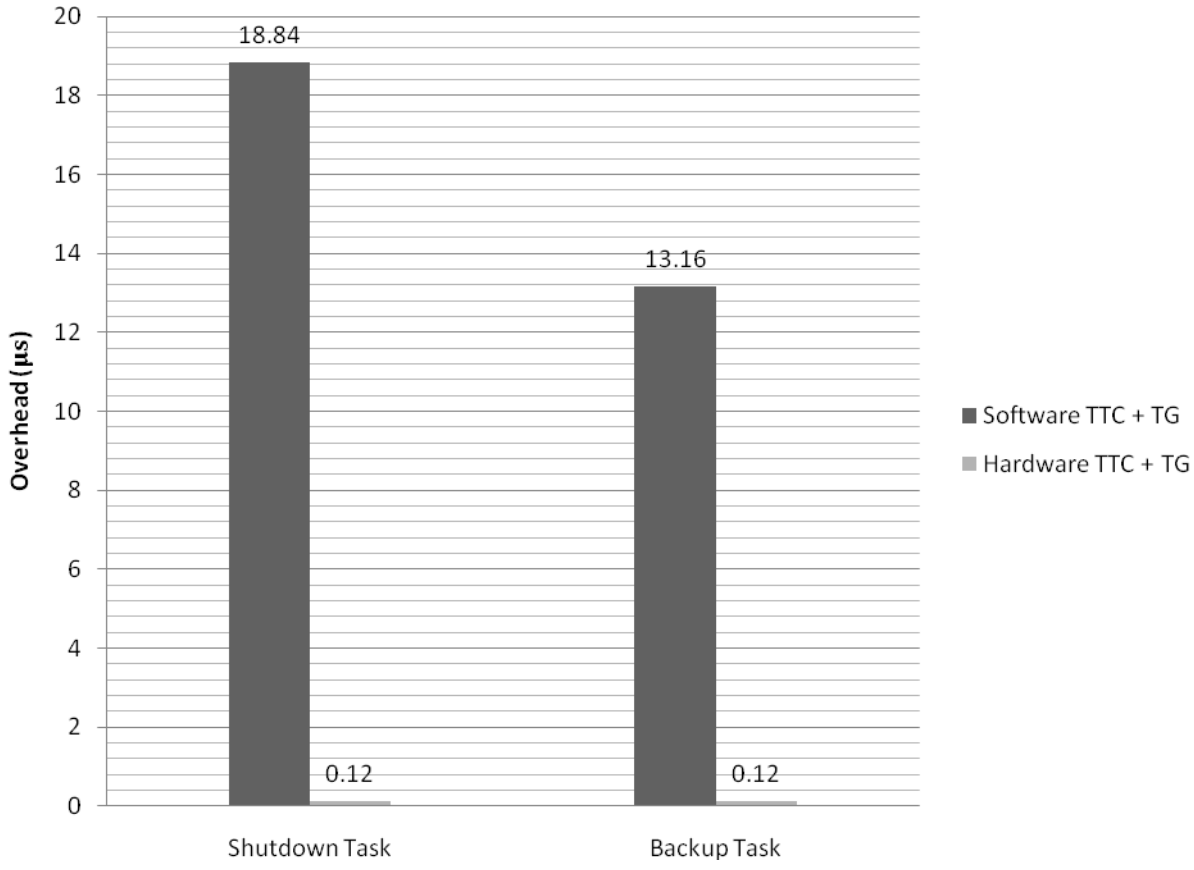


Figure 83: Software and hardware task overrun overheads

7.6 Discussion

Due to the large overhead and complexity of the software task guardian in Chapter 6, this chapter has presented a hardware solution. A significant drawback of the software solution was that task overruns would not be detected until the end of the tick interval by which time the remaining schedule may have been adversely affected. In order to prevent a domino effect and make the recovery time predictable, there were conditions when the schedule might be paused for a whole tick to allow the blocked tasks time to catch up. These issues could be avoided by the hardware task guardian which could run in parallel to the executing tasks. The hardware solution also provided a means for relaxing the WCETs entered into the task guardian by allowing tasks to overrun up to an upper bound as long as it would not interfere with any other tasks.

When comparing the hardware task guardian core which is synthesised with support for up to 8 tasks, the core was 2.5 times larger than the standard processor core and 1.7 times

larger than the hardware TTC core. This represented a significantly larger increase in silicon area. However, the overhead was dramatically reduced especially under overrun conditions where a task could be shutdown in 3 CPU cycles. In context, this overhead was 157 times smaller than the software task guardian. The code and data sizes were also much smaller as the code for the hardware task guardian was only required to implement helper functions to load the appropriate task guardian registers. When compared to the software task guardian, the code size was 14.2 times smaller and the data size was 43.6 times smaller. Most importantly, the timing characteristics of the hardware task guardian are simple and easy to understand and remain statically in line with the CPU frequency. This can be particularly useful in power conscious systems that could take advantage of dynamic frequency and voltage scaling (Phatrapornnant and Pont 2006).

It has also been discussed that the hardware TG can be expanded to operate on a variety of system errors providing the programmer flexibility to apply their own recovery code for a variety of conditions.

7.7 Conclusion

This chapter has provided an overview of the functionality of the hardware task guardian unit in which one of the key goals is to guarantee task processing time. This is principally produced by a timer unit that checks if a task exceeds its WCET. In the event that the WCET is exceeded two recovery mechanism are employed, task shutdown and backup tasks.

A key aspect that the hardware unit has over software alternatives is its very fast error detection and resolution without adding a large amount of complexity to system code and CPU overhead. The hardware task guardian complements the predictable processor and hardware scheduler by contributing to the predictability and reliability of a time triggered systems. This design is unique because of its fast error detection and predictable recovery mechanisms which can be factored into timing analysis. The outcome of this work is that it helps to make TTC systems more favourable in safety related applications where previous concerns about task overruns may lead to designers adopting alternative and less predictable systems.

Chapter 8 Discussions and Conclusions

8.1 Introduction

This chapter reflects on the novelty claims drawn from the research throughout this thesis and discusses where they are applicable. Limitations and recommendations for future work are also provided.

8.2 Summary

As introduced in Chapter 1, the key concern throughout this thesis has been temporal predictability for the reliable and successful development of embedded systems. In particular, temporal predictability helps to provide some form of guarantee that deadlines will be met. These deadlines are a crucial requirement for the correct operation of hard real-time systems, especially where safety critical applications are concerned. However, as argued in Chapter 3, one of the problems facing modern embedded systems is that new processors are exhibiting less predictable behaviour. This is due to the modifications made to the internal architecture to increase average case performance. In some cases these modifications can lead to completely unpredictable states and the consequences can be far reaching.

In physics and engineering, many details are known about the temporal properties of electrons, electronic devices, analogue circuits, digital circuits and logic. However, when it comes to the microprocessor architecture the internal state space can become very large and complex. In addition, the properties of an RTOS can further increase the temporal complexity of the system. This complexity is then present before adding the application tasks which are required to provide the desired functionality. This thesis has explored the idea that it could be possible to develop highly predictable embedded systems without impinging greatly on performance. As a result this led to a bottom up approach to address the temporal predictability problem.

8.2.1 Design of a predictable processor

The first step to the temporal predictability issue was the design and development of a predictable processor which would have comparable performance to many of today's

embedded microprocessors. Instead of considering an archaic type of architecture, an assessment was made of which modern features could be included in the design. The end result was to base the processor around a 5-stage pipelined design implementing the MIPS I instruction set (without patented instructions).

For the design to support a real-time scheduler, the processor not only had to be predictable for the sequential execution of instructions but also required the ability to handle interrupts in a predictable manner. Section 4.6 considered the potential ways interrupts could be implemented whilst conforming to the precise exceptions rule. This meant that exceptions and interrupts would occur in the correct order with regards to the sequential flow of instructions.

When the MIPS processor design was invented, the original principle was that each instruction would take only one clock cycle to complete. In practice, there are a few instructions that are either not possible or impractical to fit into the one clock cycle per instruction model. This then led to the inclusion of multi-cycle instructions in many modern RISC processors. Therefore one of the ways the work in this thesis differs from others is that the processor was designed to be predictable even when interrupting on multi-cycle instructions. To achieve this, Section 4.6.2.5 introduced a multi-pipeline design using some similar techniques as employed in multi-threaded and hyper threading processors.

Another running theme throughout this thesis has been the use of a highly predictable scheduling algorithm. In Chapter 2 it was argued that a time-triggered co-operative design is well suited for systems requiring very high temporal predictability. One of the requirements for the TTC scheduling design is that only one interrupt should be enabled per microcontroller. To enforce this rule and protect against accidental or malicious reconfiguration, a hardware mechanism was employed to allow only one interrupt at any particular time.

The outcome was a novel predictable processor design that had a fixed interrupt latency and interrupt overhead whilst supporting multi-cycle instructions and enforcing the one interrupt per microcontroller rule which is required for TTC systems.

8.2.2 Hardware approach to predictable TTC Scheduling

When using a predictable processor there still remains a problem of unpredictable scheduler loads on the processor. A software scheduler implementation can result in complex timing which may be based on the number of tasks in the system, complex control flow and compiler optimizations which can lead to different amounts of generated code. This was shown in Section 5.8 where the scheduler exhibited variable loads even when switching between different tasks.

Due to the complexity of understanding the software scheduler loads, Chapter 5 presented a design to move the scheduling mechanism from software to hardware. A large benefit of this was that the processor core no longer required the need for an interrupt system. As a result, the multi-pipelined core presented in Section 4.6.2.5 for the support of predictable interrupts was no longer necessary. This resulted in the processor core including the hardware TTC scheduler and support for up to 8 tasks being 13.5% smaller than the multi-pipelined core. However, the new processor core was still 51.2% larger than the standard unpredictable processor core.

Even though the logic consumption was larger, the design ensured that the scheduler loads were now much smaller and fixed. This design is unique because it provides very small overheads for a pipelined processor based on the TTC scheduling architecture, with just 3 CPU cycles before the execution of the first task in a tick interval and the inclusion of a single cycle instruction at the end of each task.

8.2.3 Dealing with task overruns in software

With a predictable processor and hardware scheduler, the design had predictable performance and very low overhead. However, there remained a problem with the TTC scheduling principle which contained an inherent failure mode that could have catastrophic consequences on the reliability of an embedded system. This problem related to the potential for task overruns. A task overrun could prevent other tasks from executing on time and thus meeting their respective deadlines. This situation was not ideal as it meant that any task could affect any other task in the system. In order to address this problem, Chapter 6 presented a software solution through the implementation of a task guardian. The unique task guardian implementation provided a way to shutdown an overrunning task

and execute - if requested - a backup task. The benefit of the backup task was that the user could define a specific recovery mechanism for each task.

Even though the task guardian provided a number of useful features, this came at a high price. For instance the code size was 3.9 times larger than the software scheduler and the data size was approximately 3.36 times larger. This not only represented a large increase in memory requirements but also a significant amount of complexity.

Nevertheless, whilst being large and complex, the software task guardian was designed to achieve a good level of temporal predictability. However, in order to predict some of the loads, measurements of the scheduler overhead would be required. If these measurements were not accurate then the reliability of the system could be compromised. It could be considered that because the system is large and complex, the effort required to verify if the system meets safety standards could render the software task guardian to be impractical.

8.2.4 Hardware approach to task overruns

Due to the large overhead and complexity of the software task guardian, Chapter 7 presented a hardware solution. A significant drawback of the software solution was that task overruns would not be detected until the end of the tick interval by which time the remaining schedule may have been adversely affected. These issues could be avoided by the hardware task guardian which could run in parallel to the executing tasks. The hardware solution also provided a means for relaxing the WCETs entered into the task guardian by allowing tasks to overrun to an upper bound as long as they would not interfere with any other tasks. Most importantly, the timing characteristics of the hardware task guardian are simple and easy to understand and remain statically in line with the CPU frequency. This can be particularly useful in power conscious systems that take advantage of power saving techniques such as dynamic frequency and voltage scaling (Phatrapornnant and Pont 2006). No other design has provided such features in relation to TTC scheduled systems as part of a complete processor design to maintain predictable and safe execution of tasks.

8.3 Discussions

For an embedded system to be predictable, the functional and temporal properties must be deterministic. However, predictability in a practical sense also includes the complexity and

human factors involved in constructing a system that is verifiable and will operate as it is intended to do. With the gap widening between what static analysis can achieve and the complexity due to the advancements in modern processor designs, it can therefore be advantageous for safety critical systems to make use of a predictable processor to alleviate these problems.

An inadvertent issue arising throughout this thesis is the hardware versus software argument. On the one hand, there is a predictable processor core that can support a software TTC scheduler and can be modified slightly if new functionality is required. On the other hand, there is the smaller fixed hardware TTC scheduler which will not suffer from the maintenance and verification costs associated with software.

Even with the use of reconfigurable FPGAs gaining popularity, the price point and power requirements for an FPGA can be less favourable than prefabricated chips. If the hardware TTC processor core is to be fabricated then the functionality cannot be modified and this then raises the question of the appropriate number of tasks that the hardware should support. In some circumstances, the inability to modify the functionality can also be an attractive feature as it prevents the system being unintentionally modified.

The hardware TTC core has many advantages over the software TTC core, such as better memory requirements, overheads and better silicon real-estate. On the other hand, the advantages and disadvantages between the hardware and software task guardians are more complex.

A significant balance can be seen between accepting either a large amount of memory usage versus a significant increase of silicon real-estate. Accepting the increase in logic area has the additional benefits of increased predictability, reduced overhead and responsiveness to overruns. Nevertheless, the resulting hardware task guardian core is more than double the size of the standard processor core and 46% bigger than the predictable core. In some cases this might seem a less cost effective way of achieving reliable and predictable performance. However, if predictability is the key concern then the hardware solution along with the benefits of decreased complexity and maintainability makes it more suitable for safety critical systems where cost may be a slightly less significant factor.

Another interesting issue could be power and performance factors. With the hardware cores exhibiting less overhead and thus better performance, this may - under some circumstances - translate into comparable power requirements, despite the additional logic. For instance, when used with dynamic voltage and frequency scaling, the available CPU time could be translated into power savings (Phatrapornnant and Pont 2006). However, under full load the power will be more demanding for the hardware cores.

8.4 Limitations

This thesis has demonstrated the design of a predictable embedded processor and hardware scheduling architecture. However, there are a number of limitations to the work presented here.

In Chapter 3 the issues of the processor memory architecture were described. This involved the problems of using cache, DRAM, DMA and other memory systems. As such, the processor cores used throughout this thesis adopted a true Harvard memory architecture. This memory was implemented through SRAM which is clocked at the processor core frequency. This then places the limitation that the cost of memory will increase as the frequency the processor core is increased. Another issue from utilizing a true Harvard memory architecture is that memory can be underutilized. For instance, an application requiring more code RAM cannot take advantage of any available data RAM. Therefore both the code and data memories must be adequately sized to meet all demands.

The hardware cores in this thesis have focused primarily on supporting a TTC scheduling architecture. While a number of applications can fit comfortably into a TTC schedule, there are also a few applications which require the ability to pre-empt. For instance, in a real-time Fast Fourier Transform (FFT) there may need to be a small, but very frequently called, data acquisition task to obtain samples which are then processed by the long running and less frequent called FFT task. Since the TTC schedule does not allow for pre-emption, the FFT task would have to be broken into smaller tasks so that the tick interval could occur at the rate required for the sampling task. This can be a less attractive proposition for an application programmer which could benefit from a small amount of pre-emption to make the development process easier. However, pre-emption does come with its own associated costs such as maintaining synchronisation and issues involved

around locking mechanisms. Therefore, the work presented here is limited to systems that can operate within a TTC schedule.

Although the processor core has been targeted towards safety critical applications, the designs are still applicable to soft real-time systems where the predictability may not apply to safety but can increase reliability of the system and help to protect against software errors. The implication of this is that it can help to reduce development costs by detecting bugs early and protect against problems that might result in expensive recalls from the need to change the system firmware.

8.5 Future Work

The work in this thesis is just the start of many new developments that could be added to increase the functionality and usability of the core in a predictable way. For instance, work could be undertaken in finding an appropriate predictable memory hierarchy so that larger and cheaper memory chips could be used. Adding the support for a TTH scheduling architecture would also increase the number of potential applications that could take advantage of the core.

Research work could then look into applying suitable static analysis tools based on the predictable nature of the processor in order to provide highly accurate WCETs. Programming methodologies and languages such as ADA could be supported through the hardware to ensure loops are bounded and that the numbers of control paths are limited. This could then help to tighten the bounds between BCET and WCET times.

Further enhancements to the task guardian could also include memory and IO guardians to prevent tasks from interfering with one another whilst providing a suitable inter task communication system. Function guardians could also be employed for critical functions to increase the responsiveness and traceability of problems in the system.

Work could also be undertaken in attempting to increase performance of the processor core by adding more sophisticated architectural features in a predictable way. For instance, VLIW could be used to support parallel functional units and systems on FPGAs could make use of configurable processing elements as a co-processor device.

The use of this work could also apply to FPGAs in the ways which similar hardware mechanisms could be used to schedule logic blocks and detect for problems when using reconfigurable computing.

A version of the PH core is currently being used in studies for predictable ways multi-core systems can be constructed (Athaide, Pont et al. 2008).

8.6 Conclusions

This thesis has presented a solution to the problem of predictability through four main contributions to the field of embedded systems.

The first contribution is the design and assessment of a predictable processor core for software based TTC scheduled systems.

The second contribution is a predictable hardware TTC scheduler and associated processor core to reduce the complexity of scheduling overhead.

The third contribution is a software task guardian and recovery mechanisms to deal with the problem of task overruns in TTC and TTH scheduled systems.

The fourth contribution is a hardware task guardian with associated hardware TTC scheduler unit for very high predictability even during recovery mechanisms.

If guarantees are to be made on whether a system will meet its temporal requirements, then the predictable processor design described throughout this thesis will ease the process of providing such guarantees. With the combination of a predictable processor, hardware scheduler and hardware task guardian, the only source of unpredictability that can arise in the system will be from the user tasks.

It has been recognised that the trends of modern processor designs have been leading away from predictability and towards higher performance. As a consequence, the safety of future embedded systems will be put at risk as these modern architectures are adopted. The designs in this thesis have attempted to address some of these issues and are just the beginning of a new wave of processor designs which are aimed specifically to improve the safety of future high performance hard real-time embedded systems.

References

Adee, S. (2008), "*The data: 37 Years of Moore's Law*", Spectrum, IEEE, vol. 45(5), p. 56.

Adomat, J., J. Furunäs, L. Lindh and J. Stärner (1996), "*Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems*", in The 8th Euromicro Workshop on Real-Time Systems, L'Aquila, Italy, IEEE.

Akgul, B., V. Mooney, H. Thane and P. Kuacharoen (2003), "*Hardware Support for Priority Inheritance*", in Proceedings of the IEEE Real-Time Systems Symposium (RTSS'03), pp. 246-254.

Akgul, B. E. S. and V. J. Mooney (2001), "*System-on-a-Chip Processor Synchronization Support in Hardware*", in Design Automation and Test in Europe (DATE'01), pp. 633-639.

Akgul, B. E. S. and V. J. Mooney (2002), "*The System-on-a-Chip Lock Cache*", International Journal of Design Automation for Embedded Systems, vol. 7(1-2), pp. 139-174, September 2002.

Allworth, S. T. (1981), "*An Introduction to Real-Time Software Design*": Macmillan, London., ISBN.

Anantaraman, A., K. Seth, K. Patil, E. Rotenberg and F. Mueller (2003), "*Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems*", presented at the Proceedings of the 30th annual international symposium on Computer architecture, San Diego, California.

Andrews, D., D. Niehaus and P. Ashenden (2004), "*Programming Models for Hybrid CPU/FPGA Chips*", Computer, IEEE Computer Society, vol. 37(1), pp. 118-120.

Andrews, D., D. Niehaus, R. Jidin, M. Finley, W. Peck, M. Frisbie, J. Ortiz, E. Komp and P. Ashenden (2004), "*Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link*", IEEE Micro, vol. 24(4), pp. 42-53.

Andrews, D., W. Peck, J. Agron, K. Preston, E. Komp, M. Finley and R. Sass (2005), "*hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel*", in Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation Facolta' di Ingegneria, Catania, Italy, pp. 19-22.

Anthony, F. (2003), "*Formal Specification and Verification of ARM6*", in Theorem Proving in Higher Order Logics, 2003, pp. 25-40.

ARTEMIS (2004), "*Building ARTEMIS*", Report by the High-level Group on Embedded Systems: European Communities 2004, ISBN: 92-894-8632-5.

Atanassov, P., R. Kirner and P. Puschner (2001), "*Using Real Hardware to Create an Accurate Timing Model for Execution-Time Analysis*", in Proc. IEEE Real-Time Embedded Systems Workshop, held in conjunction with RTSS 2001.

Atanassov, P. and P. Puschner (2001), "*Impact of DRAM Refresh on the Execution Time of Real-Time Tasks*", in Proc. IEEE International Workshop on Application of Reliable Computing and Communication, pp. 29-34.

Athaide, K. F., M. J. Pont and D. Ayavoo (2008), "*Shared-Clock Methodology for Time-Triggered Multi-Cores*", in The thirty-first Communicating Process Architectures Conference CPA 2008, York, IOS Press, pp. 149-162.

Audsley, N. and A. Burns (1990), "*Real-time System Scheduling*", University of York, YCS 134.

Audsley, N. C., A. Burns, R. I. Davis, K. Tindell and A. J. Wellings (1995), "*Fixed Priority Pre-emptive Scheduling: A Historical Perspective*", Real-Time Systems, vol. 8(2), pp. 173 - 198.

Audsley, N. C., A. Burns, M. F. Richardson and A. J. Wellings (1991), "*Hard Real-Time Scheduling: The Deadline Monotonic Approach*", in The 8th IEEE Workshop on Real-Time Operating Systems and Software, Atalanta, pp. 133-137.

Ayavoo, D., M. J. Pont, J. Fang, M. Short and S. Parker (2005), "*A 'Hardware-in-the Loop' testbed representing the operation of a cruise-control system in a passenger car*", in Proceedings of the Second UK Embedded Forum, Birmingham, UK, Published by University of Newcastle upon Tyne, pp. 60-90.

Baker, T. P. and A. Shaw (1988), "*The cyclic executive model and Ada*", in Proceedings of the Real-Time Systems Symposium Huntsville, AL, USA, pp. 120-129.

Bardeen, J. and W. H. Brattain (1947), "*The transistor, a semi-conductor triode*", Phys. Rev., vol. 74, pp. 230-231.

Basumallick, S. and K. Nilsen (1994), "*Cache Issues in RealTime Systems*", in ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems.

Bate, I., P. Conmy and T. Kelly (2001), "*Use of Modern Processors in Safety-Critical Applications*", Computer Journal, vol. 44(6), pp. 531-543.

Bate, I. J. (1997), "*An architecture for Distributed Real-time Systems*", University of York, Department of Computer Science, University Technology Centre (UTC), YUTC file, YUTC/TR/97.2.

Bate, I. J. (1998), "*Scheduling and Timing Analysis for Safety Critical Real-Time Systems*", Doctor of Philosophy, Department of Computer Science, University of York, York, 1998.

Bate, I. J. (2000), "*Introduction to scheduling and timing analysis*", in The Use of Ada in Real-Time System, IEE Conference Publication 00/034.

Bauer, G., H. Kopetz and W. Steiner (2003), "*The central guardian approach to enforce fault isolation in the time-triggered architecture*", in Autonomous Decentralized Systems, 2003. ISADS 2003. The Sixth International Symposium on, pp. 37-44.

Beaty, D. (1995), "*The Naked Pilot: The Human Factor in Aircraft Accidents*": The Crowood Press Ltd, ISBN: 1853104825.

Bennett, M. D. and N. C. Audsley (2001), "*Predictable and efficient virtual addressing for safety-critical real-time systems*", in Real-Time Systems, 13th Euromicro Conference on, 2001., pp. 183-190.

Berg, C., J. Engblom and R. Wilhelm (2004), "*Requirements for and Design of a Processor with Predictable Timing*", in Proc. of the Dagstuhl Perspectives Workshop on Design of Systems with Predictable Behaviour Schloss Dagstuhl, Germany, Internationales Begegnungs und Forschungszentrum fuer Informatik (IBFI).

Bernat, G., R. Davis, N. Merriam, J. Tuffen, A. Gardner, M. Bennett and D. Armstrong (2007), "*Identifying Opportunities for Worst-Case Execution Time Reduction in an Avionics System*", Ada User Journal, vol. 28(3), pp. 189-194.

Bini, E. and G. C. Buttazzo (2004), "*Schedulability Analysis of Periodic Fixed Priority Systems*", IEEE Transactions on Computers vol. 53(11), pp. 1462-1473 November 2004.

Blake, B. A. (1992), "*Assignment of independent tasks to minimize completion time*", Softw. Pract. Exper., vol. 22(9), pp. 723-734.

Boltzmann, L. and S. G. Brush (1995), "*Lectures on Gas Theory*": Dover Publications, ISBN: 0486684555.

Boolos, G. S., J. P. Burgess and R. C. Jeffrey (2007), "*Computability and Logic*": Cambridge University Press, ISBN: 9780521701464.

Boussemart, Y., M. Ouimet, S. Gorelov and I. K. Lundqvist (2006), "*Non-Intrusive System-Level Fault Tolerance for an Electronic Throttle Controller*", in To appear in: International Conference on Systems ICONS 2006.

Bowen, J. and V. Stavridou (1993), "*Safety-critical systems, formal methods and standards*", Software Engineering Journal, vol. 8(4), pp. 189-209.

Bulpin, J. R. and I. A. Pratt (2004), "*Multiprogramming performance of the Pentium 4 with Hyper-Threading*", in In Third Annual Workshop on Duplicating, Deconstruction and Debunking (at ISCA'04), pp. 53-62.

Burns, A. (1991), "*Scheduling Hard Real-Time Systems: A Review*", Software Engineering Journal, vol. 6, pp. 116-128.

Buttazzo, G. C. (2005), "*Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*", Second edition ed.: Springer, ISBN: 0-387-23137-4.

Buttazzo, G. C. (2005), "*Rate monotonic vs. EDF: judgment day*", Real-Time Syst., vol. 29(1), pp. 5-26.

Buttazzo, G. C. and M. Caccamo (1999), "*Minimizing Aperiodic Response Times in a Firm Real-Time Environment*", IEEE Trans. Softw. Eng., vol. 25(1), pp. 22-32.

Caccamo, M., G. Buttazzo and L. Sha (2002), "*Handling Execution Overruns in Hard Real-Time Control Systems*", IEEE Trans. Comput., vol. 51(7), pp. 835-849.

Campoy, A., A. Ivars and J. Mataix (2002), "*Dynamic Use Of Locking Caches In Multitask, Preemptive Real-Time Systems*", in The 15th World Congress of the International Federation of Automatic Control, Barcelona, Spain.

Campoy, A. M., A. Perles, F. Rodriguez and J. V. Busquets-Mataix (2003), "*Static use of locking caches vs. dynamic use of locking caches for real-time systems*", in Canadian Conference on Electrical and Computer Engineering, IEEE CCECE 2003., Canada, pp. 1283-1286

Campoy, M., A. Ivars and J. Busquets-Mataix (2001), "*Static use of locking caches in multitask preemptive real-time systems*", in The IEEE/IEE Real-Time Embedded Systems Workshop.

Carlow, G. (1984), "*"Architecture of the Space Shuttle Primary Avionics Software System"*", CACM, vol. 27(9), September 1984.

Ceruzzi, P. E. (2003), "*A History of Modern Computing*", 2nd Edition ed.: The MIT Press, ISBN: 978-0-262-53203-7.

Cervin, A., D. Henriksson, B. Lincoln, J. a. Eker and K.-E. Arzen (2003) "*How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime*", *IEEE Control Systems Magazine*. pp. 16-30.

Charette, R. N. (2005) "*Why Software Fails*", *IEEE Spectrum*. Available: http://www.spectrum.ieee.org/sep05/1685

Chi, C.-H. and H. Dietz (1989), "*Unified management of registers and cache using liveness and cache bypass*", SIGPLAN Not., vol. 24(7), pp. 344-353.

Colnaric, M. and W. A. Halang (1993), "*Architectural support for predictability in hard real time systems*", Control Engineering Practice, vol. 1(1), pp. 51-57, February 1993.

Colnaric, M., D. Verber and W. Halang (1995), "*Supporting high integrity and behavioural predictability of hard real-time systems*", Informatica, Special Issue on Parallel and Distributed Real-Time Systems, vol. 19(1), pp. 59-69, Feb 1995.

Cottet, F., J. Delacroix, C. Kaiser and Z. Mammeri (2002), "*Scheduling in Real-Time Systems*": Wiley, ISBN: 0-470-84766-2.

Dauben, J. W. (1990), "*Georg Cantor - His Mathematics and Philosophy of the Infinite*": Princeton University Press, ISBN: 9780691024479.

Davis, M. (2004), "*The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*": Dover Publications, Incorporated, ISBN: 0486432289.

Delvai, M., W. Huber, P. Puschner and A. Steininger (2003), "*Processor Support for Temporal Predictability -- The SPEAR Design Example*", in 15th Euromicro Conference on Real-Time Systems.

Deshmukh, A. V. (2005), "*Microcontrollers: Theory and Applications*": McGraw-Hill, ISBN: 0070585954.

Douglass, B. P. (1997), "*Real-Time UML: Developing Efficient Objects for Embedded Systems*": Addison-Wesley Longman Publishing Co., Inc., ISBN: 0201325799.

Driscoll, K., B. Hall, H. Sivencrona and P. Zumsteg (2003), "*Byzantine Fault Tolerance, from Theory to Reality*", 2003, pp. 235-248.

Edwards, S. A. and E. A. Lee (2007), "*The case for the precision timed (PRET) machine*", presented at the Proceedings of the 44th annual Design Automation Conference, San Diego, California.

Engblom, J. (2002), "*Processor Pipelines and Static Worst-Case Execution Time Analysis*", PhD thesis Dissertations from the Faculty of Science and Technology 36, Dept. of Information Technology, Uppsala University, Acta Universitatis Upsaliensis, 2002.

Engblom, J. (2003), "*Analysis of the execution time unpredictability caused by dynamic branch prediction*", in Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 152-159.

Engblom, J., A. Ermedahl and F. Stappert (2001), "*Validating a Worst-Case Execution Time Analysis Method for an Embedded Processor*", Dept. of Information Technology, Uppsala University, Technical Report 2001-030, December 2001.

Engblom, J. and B. Jonsson (2002), "*Processor pipelines and their properties for static WCET analysis*", in Proceedings of the Second International Conference on Embedded Software, London, UK, Springer-Verlag, pp. 334-348.

Engel, F., I. Kuz, S. M. Petters and S. Ruocco (2004), "*Operating Systems on SoCs: A good idea?*", in Embedded Real-Time Systems Implimentation (ERTSI 2004) Workshop, Lisbon, Portugal.

Feiler, P. H., B. Lewis and S. Vestal (2000), "*Improving predictability in embedded real-time systems*", Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

Ferdinand, C., R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing and R. Wilhelm (2001), "*Reliable and Precise WCET Determination for a Real-Life Processor*", in Embedded Software, 2001, pp. 469-485.

Fernandez-Leon, A., A. Pouponnot and S. Habinc (2002), "*ESA FPGA Task Force: Lessons Learned*", in presented at MAPLD 2002, Laurel, MD.

Fernando, J. C. (1991), "*On building systems that will fail*", Commun. ACM, vol. 34(9), pp. 72-81.

Fiddler, J., E. Stromberg and D. N. Wilner (1990), "*Software considerations for real-time RISC*", in Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference., pp. 274-277.

Flis, T. J. (1983), "*The Use of Microprocessors for Electronic Engine Control*", Industrial Electronics, IEEE Transactions on, vol. IE-30(2), pp. 75-87.

Furunäs, J. (2000), "*Benchmarking of a Real-Time System that utilises a Booster*", in International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2000), LasVegas, USA.

Furunäs, J., J. Stärner, L. Lindh and J. Adomat (1995), "*RTU94 - Real Time Unit 1994 - Reference Manual*", Mälardalen University.

Garcia, A., J. Vila, A. Crespo and S. Saez (1999), "*A Binary-Tree Architecture for Scheduling Real-Time Systems with Hard and Soft Tasks*", sbcci, vol. 00, p. 0078.

Garman, J. R. (1981), "*The "BUG" heard 'round the world: discussion of the software problem which delayed the first shuttle orbital flight*", SIGSOFT Softw. Eng. Notes, vol. 6(5), pp. 3-10.

Gendy, A. K. and M. J. Pont (2007), "*Towards a Generic "Single-Path Programming" Solution With Reduced Power Consumption*", in Proceedings of the ASME 2007 International Design Engineering Technical Conference & Computers and Information in Engineering Conference, Las Vegas, Nevada, USA.

Gendy, A. K. and M. J. Pont (2008), "*Automatically Configuring Time-Triggered Schedulers for Use With Resource-Constrained, Single-Processor Embedded Systems*", IEEE Transactions on Industrial Informatics, vol. 4(1), pp. 37-46.

George, A. D. (1990), "*An overview of RISC vs. CISC*", in System Theory, 1990., Twenty-Second Southeastern Symposium on, pp. 436-438.

Graham, R. L. (1969), "*Bounds on Multiprocessing Timing Anomalies*", SIAM Journal on Applied Mathematics, vol. 17(2), pp. 416-429.

Großschädl, J. and E. Savaş (2004), "*Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2 m)*", 2004, pp. 161-169.

Gulati, M. and N. Bagherzadeh (1996), "*Performance study of a multithreaded superscalar microprocessor*", in Proceedings of Second International Symposium on High-Performance Computer Architecture, San Jose, CA, IEEE, pp. 291-301.

Gwennap, L. (1995), "*New Algorithm Improves Branch Prediction; Better Accuracy required for Highly Super-Scalar Designs*", Microprocessor Report, vol. 9(4), pp. 1-5, March 27, 1995.

Halang, W. A. and A. D. Stoyenko (1994), "*Real Time Computing*", NATO ASI Series, Series F: Computer and Systems Sciences vol. 127: Springer-Verlag ISBN.

Hames, R. (2009, Spring Issue 2009), "*The Computer Chronicles*". Available: http://www.crews.org/curriculum/ex/compsci/articles/generations.htm

Hannibal (2004, Aug 2009), "*RISC vs. CISC: the Post-RISC Era*". Available: http://arstechnica.com/cpu/4q99/risc-cisc/rvc-1.html

Hardung, B., T. Kölzow and A. Krüger (2004), "*Reuse of software in distributed embedded automotive systems*", presented at the Proceedings of the 4th ACM international conference on Embedded software, Pisa, Italy.

Harris, D. M. and S. L. Harris (2007), "*Digital Design and Computer Architecture*": Morgan Kaufmann, ISBN: 978-0-12-370497-9.

Heath, S. (2002), "*Embedded Systems Design*": Newnes, ISBN.

Heckmann, R., M. Langenbach, S. Thesing and R. Wilhelm (2003), "*The influence of processor architecture on the design and the results of WCET tools*", Proceedings of the IEEE, vol. 91(7), pp. 1038-1054.

Hennessy, J., N. Jouppi, F. Baskett, T. Gross and J. Gill (1982), "*Hardware/software tradeoffs for increased performance*", presented at the Proceedings of the first international symposium on Architectural support for programming languages and operating systems, Palo Alto, California, United States.

Hennessy, J. and D. Patterson (2006), "*Computer Architecture - A Quantitative Approach*", 4th ed.: Morgan Kauffman, ISBN: 0-12-370490-1.

Hennessy, J. L. and T. R. Gross (1982), "*Code generation and reorganization in the presence of pipeline constraints*", presented at the Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Albuquerque, Mexico.

Hennessy, J. L., N. Jouppi, F. Baskett and J. Gill (1981), "*MIPS: a VLSI processor architecture*", Stanford University.

Hoganson, K. E. (2007), "*Concepts in Computing*": Jones and Bartlett, ISBN: 0763742953.

Hughes, Z. H. and M. J. Pont (2004), "*Design and test of a task guardian for use in TTCS embedded systems*", in Proceedings of the UK Embedded Forum 2004, (Birmingham, UK), Published by University of Newcastle upon Tyne, pp. 16-25.

Ip, N. and S. Edwards (2006), "*A Processor Extension for Cycle-Accurate Real-Time Software*", 2006, pp. 449-458.

Isaksen, U., J. Bowen and N. Nissanke (1997), "*System and Software Safety in Critical Systems*", Department of Computer Science, The University of Reading, UK.

Issacson, S. and D. Wilde (2004), "*The Task-Resource Matrix: Control for a Distributed Reconfigurable Multi-Processor Hardware RTOS*", in ERSA'04.

James, H. A., K. A. Hawick and P. D. Coddington (1999), "*Scheduling Independent Tasks on Metacomputing Systems*", in Proc. of Parallel and Distributed Computing Systems (PDCS'99).

Jamil, T. (1995), "*RISC versus CISC*", Potentials, IEEE, vol. 14(3), pp. 13-16.

Jeffay, K., D. F. Stanat and C. U. Martel (1991), "*On non-preemptive scheduling of periodic and sporadic tasks*", in the 12 th IEEE Symposium on Real-Time Systems, pp. 129-139.

Jensen, E. D., C. D. Locke and H. Tokuda (1985), "*A time driven scheduling model for real-time operating systems*", in Proceedings of the 6th IEEE Real-Time Systems Symposium RTSS 85 IEEE Computer Society, pp. 112-122.

Kain, G. and J. Heinrich (1992), "*MIPS RISC Architecture - Introducing the R4000 Technology*". New Jersey: Prentice Hall, ISBN.

Kaminski-Morrow, D. (2008), "*Airbus includes surcharge in 2008 catalogue prices*". Available: http://www.flightglobal.com/articles/2008/04/22/223184/airbus-includes-surcharge-in-2008-catalogue-prices.html

Katcher, D. I., H. Arakawa and J. K. Strosnider (1993), "*Engineering and analysis of fixed priority schedulers*", Software Engineering, IEEE Transactions on, vol. 19(9), pp. 920-934.

Kilby, J. S. (1959), "*Miniaturized Electronic Circuits, US3138743*", 1959.

Kilby, J. S. (1976), "*Invention of the integrated circuit*", IEEE Trans. Electron Devices, vol. ED-23, p. 653.

Kirk, D. B. (1989), "*SMART (strategic memory allocation for real-time) cache design*", in IEEE Real-Time Systems Symposium 1989, pp. 229-237.

Kirk, D. B. and J. K. Strosnider (1990), "*SMART (strategic memory allocation for real-time) cache design using the MIPS R3000*", in IEEE Real-Time Systems Symposium, 1990., pp. 322-330.

Kirner, R. and P. Puschner (2007), "*Time-Predictable Task Preemption for Real-Time Systems with Direct-Mapped Instruction Cache*", in The 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, IEEE Computer Society - Washington, DC, USA pp. 87-93.

Kissell, K. (2008), "*MIPS MT: A Multithreaded RISC Architecture for Embedded Real-Time Processing*", 2008, pp. 9-21.

Klingler, R. and D. Wilde (2004), "*SDCC-RTP and RTPGen: A C-to-FPGA System-on-Programmable-Chip System Generator for Multiprocessor Embedded Systems*", Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT, U.S.A.

Kohout, P., B. Ganesh and B. Jacob (2003), "*Hardware support for real-time operating systems*", in Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, Newport Beach, CA, USA, ACM.

Kopetz, H. (1991), "*Event-Triggered Versus Time-Triggered Real-Time Systems*", in The International Workshop on Operating Systems of the 90s and Beyond, Springer-Verlag pp. 87-101.

Kopetz, H. (1997), "*Real-Time Systems: Design Principles for Distributed Embedded Applications*": Kluwer Academic Publishers, ISBN: 0792398947.

Kopetz, H. (2008), "*The Complexity Challenge in Embedded System Design*", in Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, IEEE Computer Society, pp. 3-12.

Kopetz, H. and G. Grünsteidl (1994), "*TTP-A Protocol for Fault-Tolerant Real-Time Systems*", Computer, vol. 27(1), pp. 14-23.

Krishna, C. M. and K. Shin, G. (1997), "*Real-Time Systems*": MIT Press and McGraw-Hill, ISBN.

Kuacharoen, P., M. Shalan and V. Mooney (2003), "*A configurable hardware scheduler for real-time systems*", in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, pp. 96-101.

Lee, J., K. Ingström, A. Daleby, T. Klevin, V. J. M. III and L. Lindh (2003), "*A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS*", in ASP-DAC 2003 (Asia and South Pacific Design Automation Conference 2003), Kitakyushu International Conference Center, Japan, p. 6.

Leen, G. and D. Heffernan (2002), "*Expanding Automotive Electronic Systems*", Computer, vol. 35(1), pp. 88-93.

Leen, G., D. Heffernan and A. Dunne (1999), "*Digital networks in the automotive vehicle*", Computing & Control Engineering Journal, vol. 10(6), pp. 257-266, Dec 1999.

Levy, M. (2002), "*Exploring the ARM1026EJ-S Pipeline*", ARM, Cambridge, 30th April.

Lilienfeld, J. E. (1926), "*Method and Apparatus for Controlling Electric Currents, US1745175*", 1926.

Lindh, L. (1992), "*FASTHARD - A Fast Time Deterministic Hardware Based Real-Time Kernel*", in Real-Time Workshop, Athens.

Lindh, L. (1993), "*FASTHARD Prototype - A Real-Time Kernel Implemented In One Chip*", in Real-Time Workshop, Oulu, Finland.

Lindh, L., J. Furunäs and J. Stärner (1995), "*From Single to Multiprocessor Real-Time Kernels in Hardware*", First IEEE Real-Time Technology and Applications Symposium (RTAS'95), vol. 0, p. 42, January, 1995.

Lindh, L., T. Klevin and J. Furunäs (1999), "*Scalable Architecture for Real-Time Applications - SARA*", in Swedish National Real-Time Conference SNART'99, Linköping, Sweden.

Lindh, L. and F. Stanischewski (1991), "*FASTCHART - A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel*", in Real-Time Workshop, Paris, France.

Lindh, L. and F. Stanischewski (1991), "*FASTCHART - Idea and Implementation*", in International Conference on Computer Design (ICCD), Cambridge MIT, USA, IEEE Press.

Lindh, L., J. Stärner, J. Furunäs, J. Adomat and M. E. Shobaki (1998), "*Hardware Accelerator for Single and Multiprocessor Real-Time Operating Systems*", in Seventh Swedish Workshop on Computer Systems Architecture (Chalmers), Göteborg, Sweden.

Liu, C. L. and J. W. Layland (1973), "*Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment* ", J. ACM vol. 20 (1), pp. 46-61

Liu, J. W. S. (2000), "*Real-Time Systems*": Prentice Hall, ISBN.

Liu, J. W. S., K. J. Lin and S. Natarajan (1987), "*Scheduling Real-Time, Periodic Jobs Using Imprecise Results*", IEEE Real-Time Systems Symposium, pp. 252-260.

Locke, C. D. (1992), "*Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives*", Real-Time Syst., vol. 4(1), pp. 37-53.

Logue, J. C. (1998), "*From Vacuum Tubes to Very Large Scale Integration: A Personal Memoir*", IEEE Ann. Hist. Comput., vol. 20(3), pp. 55-68.

Lundqvist, K., J. Srinivasan and S. Gorelov (2005), "*Non-intrusive System Level Fault-Tolerance*", in Reliable Software Technology – Ada-Europe 2005, 2005, pp. 156-166.

Lundqvist, T. and P. Stenstr (1999), "*Timing Anomalies in Dynamically Scheduled Microprocessors*", presented at the Proceedings of the 20th IEEE Real-Time Systems Symposium.

Maaita, A. and M. J. Pont (2005), "*Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid scheduler*", in Proceedings of the Second UK Embedded Forum, Birmingham, UK, University of Newcastle upon Tyne, pp. 18-35.

Malone, D. (2008), "*Dangerous Knowledge*". UK: BBC Documentary, 11th June 08, 11.30 pm.

Marr, D. T., F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller and M. Upton (2002), "*Hyper-Threading Technology Architecture and Microarchitecture*", Intel Technology Journal.

Marti, P., R. Villa, J. M. Fuertes and G. Fohler (2001), "*On Real-Time Control Tasks Schedulability*", in European Control Conference, Porto, Portugal, pp. 2227-2232.

Mazor, S. (1995), "*The history of the microcomputer-invention and evolution*", Proceedings of the IEEE, vol. 83(12), pp. 1601-1608.

McFarling, S. (1989), "*Program optimization for instruction caches*", SIGARCH Comput. Archit. News, vol. 17(2), pp. 183-191.

Mentor Graphics (2010), "*ModelSim - Advanced Simulation and Debugging*". Available: http://model.com

Merrick, J. R., S. Wang, K. G. Shin, J. Song and W. Milam (2005), "*Priority Refinement for Dependent Tasks in Large Embedded Real-Time Software*", presented at the Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium.

Micea, M. V., V. Cretu and V. Groza (2005), "*Predictable Signal Generation with the Hard Real-Time Operating Kernel HARETICK*", in Instrumentation and Measurement Technology Conference, 2005. IMTC 2005. Proceedings of the IEEE, pp. 2097-2102.

MIPS-Technologies (2009), "*Markets Overview* ". Available: http://www.mips.com/everywhere/vertical-markets

MISRA (2004), "*Guidelines for the use of the C language in vehicle based software*", Motor Industry Software Reliability Report, October 2004.

Mohammadi, A. and S. Akl (2005), "*Scheduling Algorithms for Real-time Systems*", School of Computing, Queen's University, Kingston, Ontario.

Mooney, V. J. and D. M. Blough (2002), "*A Hardware-Software Real-Time Operating System Framework for SOCs*", IEEE Design and Test of Computers, pp. 44-51, November-December 2002.

Moore, G. E. (1965) "*Cramming more Components onto Integrated Circuits*", *Electronics*.

Mueller, F. (1995), "*Compiler support for software-based cache partitioning*", SIGPLAN Not., vol. 30(11), pp. 125-133.

Mueller, F., D. Whalley and M. Harmon (1993), "*Predicting instruction cache behavior*", in ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems.

Muller, F. (2004), "*Timing analysis: in search of multiple paradigms*", in Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, p. 126.

Nett, E., H. Streich, P. Bizzarri, A. Bondavalli and F. Tarini (1996), "*Adaptive Software Fault Tolerance Policies with Dynamic Real-Time Guarantees*", in WORDS 96, IEEE Second Int. Workshop on Object-oriented Real-time Dependable Systems, Laguna Beach, California, U.S.A., pp. 78--85.

Niehaus, D. (1994), "*Program representation and execution in real-time multiprocessor systems*", University of Massachusetts, 1994.

Niehaus, D. and D. Andrews (2003), "*Using the Multi-Threaded Computation Model as a Unifying Framework for Hardware-Software Co-Design and Implementation*", in Proceedings of the 9th International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003).

Niehaus, D., E. Nahum and J. A. Stankovic (1991), "*Predictable Real-Time Caching in the Spring System*", in In Proceedings of the 8th Workshop on RealTime Operating Systems and Software.

Nissanke, N. (1997), "*Realtime systems*": Prentice-Hall, Inc., ISBN: 0-13-651274-7.

Nolte, T. (2003), "*Reducing Pessimism and Increasing Flexibility in the Controller Area Network*", PhD thesis, Department of Computer Science and Engineering, Malardalen University, Vasteras, SWEDEN, 2003.

Noyce, R. N. (1959), "*Semiconductor device-and-lead Structure, US2981877*", 1959.

Null, L. and J. Lobur (2006), "*The Essentials of Computer Organization And Architecture*", 2nd Ed ed.: Jones & Bartlett Publishers, ISBN: 978-0763737696

Obermaisser, R. (2004), "*Event-Triggered and Time-Triggered Control Paradigms*": Springer-Verlag TELOS, ISBN: 0387230432.

Ong, R. H. L. and M. J. Pont (2002), "*The impact of instruction pointer corruption on program flow: A computational modelling study*", Microprocessors and Microsystems, vol. 25, pp. 409-419.

Ortega, R. (1994), "*Timing Predictability in Real-Time Systems*": University of Washington, 1994.

Oxford-Dictionaries (2008), "*Compact Oxford English Dictionary of Current English*", O. Corpus, O. Corpus, Third edition revised ed: Oxford University Press, 2008.

Panda, P. R., N. D. Dutt and A. Nicolau (1997), "*Exploiting off-chip memory access modes in high-level synthesis*", in Computer-Aided Design, 1997. Digest of Technical Papers., 1997 IEEE/ACM International Conference on, pp. 333-340.

Park, C. Y. and A. Shaw (1990), "*Experiments with a Program Timing Tool Based on Source level Timing Schema*", in Proceedings of the Real-Time Systems Symposium, IEEE computer society press, pp. 72-81.

Patterson, D. A. (1985), "*Reduced instruction set computers*", Commun. ACM, vol. 28(1), pp. 8-21.

Patterson, D. A. and D. R. Ditzel (1980), "*The case for the reduced instruction set computer*", SIGARCH Comput. Archit. News, vol. 8(6), pp. 25-33.

Patterson, D. A. and J. L. Hennessy (2005), "*Computer Organization and Design: The Hardware/Software Interface*", 3rd Edition ed.: Morgan Kaufmann, ISBN.

Patterson, D. A. and C. H. Séquin (1998), "*Retrospective: RISC I: a reduced instruction set computer*", presented at the 25 years of the international symposia on Computer architecture (selected papers), Barcelona, Spain.

Petters, S. M. (2002), "*Worst Case Execution Time Estimation for Advanced Processor Architectures*", Elektrotechnik und Informationstechnik, Technischen Universität at München, 2002.

Phatrapornnant, T. and M. J. Pont (2006), "*Reducing Jitter in Embedded Systems Employing a Time-Triggered Software Architecture and Dynamic Voltage Scaling*", IEEE Transactions on Computers, vol. 55(2), pp. 113-124, February 2006.

Pitter, C. and M. Schoeberl (2007), "*Time Predictable CPU and DMA Shared Memory Access*", in International Conference on Field Programmable Logic and Applications, Amsterdam, pp. 317-322.

Pont, M. J. (2001), "*Patterns for time-triggered embedded systems: building reliable applications with the 8051 family of microcontrollers*": ACM Press/Addison-Wesley Publishing Co., ISBN: 0-201-33138-1.

Pont, M. J. (2002), "*Embedded C*": Addison - Wesley, ISBN: 020179523X.

Pont, M. J. (2003), "*Supporting the development of time-triggered co-operatively scheduled (TTCS) embedded software using design patterns*", Informatica, vol. 27, pp. 81-88.

Pont, M. J. and H. L. R. Ong (2003), "*Using watchdog timers to improve the reliability of TTCS embedded systems*", in Proceedings of the First Nordic Conference on Pattern Languages of Programs, pp. 159-200.

Pont, M. J. and R. H. L. Ong (2002), "*Using watchdog timers to improve the reliability of single-processor embedded systems: Seven new patterns and a case study*", in Proceedings of the First Nordic Conference on Pattern Languages of Programs, pp. 159-200.

Puaut, I. and D. Hardy (2007), "*Predictable Paging in Real-Time Systems: A Compiler Approach*", in Proceedings of the 19th Euromicro Conference on Real-Time Systems, ECRTS '07, IEEE Computer Society, pp. 169-178.

Puente, J. A. d. l. and J. Zamorano (2003), "*Execution-time clocks and Ravenscar kernels*", presented at the Proceedings of the 12th international workshop on Real-time Ada, Viana do Castelo, Portugal.

Puschner, P. (2002), "*Is WCET Analysis a Non-Problem? - Towards New Software and Hardware Architectures*", in Proc. 2nd Euromicro International Workshop on WCET Analysis, York YO10 5DD, United Kingdom, Department of Computer Science, University of York.

Puschner, P. (2003), "*The Single-Path Approach Towards WCET-Analysable Software*", in Proc. IEEE International Conference on Industrial Technology, pp. 699-704.

Puschner, P. and A. Burns (2002), "*Writing Temporally Predictable Code*", in Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, pp. 85-91.

Ramamritham, K. and J. A. Stankovic (1994), "*Scheduling algorithms and operating systems support for real-time systems*", Proceedings of the IEEE*,* vol. 82(1), pp. 55-67.

Rapita (2008), "*RapiTime White Paper*". Available: http://www.rapitasystems.com/system/files/RapiTime-WhitePaper.pdf

Ravi, H., G. Ganesh and S. Mandayam (2003), "*Formal Verification of a Complex Pipelined Processor*", Form. Methods Syst. Des.*,* vol. 23(2), pp. 171-213.

Reeves, G. (1998), "*Re: What Really Happened on Mars?*", Risks-Forum Digest vol. 19(58). Available: http://catless.ncl.ac.uk/Risks/19.54.html#subj6

Reineke, J., B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger and B. Becker (2006), "*A Definition and Classification of Timing Anomalies*", in Proc. 6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis.

Robert, F. C., I. K. Shing, R. D. David and J. K. Edmund (1991), "*An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks*", SIGARCH Comput. Archit. News*,* vol. 19(2), pp. 290-302.

Ross, I. M. (1998), "*The invention of the transistor*", Proceedings of the IEEE*,* vol. 86(1), pp. 7-28.

SAE (1993), "*Class C Application Requirement Considerations, SAE Recommended Practice J2056/1*", SAE, June 1993.

SAE (1994), "*Survey of Known Protocols, SAE Information Report J2056/2*", SAE, April 1993.

Saez, S., J. Vila, A. Crespo and A. Garcia (1999), "*A hardware scheduler for complex real-time systems*", in Industrial Electronics, 1999. ISIE '99. Proceedings of the IEEE International Symposium on Industrial Electronics, Bled, IEEE, pp. 43-48.

Schroeder, H. R. (2006), "*The Man Behind the Microchip: Robert Noyce and the Invention of Silicon Valley [Book Review]*", Technology and Society Magazine, IEEE, vol. 25(3), pp. 50-51.

Sebek, F. (2001), "*Measuring Cache Related Pre-emption Delay on a Multiprocessor Real-Time System*", in IEEE Workshop on Real-Time Embedded Systems (RTES'01), London.

Shalan, M. and V. J. Mooney (2000), "*A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip*", in International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'00), pp. 180-186.

Shiu, P., Y. Tan and V. J. Mooney (2001), "*A Novel Parallel Deadlock Detection Algorithm and Architecture*", in 9th International Workshop on Hardware/Software Co-Design (CODES'01), pp. 30-36.

Shockley, W. (1950), "*Semiconductor Amplifier, US2502488*", 2502488, 1950.

Shockley, W. (1953), "*Bistable circuits, including transistors, US2655609*", 1953.

Shockley, W. (1957), "*Forming semiconductive devices by ionic bombardment, US2787564*", 1957.

Short, M. and M. J. Pont (2007), "*Fault-Tolerant Time-Triggered Communication Using CAN*", IEEE Transactions on Industrial Informatics, vol. 3(2), pp. 131-142.

Simonson, J. and J. H. Patel (1995), "*Use of preferred preemption points in cache-based real-time systems*", in Computer Performance and Dependability Symposium, 1995. Proceedings., International, pp. 316-325.

Smullyan, R. M. (1992), "*Gödel's Incompleteness Theorems*": Oxford University Press, ISBN: 0195046722

Souyris, J., E. L. Pavec, G. Himbert, V. Jégu, G. Borios and R. Heckmann (2005), "*Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation*", in Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis.

Spuri, M., G. Buttazzo and F. Sensini (1995), "*Robust Aperiodic Scheduling under Dynamic Priority Systems*", in 16 th IEEE Real-Time Systems Symposium, Pisa, Italy, pp. 210-219.

Stallings, W. (1988), "*Reduced instruction set computer architecture*", Proceedings of the IEEE, vol. 76(1), pp. 38-55.

Stankovic, J. A. (1988), "*Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems*", Computer, vol. 21(10), pp. 10-19.

Stankovic, J. A., D. Niehaus and K. Ramamritham (1991), "*SpringNet: A Scalable Architecture For High Performance, Predictable, and Distributed Real-Time Computing*", University of Massachusetts, Technical Report.

Stankovic, J. A., M. Spuri, M. Di Natale and G. C. Buttazzo (1995), "*Implications of classical scheduling results for real-time systems*", Computer, vol. 28(6), pp. 16-25.

Stärner, J. (1998), "*Controlling cache behavior to improve predictability in real-time systems*", in 10th Euromicro Workshop on real-time systems.

Stärner, J. (1998), "*Increasing Predictability of Real-Time Operating Systems*", in 7th Swedish Workshop on Computer System Architecture.

Stewart, D. B. (2001), "*Twenty-Five Most Common Mistakes with Real-Time Software Development*", in International Conference on Embedded Systems, San Francisco, USA.

Sun, D., D. Blough and V. J. Mooney (2002), "*Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications*", Georgia Institute of Technology, April 2002.

Thiele, L. and R. Wilhelm (2004), "*Design for Time-Predictability*", in Design of Systems with Predictable Behaviour, Dagstuhl, Germany, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, pp. 157-177.

Tia, T., J. W. S. Liu and M. Shankar (1996), "*Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems*", Real-Time Systems, vol. 10(1), pp. 23-43, January 1996.

Tiggeler, H., T. Vladimirova and D. Zheng (2000), "*A System-on-a-Chip for Small Satellite Data Processing and Control*", in Proceedings of Military and Aerospace Applications of Programmable Devices and Technologies International Conference (MAPLD'2000), Laurel, Maryland US.

Tindell, K., H. Kopetz, F. Wolf and R. Ernst (2003), "*Safe Automotive Software Development*", presented at the Proceedings of the conference on Design, Automation and Test in Europe - Volume 1.

Turing, A. M. (1939), "*Systems of logic based on ordinals*", Proceedings of the London Mathematical Society. Second Series, vol. 45, pp. 161-228.

Turley, J. (1999, 5th May 1999), "*Embedded Processors by the Numbers*", Embedded Systems Programming vol. 12(5). Available: http://www.embedded.com/1999/9905/9905turley.htm

Turley, J. (2003), "*Motoring with microprocessors*", Embedded Systems Design Available: http://www.embedded.com/columns/significantbits/13000166?_requestid=439910

Uhrig, S., S. Maier and T. Ungerer (2005), "*Toward a processor core for real-time capable autonomic systems*", in Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on, pp. 19-22.

Vera, X., B. Lisper and J. Xue (2003), "*Data cache locking for higher program predictability*", SIGMETRICS Perform. Eval. Rev., vol. 31(1), pp. 272-282.

Vera, X., B. Lisper and J. Xue (2007), "*Data cache locking for tight timing calculations*", ACM Trans. Embed. Comput. Syst., vol. 7(1), pp. 1-38.

Vidler, P. J. and M. J. Pont (2006), "*Computer Assisted Source-Code Parallelisation*", in Computational Science and Its Applications - ICCSA 2006, Glasgow, Springer, pp. 22-31.

Vladimirova, T. and X. Wu (2006), "*On-Board Partial Run-Time Reconfiguration for Pico-Satellite Constellations*", in Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on, pp. 262-269.

W. Peck, J. Agron, D. Andrews, M. Finley and E. Komp (2004), "*Hardware/Software Co-Design of Operating System Services for Thread Management and Scheduling*", in In Proceedings of the 25th IEEE International Real-Time Systems Symposium, Works In Progress Session (RTSS, WIP 2004), Lisbon, Portugal.

Weik, M. H. (1961), "*A Third Survey of Domestic Electronic Digital Computing Systems*", Ballistic Research Laboratories, Report No. 1115, Aberdeen Proving Ground, Maryland.

Whitham, J. and N. Audsley (2006), "*MCGREP--A Predictable Architecture for Embedded Real-Time Systems*", presented at the Proceedings of the 27th IEEE International Real-Time Systems Symposium.

Wilhelm, R. (2004), "*Formal Analysis of Processor Timing Models*", in Model Checking Software, 2004, pp. 1-4.

Wilhelm, R., J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat and P. Stenstr (2008), "*The worst-case execution-time problem - overview of methods and survey of tools*", Trans. on Embedded Computing Sys., vol. 7(3), pp. 1-53.

Wilkes, M. V. (1969), "*The Growth of Interest in Microprogramming: A Literature Survey*", ACM Comput. Surv., vol. 1(3), pp. 139-145.

Wolf, W. (2008), "*Computers as Components*", Principles of Embedded Computing System Design Second Edition ed.: Morgan Kaufmann, ISBN: 978-0-12-374397-8.

Wolfgang, H. (2004), "*Simplicity Considered Fundamental to Design for Predictability*", in Design of Systems with Predictable Behaviour, Dagstuhl, Germany, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

Wu, X. and T. Vladimirova (2006), "*A Self-reconfigurable System-on-Chip Architecture for Satellite On-Board Computer Maintenance*", in Proc. of the Asia-Pacific Computer Systems Architecture Conference 2006, pp. 552-558.

Xu, J. and D. L. Parnas (1993), "*On Satisfying Timing Constraints in Hard-Real-Time Systems*", IEEE Trans. Softw. Eng., vol. 19(1), pp. 70-84.

Young, M. and D. Wilde "*A Customizable Hardware/Software Real Time Operating System for a System on a Programmable Chip*", School of Electrical and Computer Engineering, Brigham Young University, Provo, UT, U.S.A.

Zheng, D., T. Vladimirova and H. Tiggeler (2001), "*Reconfigurable Single-Chip On-Board Computer for a Small Satellite*", in 52nd International Astronautical Congress, Toulouse, France.

Appendix A: Instruction Set of PH Processor

Instructions taken from MIPS I ISA (Kain and Heinrich 1992).

Non R-type instructions

Mnemonic	Name	Opcode
LW	Load Word	100011
SW	Store Word	101011
LB	Load Byte	100000
LBU	Load Byte Unsigned	100100
LH	Load Halfword	100001
LHU	Load Halfword Unsigned	100101
SB	Store Byte	101000
SH	Store Halfword	101001
BEQ	Branch On Equal	000100
BNE	Branch On Not Equal	000101
BLEZ	Branch Less Than and Equal Zero	000110
BGTZ	Branch Greater Than Zero	000111
ADDI	Add Immediate	001000
ADDIU	Add Immediate Unsigned	001001
ANDI	And Immediate	001100
J	Jump	000010
JAL	Jump And Link	000011
LUI	Load Upper Immediate	001111
ORI	Or Immediate	001101
XORI	Exclusive Or Immediate	001110
SLTI	Set Less Than Immediate	001010
SLTIU	Set Less Than Immediate Unsigned	001011

Mnemonic	Name	Opcode	RT
BLTZ	Branch On Less Than Zero	000001	00000
BLTZAL	Branch On Less Than Zero and Link	000001	10000
BGEZ	Branch On Greater Than and Equal Zero	000001	00001
BGEZAL	Branch On Greater Than and Equal Zero and Link	000001	10001

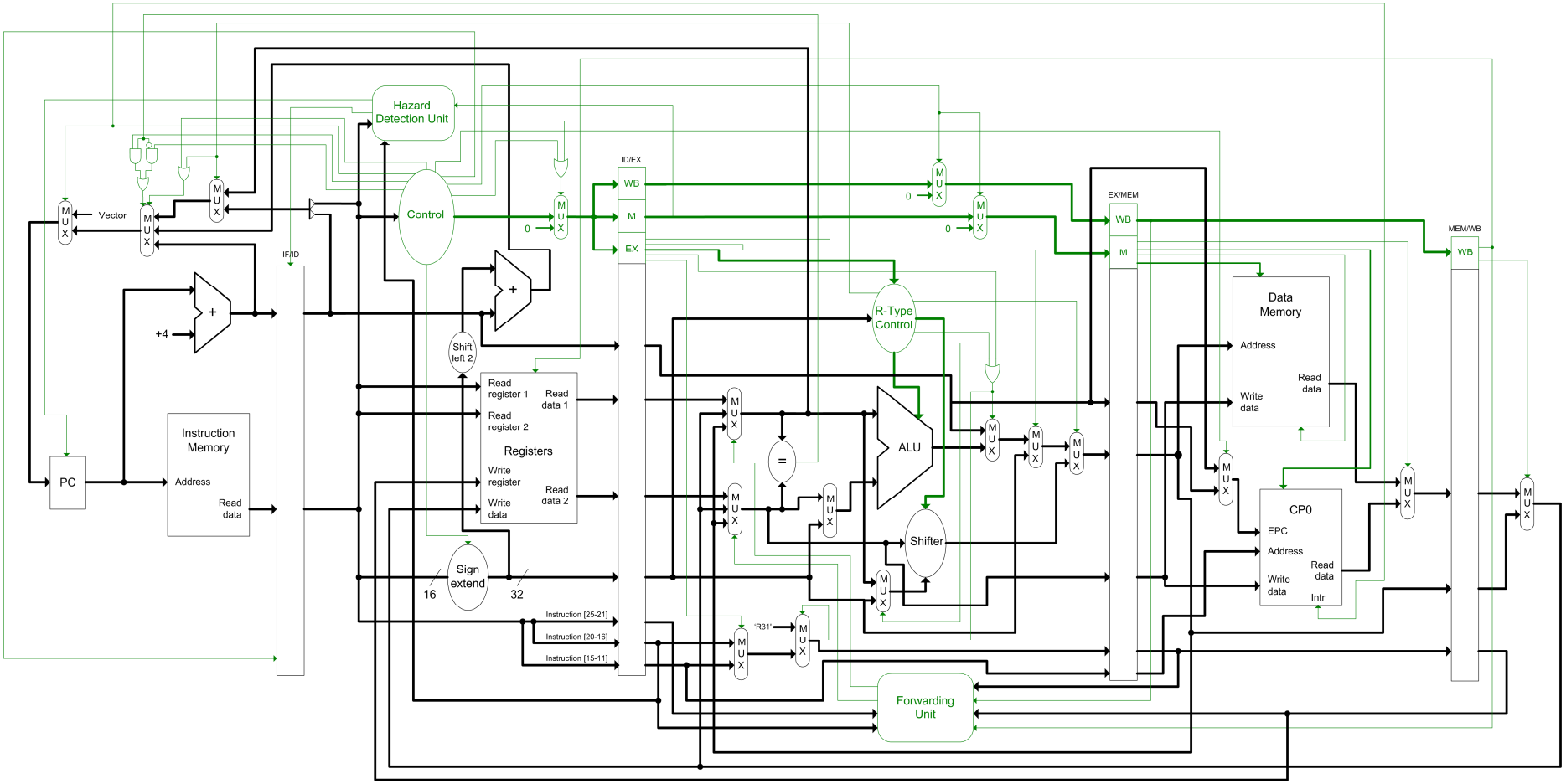
R TYPE

Mnemonic	Name	Funct
JR	Jump Register	001000
JALR	Jump And Link Register	001001
ADD	Add	100000
ADDU	Add Unsigned	100001
SUB	Subtract	100010
SUBU	Subtract Unsigned	100011
AND	And	100100
OR	Or	100101
SLT	Set Less Than	101010
SLTU	Set Less Than Unsigned	101011
NOR	Nor	100111
XOR	Exclusive Or	100110
SLL	Shift Left Logical	000000
SLLV	Shift Left Logical Variable	000100
SRA	Shift Right Arithmetic	000011
SRAV	Shift Right Arithmetic Variable	000111
SRL	Shift Right Logical	000010
SRLV	Shift Right Logical Variable	000110
MULT	Multiply	011000
MULTU	Multiply Unsigned	011001
DIV	Divide	011010
DIVU	Divide Unsigned	011011
BREAK	Breakpoint	001101

Coprocessor instructions

Mnemonic	Name	Opcode	MT/MF
MTC0	Move To System Control Coprocessor	010000	00100
MFC0	Move From System Control Coprocessor	010000	00000

Appendix B: PH Processor control and data paths



Appendix C: Additional Figures

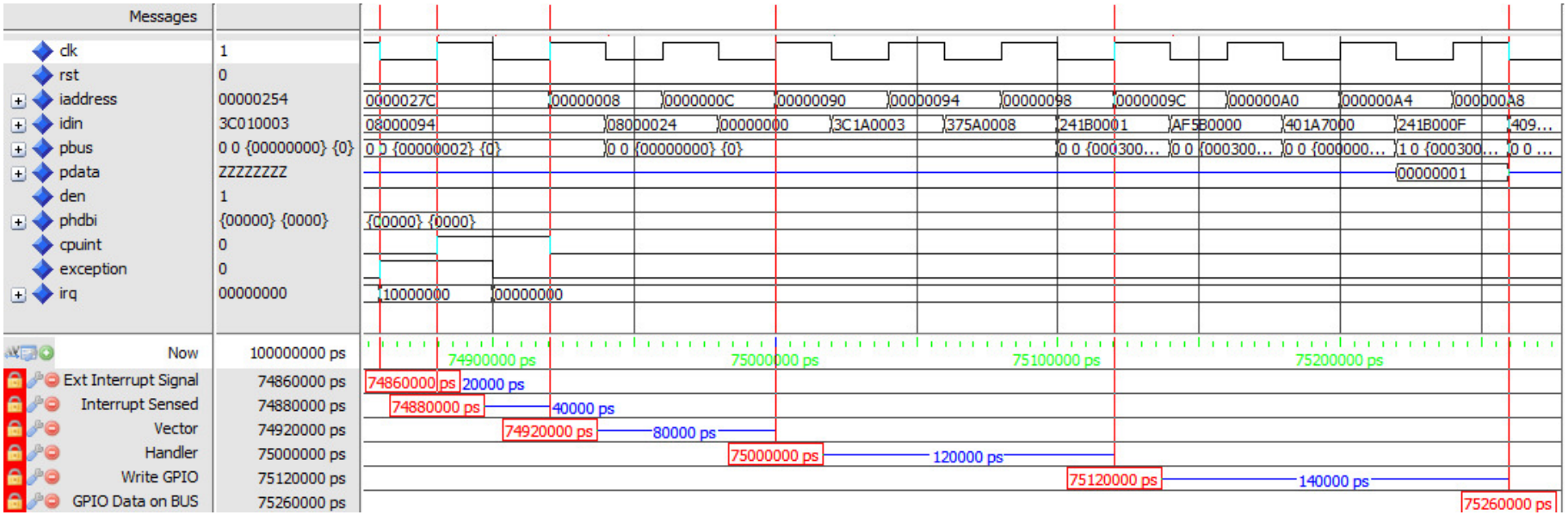


Figure 84: PH Core interrupt latency simulation

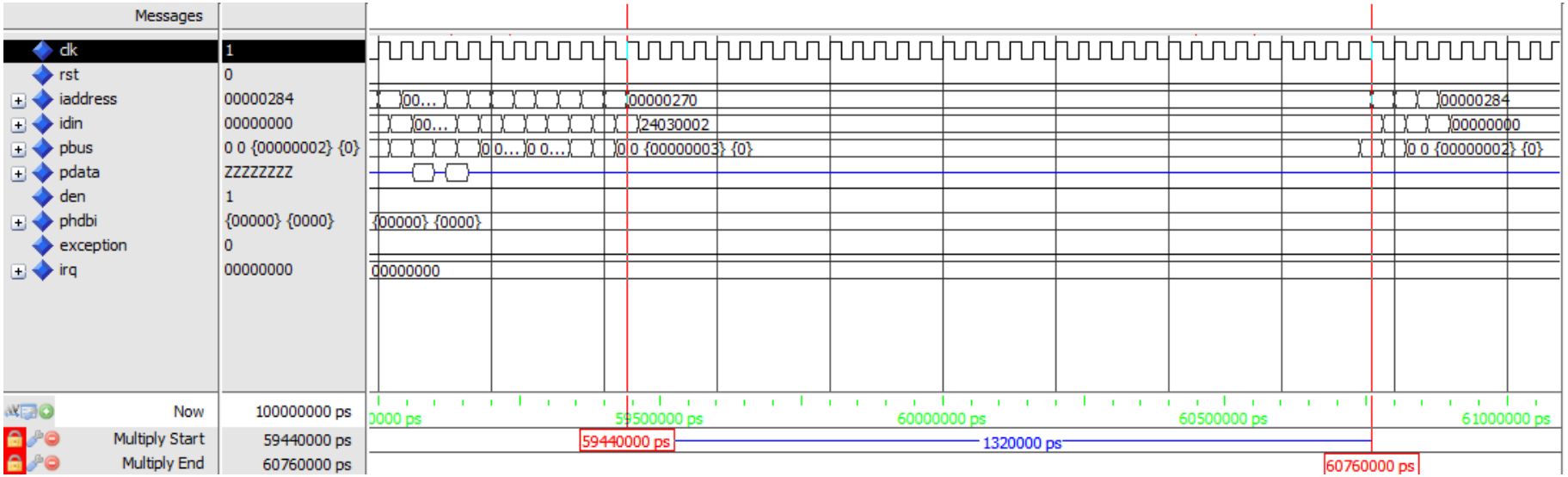


Figure 85: Multiply instruction (33 CPU clock cycles)

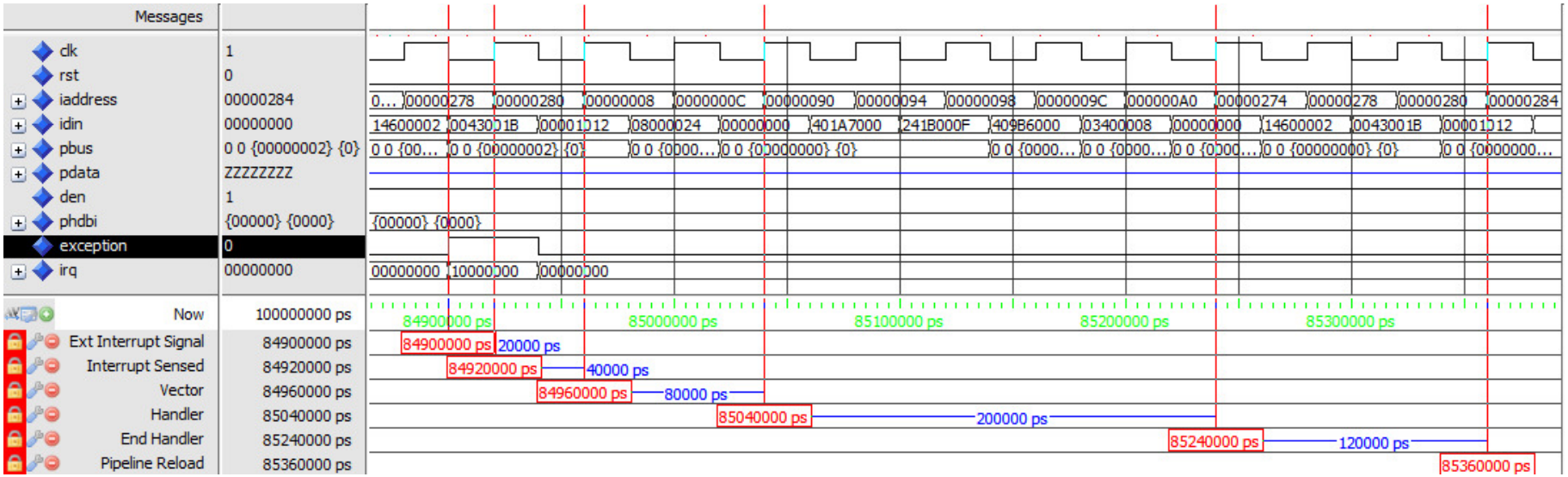


Figure 86: Interrupt Overhead (10 CPU clock cycles)

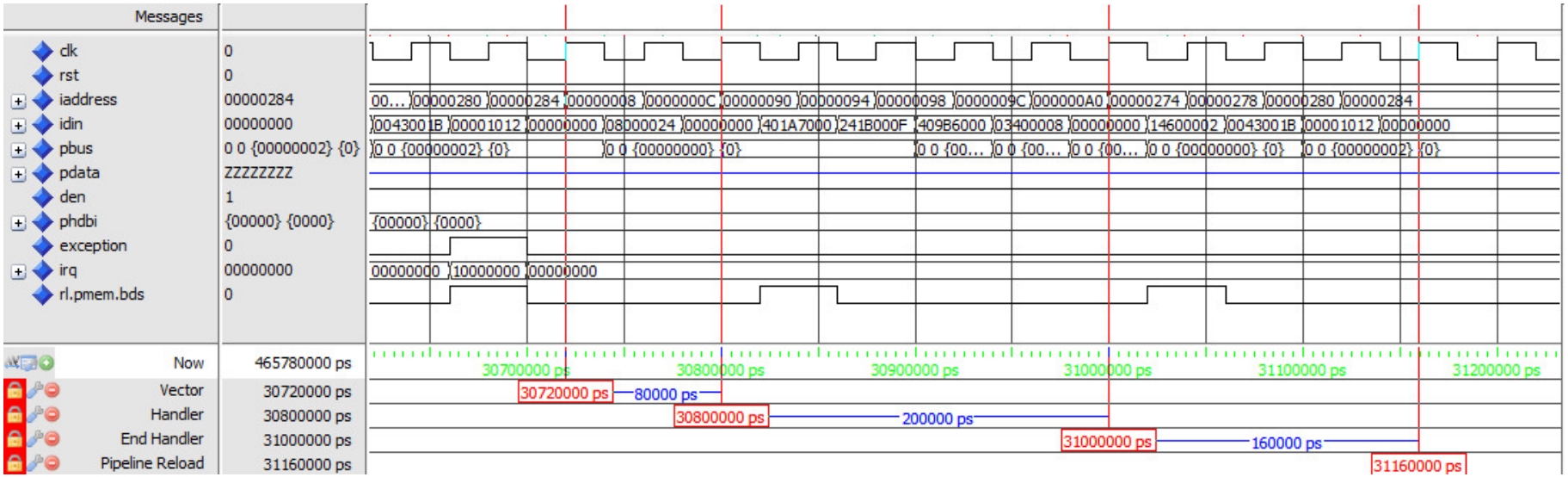


Figure 87: Interrupt on a Branch Delay Slot instruction (11 CPU clock cycles)

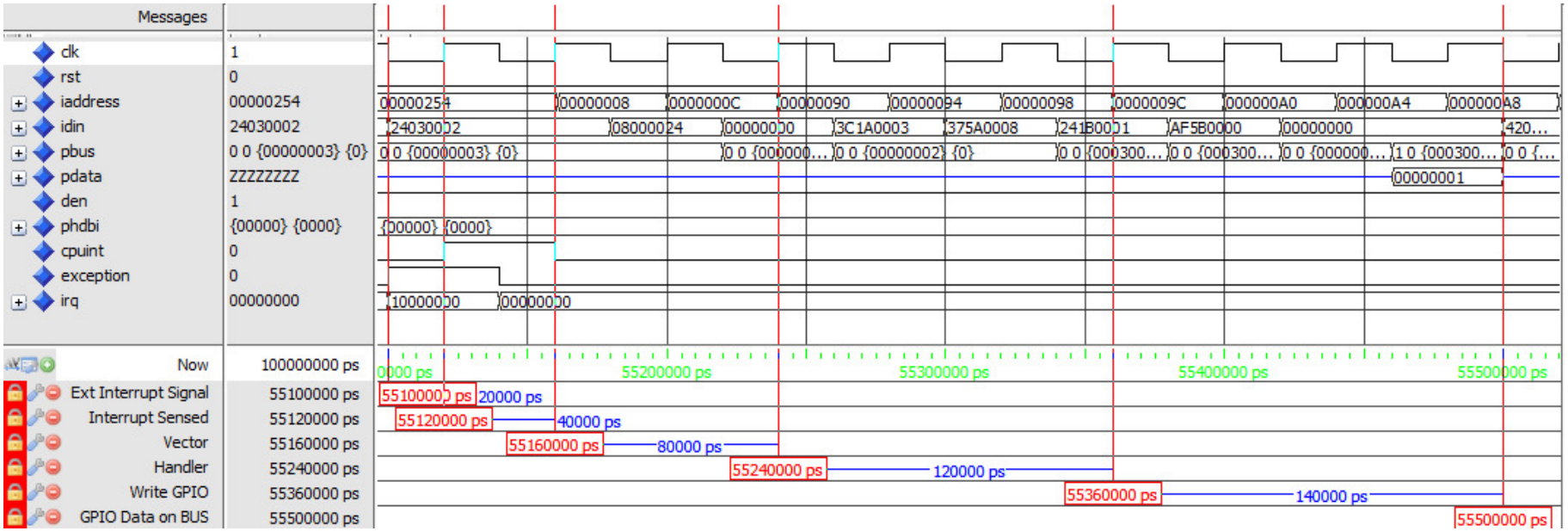


Figure 88: PH-Predictable core interrupt latency simulation

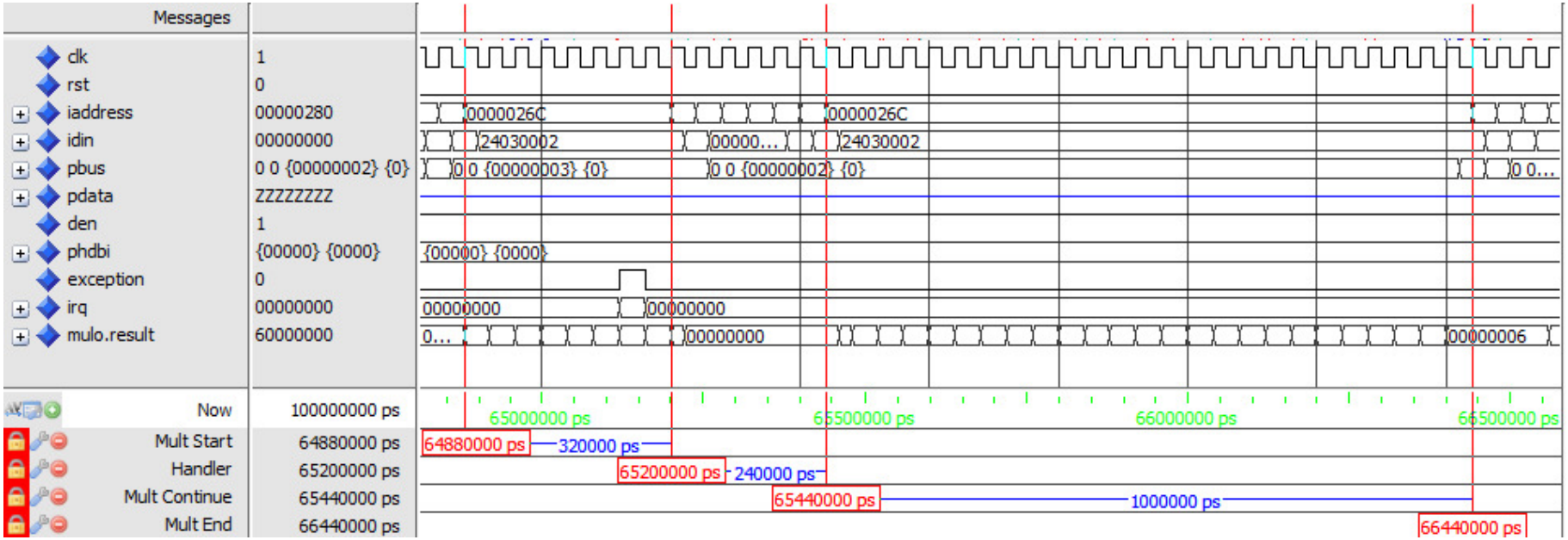


Figure 89: Multiply instruction paused as the interrupt handler is executed

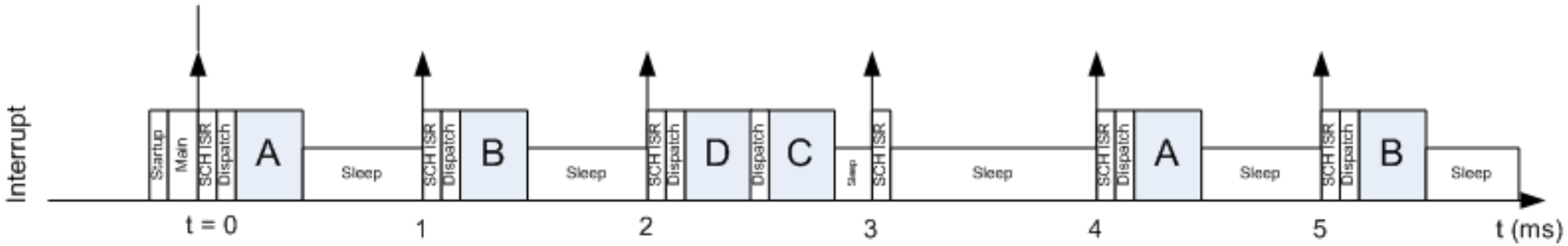


Figure 90: Detailed Time-Triggered Co-operative Schedule

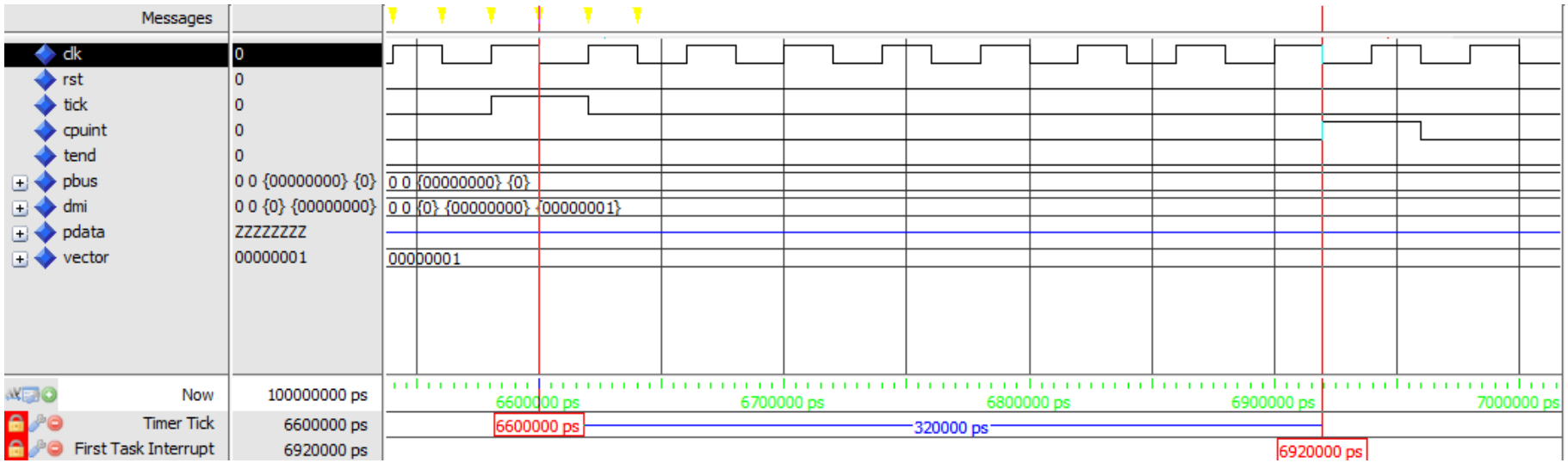


Figure 91: Hardware scheduler offset delay

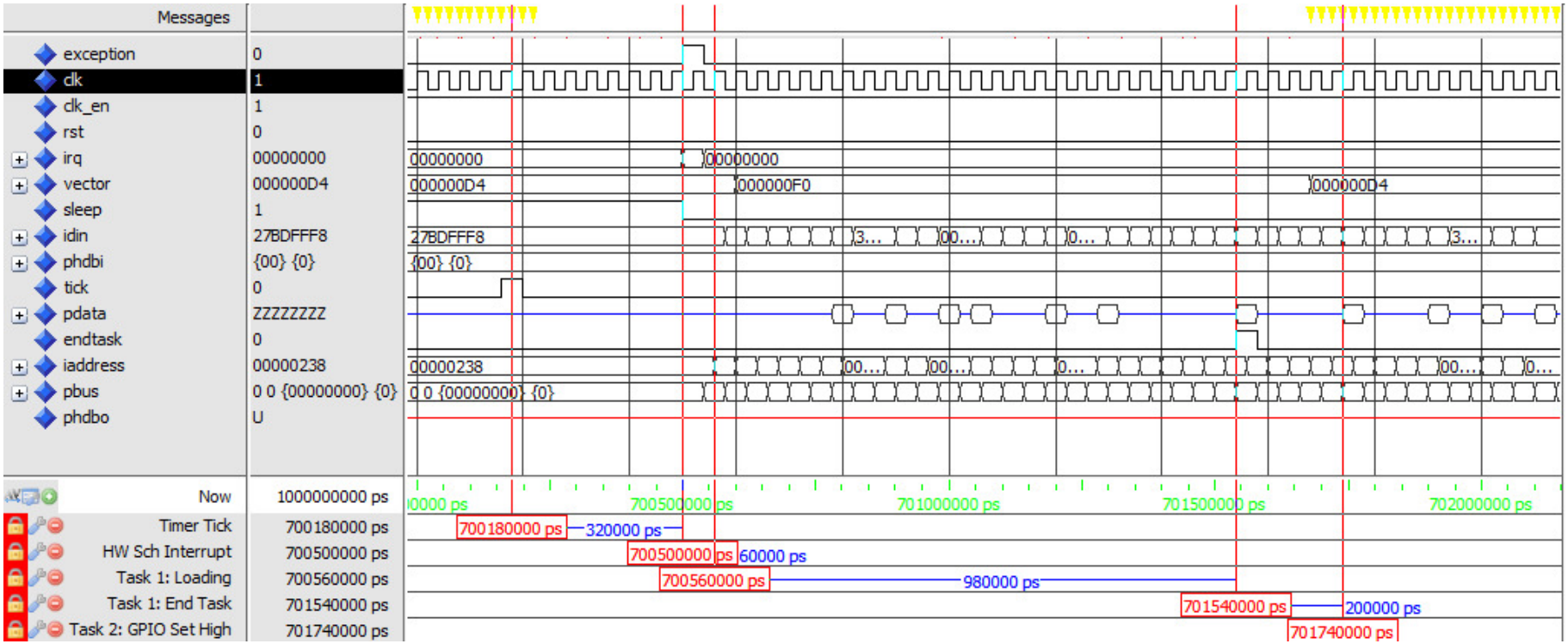


Figure 92: Simulated between task overhead

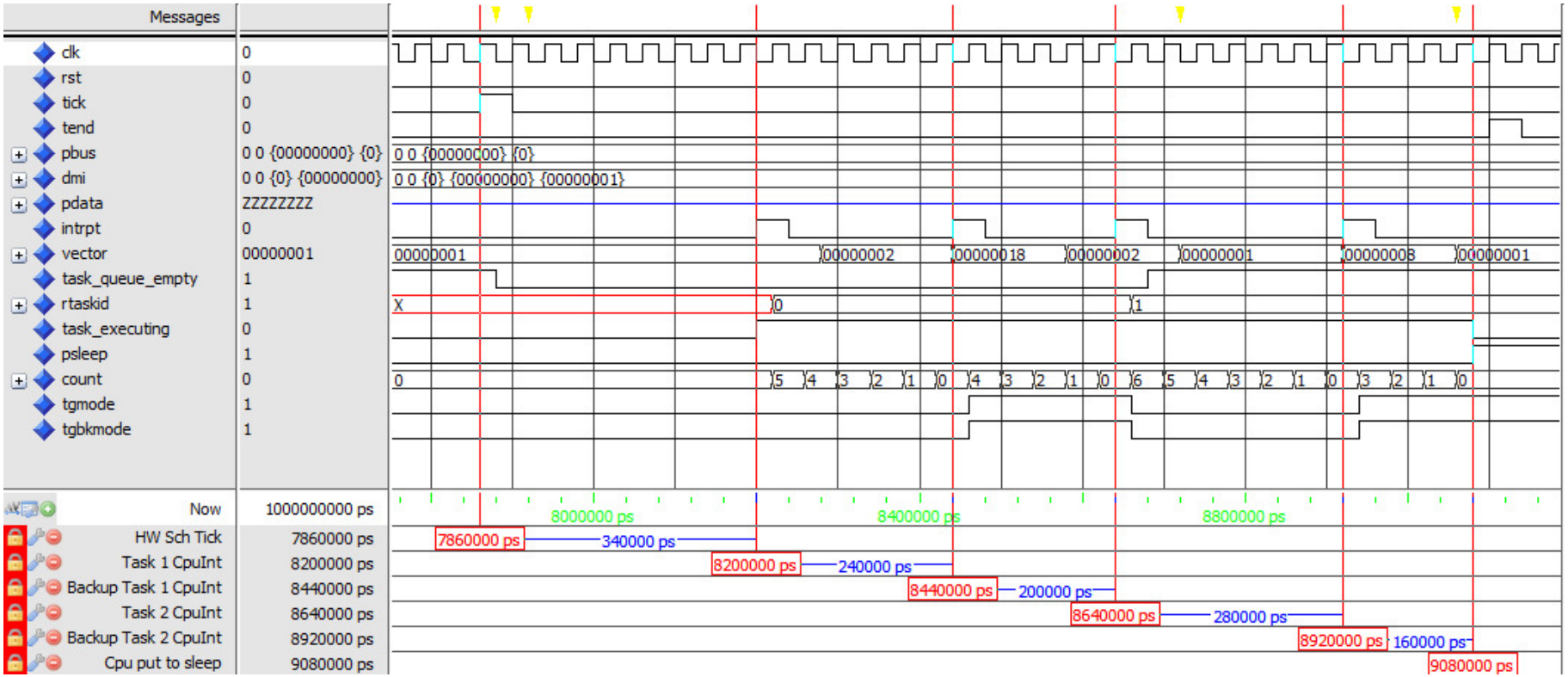


Figure 93: Modelsim simulation of the hardware task guardian unit in action

Appendix D: The evolution of the modern microprocessor

D.1 Introduction

This chapter aims to show that throughout the design and evolution of the computer, there has been an overriding interest in increasing performance which has led to a significant increase in complexity.

By modern definition, a computer is a machine that performs logical operations based on a list of instructions (Oxford-Dictionaries 2008). The name was originally used for people who performed numerical calculations, often for large banks with the aid of a mechanical calculating device. Today, the word computer is commonly associated with an electronic device that executes a stored program. As a result it can be difficult to identify a single device as the earliest form of computer.

Prior to the advent of machines that resemble today's processors, computers were generally very large, expensive and often unreliable. As with all digital computers, processors required switching elements to differentiate between discrete states. In early machines electrical relays and vacuum tubes (thermionic valves) were commonly used. When compared to mechanical designs, these systems had distinct speed advantages. However, they were generally unreliable, for instance, the EDVAC had an average error-free operating time of 8 hours (Weik 1961).

In December 1947, following Lilienfeld's patent (Lilienfeld 1926), scientists at the Bell Telephone Laboratories demonstrated their invention of the point-contact transistor amplifier (Bardeen and Brattain 1947). This began the work with transistors, which was then further expanded by Shockley (Shockley 1950; Shockley 1953; Shockley 1957; Ross 1998).

One transistor could replace the equivalent of about 40 vacuum tubes and was faster, more reliable, smaller, and much cheaper to build than a vacuum tube (Hames 2009). Transistors allowed for the design of more complex CPUs and facilitated the building of smaller machines. With this improvement more complex and reliable CPUs were built onto printed circuit boards containing discrete (individual) components.

Further work with transistor technology then paved the way in semiconductors for modern integrated circuits and microchips (Kilby 1976; Schroeder 2006). Following the invention of the integrated circuit (Kilby 1959; Noyce 1959), parts or whole circuits could be placed onto a single microchip. This meant that electronic circuits could be miniaturized and the number of transistors increased whilst keeping the cost low.

These advancements allowed for a great reduction in power, significant miniaturization of systems and consequently a significant increase in speed. As a result, Intel announced a new era in integrated circuits in 1971 (Mazor 1995). The natural progression for computing was then to attempt to put a computer on a chip.

Although it's often cited that in 1971 the Intel 4004 was the first microprocessor, this has subsequently been disputed (Logue 1998). Regardless of which microprocessor came first, there was an overwhelming and concurrent drive to make computers smaller and faster.

In 1974 the miniaturization continued and gave birth to the first microcontroller TMS 1000 (Deshmukh 2005), which not only contained a processor on a chip, but also the memory and digital IO. The microcontroller became a very popular platform for a wide range of embedded devices and meant that it could be used in areas where traditionally a computer would not have otherwise been considered (Zheng, Vladimirova et al. 2001).

Throughout the development of computers from the earliest digital machines such as the Zuse Z3, Colossus and the ENIAC, to the first microprocessors, microcontrollers and the processors available today, the advancements in silicon semiconductor technology has increased significantly.

During this progression the number of transistors that could be placed on a silicon die was increasing exponentially. This trend was known as Moore's law (Moore 1965; Adee 2008). Conversely as the silicon circuits grew, the speed of these circuits also increased.

This advancement of speed and silicon real-estate has been so significant that companies today make use of high level reprogrammable logic devices - such as Field Programmable Gate Arrays (FPGA) - to prototype what would normally be expensive and risky application-specific integrated circuit (ASIC) component developments. These generic FPGA devices are now fast and large enough to be used out in the field as a direct replacement to the ASIC (Tiggeler, Vladimirova et al. 2000; Zheng, Vladimirova et al.

2001; Fernandez-Leon, Pouponnot et al. 2002) and allow for more advanced features such as partial runtime reconfiguration (Vladimirova and Wu 2006; Wu and Vladimirova 2006).

More importantly, it can be shown that throughout the history of the computer, the processor market has been largely concerned at reducing size and maximizing processing performance. Some of these traits can be observed in microcontrollers and embedded systems. However, whilst the increase in hardware speed and silicon die area does not necessarily link to a reduction in predictability, this drive for increased performance has sparked the exploration of various developments in processor architectures.

D.2 The evolution of early CPU architecture

To meet the demands of technology and industry, the CPU architecture has evolved over the years to become the modern processor of today. This section will describe how some of the changes involved in the CPU architecture were driven mainly by the need for more performance.

D.2.1 CISC architecture

In the early days of computing, memory was often expensive, as were the computers themselves. Up to the 1970's, computers generally used magnetic core memory which was expensive and slow (Patterson 1985). Whilst RAM came shortly after and was faster, it too was expensive when it was just gaining popularity in the early 90s. This meant that good code was considered to be compact code.

In the 1960s to early 1970s the cost of software was far out growing the cost of hardware and even though compilers were being designed, the compilation stage took a long a time and the output was not very optimal (Hennessy and Patterson 2006). Along with memory limitations the best way to obtain compact and optimized code was to program in assembly (Hennessy and Patterson 2006). This meant that programming bug free code was very challenging and people would accept almost any code as long as it worked.

At the time, the state of the art in VLSI design was limited as it wasn't possible to fit much functionality on a chip. Due to the scarce on-chip resources, machines had their units spread across several chips. This limited overall performance because the delay and power

penalty of running data transfers between chips (Patterson and Séquin 1998). If it was possible a single chip solution would have been ideal.

Therefore, due to the cost of memory, the limitations of compilers and the demand for more complex applications there was a fear the industry was leading towards a potential software crisis (Patterson 1985). However, whilst the software costs were escalating, the hardware costs were starting to become cheaper. The reaction was to shift the complexity from software into hardware by implementing common complex functions in hardware. This was the main idea behind the Complex Instruction Set Computer (CISC) architecture, which was believed to provide some performance benefits (Patterson and Séquin 1998).

D.2.2 Load/Store architecture

In order to support CISC, it was necessary that the architecture should support complex addressing modes.

In the past, computers had a register known as an 'accumulator' (Ceruzzi 2003). The accumulator register would be used to store a value temporarily and be used in conjunction with another value to perform an arithmetic operation. This register would have specific instructions associated with it to load and store its contents from memory. This led towards the basis of the load/store principle.

A load/store architecture takes the approach that access to memory can 'only' take place by designated load and store instructions which place values into CPU registers (Null and Lobur 2006). All other instructions would then only operate on those values stored in registers. In other words, no other instruction can access memory directly other than specific load and store instructions ().

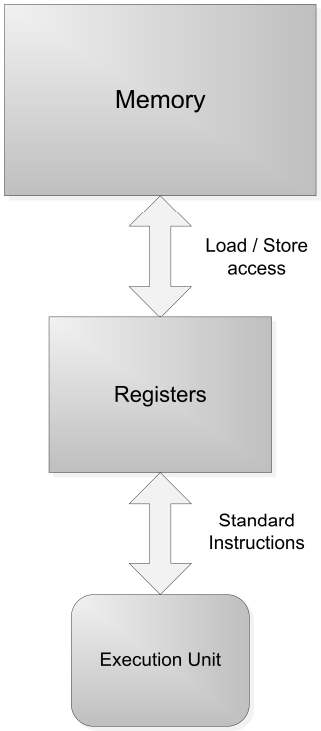


Figure 94: Load/Store hierarchy, reproduced from (Hannibal 2004)

A drawback of using the load/store principle meant the programmer had to spell out explicitly the steps required to do a particular operation (Hannibal 2004). For instance, a multiplication operation on two values in memory would require the following operations, where 'A' and 'B' are internal processor registers ().

```
LOAD A,(0x02)
LOAD B,(0x04)
MUL A,B
STORE A,(0x02)
```

Listing 19: Multiply in assembly language using Load/Store addressing

In the CISC architecture, it may be possible for the MUL instruction to be able to access memory directly. The result is that the architecture could be modified so that the above multiplication operation could be done within a single instruction as shown in . Please note that the resulting value is automatically written back to the address of the first operand.

```
MUL (0x02),(0x04)
```

Listing 20: Multiply in assembly using complex addressing

By being able to convert four instructions into one allowed for some large reductions in code size. This also meant that all the lower level operations could be hidden from the programmer leaving them to concentrate on the high level operations at hand. However, implementing these instructions was not easy and required more hardware resources (Hannibal 2004).

D.2.3 Microcode

Originally, instruction sets were hardwired into the processor where the circuitry to complete an instruction was controlled directly by combinational logic. This was known as direct execution and was fast and efficient (Patterson 1985). However, the problem was that it required circuitry for each new instruction and consumed quite a bit of silicon real estate. This becomes a bigger problem when introducing larger and more complex instructions which take a lot of work to execute.

Although hardware was becoming cheaper, the number of transistors was still limited and therefore directly executing CISC instructions was not really feasible.

A solution to this problem was to use a microcode engine which would decode a CISC instruction and translate it into several smaller microinstructions that could then be directly executed on the processor (Fiddler, Stromberg et al. 1990). This extra translation was slower than the direct execution method. However, the control memory ROM used to hold the microinstructions was about 10 times faster than magnetic core memory at the time so it still offered an acceptable performance (Patterson 1985).

Over time, microcode technology was improving and it made more sense to move functionality from slower and more expensive software to faster and cheaper hardware. To this end, instruction set counts grew requiring increasing numbers of microprogramming (Wilkes 1969). To accommodate the increase and keep performance up, the microcode routines had to be highly optimized and kept extremely compact in order to maintain low costs.

As the microprogramming increased it became increasingly difficult to test and debug the code. To fix a bug would often require the need to patch the microcode ROM out in the field. These difficulties then led to questions as to whether implementing all the elaborate instructions in microcode was the best use of resources (Patterson 1985).

D.2.4 RISC architecture

By 1981, technology had changed but the complexity of CISC implementations meant that most processors still spanned across multiple chips (Hannibal 2004). This was not ideal for performance. Therefore there was a drive for a single chip solution which would make optimal use of transistor resources (Patterson and Séquin 1998).

To fit the entire CPU onto a single chip, some functionality would have to be removed and tradeoffs made in favour of speed. Part of these tradeoffs was to see if all the additional instructions were required. Studies were therefore undertaken at profiling application code to find which instructions were used the most (Patterson and Ditzel 1980; Fiddler, Stromberg et al. 1990).

At the time, compiler technology was improving and memory costs were becoming cheaper. As a result, it was reasoned that high level language support could be better placed in software (Stallings 1988). This was backed up by the application profiling which showed that many of the elaborate instructions were not being used by compilers as they were difficult to implement (Patterson 1985; Jamil 1995). Instead compilers were opting for groups of smaller instructions which did the same thing (Hannibal 2004).

It was found that the extra instructions could be removed without really losing any functionality meaning the microcode engine was no longer needed, leaving only the simpler directly executing reduced instruction set. This trend of moving complexity out of hardware and back into software then gave rise to RISC (Patterson and Ditzel 1980).

D.2.5 Pipelining

Part of the RISC mentality was to make each instruction uniform in size and wherever possible take only one cycle to complete (Patterson 1985). This was possible due to the removal of complex microcode instructions and keeping the small and fast assembly instructions. This then made it feasible to employ a technique called pipelining.

Pipelining was a way that allowed separate portions of different instructions to be completed in parallel (Figure 8), thereby lowering the average number of cycles per instruction (Hennessy and Patterson 2006). This was only really made feasible by dealing with instructions which do not have varying degrees of complexity.

The outcome was that by reducing the number of cycles it took to process each instruction, performance would increase. However, since a program using RISC instructions required more instructions to perform the same task, and that the memory was slower than the processor's ability to execute instructions, then the memory bottleneck could negate any potential performance benefits.

D.2.6 Compilers, registers and high level languages

In order to deal with the problem of increased code size when using RISC machines and slow speed of memory access times, a few techniques were required to increase performance. The first step was to increase the number of processor registers. This helped because researchers noticed when doing code profiling, that 80% of the variables in a program were local scalars (Patterson and Séquin 1998). Therefore, rather than loading values directly to and from memory for each operation, as was common in CISC instructions, these values could be loaded into a processor register on which a number of operations are performed before only then writing the value back out to memory.

Since CISC architectures would often charge load/stores to individual instructions, the results was that more memory cycles were required compared to RISC architectures. The problem for CISC machines was that the compiler does not have the ability to arrange and manage load/stores intelligently for maximum efficiency. However, for RISC architectures the compilers role is more prominent if performance is to be maximised (George 1990).

The architectural decisions for RISC machines meant that hardware could be made simpler while software compensated by absorbing the complexity to retain a good level of performance. It is stated that the RISC and CISC design philosophies deal with much more than just the simplicity or complexity of an instruction set (Hannibal 2004).

D.3 Post RISC

Since the early RISC designs transistor counts have risen and architectures have taken advantage of the increased resources in a number of ways to gain increasing performance

D.3.1 Superscalar pipeline

One of the first architectural enhancements to emerge after the introduction of the RISC movement was the addition of extra parallel pipelines or parallel functional units in order to facilitate instruction level parallelism (ILP). This meant that the same parts of two or more instructions could be processed at the same time (Hennessy and Patterson 2006).

Using a second pipeline unit to support floating point instructions became an attractive option as processors were required to perform more mathematically complex operations. This type of pipeline could be allowed to execute a floating point operation in parallel with integer instructions. Prior to the split, the instructions proceed in sequential order through the pipeline stages ().

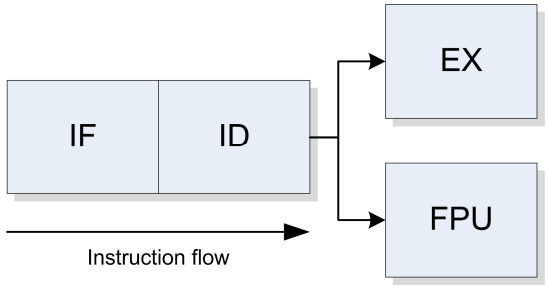


Figure 95: Superscalar pipeline with parallel floating point unit

The technique was then used to including even more functional units for the ability to allow several instructions to start each clock cycle (). However to make efficient use of the multiple execution pipelines an instruction scheduler was employed to deal with instruction dependencies.

For dynamic multiple issue superscalar pipelines, instructions would be grouped for execution at runtime (Patterson and Hennessy 2005). Older superscalar architectures would issue multiple instructions in-order only where dependencies can be met. This meant that there would be conditions where some execution units would remain idle until it was possible to issue the next instruction sequence in parallel.

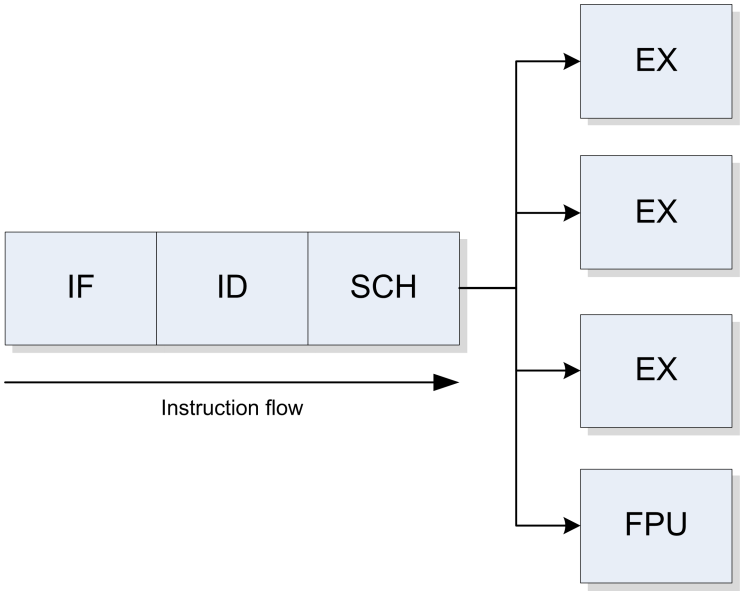


Figure 96: Superscalar pipeline with multiple parallel execution units

This system gave rise to a new set of complexity to schedule parallel instructions at runtime whilst allowing the hardware to take into account and manage the dependencies dynamically.

D.3.2 Very Long Instruction Word

Due to the unpredictable nature of estimating how the instruction scheduler would affect performance, DSP manufactures who rely on fast and predictable performance decided to adopt different solution.

The ability to process several instructions at the same time was a great performance benefit, however dynamic multiple issue pipelines made timing more complex. Therefore, a solution was sought to group instructions statically for execution at compile time. This technique is known as the Very Long Instruction Word (VLIW) which contains several operations that the compiler has determined can be issued independently as a group (Hennessy and Patterson 2006). Each operation bundle gets executed in parallel without the need for a hardware scheduler.

Initially VLIW was not very popular because compilers were not efficient at dealing with instructions dependencies and ordering the instructions for maximum ILP (Hennessy and Gross 1982; Hannibal 2004). However, due to the power of modern compilers VLIW is

more feasible and is in keeping with the RISC tradition of putting the complexity from hardware back into software.

D.3.3 Out-of-order pipelines

Whilst VLIW had limited attention, the problem of data dependencies in early in-order superscalar architectures meant that there would be parallel execution units which would often sit idle. This then led to a technique for executing instructions out-of-order to that of the program text. This meant that the hardware instruction scheduler could dynamically reorder the instructions at runtime to optimize the code so that it can get maximum usage out of all the parallel units.

The out-of-order superscalar pipeline technique is very popular in modern desktop architectures as it provides good average-case performance. However, it introduces a large amount of complexity into the pipeline and the behaviour of the instruction scheduler is not visible to the programmer who will be unaware which instructions will be issued as a group (Bate, Conmy et al. 2001).

D.3.4 Branch Prediction

A problem common to pipeline architectures is the condition when the sequential flow of instructions is altered. This flow can be interrupted when executing branch or jump instructions. When one of these instructions is executed it can result in parts of the pipeline being flushed and has the knock-on effect of reducing performance. Therefore modern processor architectures employ various techniques to counteract the wasted cycles.

The cause for pipeline stages being flushed is that the test condition for a branch instruction may not be known until a later pipeline stage (Figure 97). This means that the previous pipeline stages may need to be flushed when the test condition is true. This is a problem that can increase as the number of pipeline stages in the processor design increases.

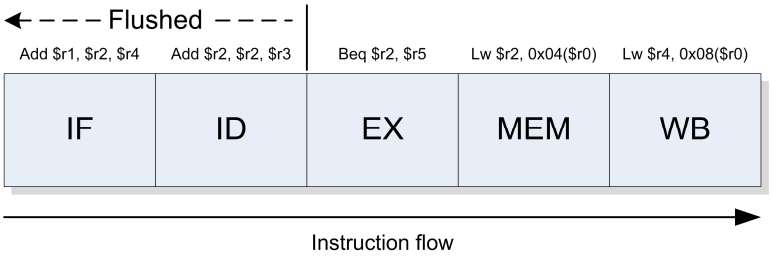


Figure 97: Flushed pipeline stages due to a branch instruction

One way the performance loss can be reduced is to perform the test condition in an early as possible pipeline stage and therefore reduce the number of stages that would need to be flushed. However, this is not always practical.

Another approach to the problem was to allow the instructions that would ordinarily be flushed to execute regardless if the branch condition was taken. It would then be up to the compiler to locate suitable instructions to place in these locations, which are also known as branch delay slots (BDS). If the compiler was unable to re-order the existing instructions into the slot, then it would use no-operation (NOP) instructions. Therefore in the case of NOP instructions there was no performance gain and the code size was increased. This then led to the use of branch prediction techniques.

One of the simplest forms of branch prediction is the static BTFN (backward branches taken, forward branches not taken) predictor which is said to be about 60-70% accurate for typical embedded applications (Berg, Engblom et al. 2004).

A further approach to branch prediction which is often the only feasible means of decreasing the branch penalty for superscalar architectures is to predict dynamically which way branches are likely to go based on previous executions. This technique does have additional hardware complexity over the other methods due to the need to store the data about previous executions.

D.3.5 Exceptions

Exceptions and interrupts suffer from similar performance problems faced by branch and jump instructions in that traditionally parts of the pipeline are flushed. This allows the old uncompleted instructions to be removed whilst the new instruction stream from the interrupt service routine is executed.

Exceptions such as hardware malfunctions and 'prefetch aborts' the CPU is often only required to terminate execution which is comparatively simple to implement. However, exceptions such as 'user calls' and interrupts require that execution be restarted once the handler has completed. It must then be possible to shut down the pipeline safely and save its state so that the uncompleted instruction stream can be resumed.

The steps for performing an exception or interrupt are as follows (Hennessy and Patterson 2006):

1. Force the first instruction of the handler into the pipeline on the next instruction fetch.

2. Until the interrupt or exception handler instruction is processed, flush out all the pipeline stages from the faulty or uncompleted instruction (). (This is often implemented by zeroing out all the write control lines in those respective pipeline stages and therefore preventing those instructions from making any state changes).

3. The program counter (PC) of the faulty instruction is saved so that it can be used later when the handler returns.

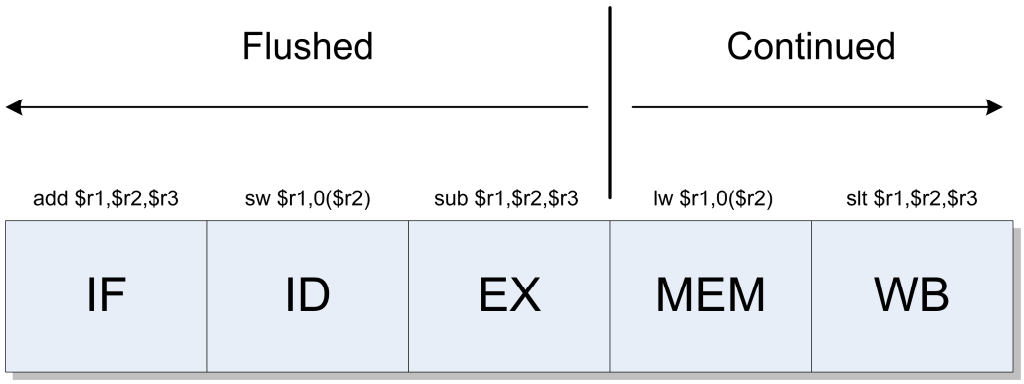


Figure 98: Flushed pipeline stages due to an exception or interrupt

When an exception or interrupt occurs on branches it is no longer a simple matter of re-creating the processor state because the instructions in the pipeline may no longer be sequentially related. Often this may result in not returning to the next instruction after the handler, but re-executing the branch condition in order to determine which instructions should follow.

Whilst it is easier to implement precise exceptions for in-order integer pipelines, there are cases for high-performance CPUs using an out of order superscalar architecture where using an imprecise mode can yield a greater than ten times performance. This is because instructions such as floating point operations may run for several cycles and could be allowed to execute in parallel to the normal pipeline. In such cases, an exception for a new instruction may occur before the older floating point operation has completed or even had chance to raise its own exception. In such a scenario, it can then be hard for someone debugging the system to ascertain which instruction caused the exception or in which order they were executed.

The problem of keeping track of the order in which exceptions should occur doesn't only relate to out-of-order superscalar pipelines but also to simple scalar pipelines. This problem arises due to the possibility of multiple exceptions occurring during the same cycle.

Pipeline stage	Exceptions that can occur
IF	Page fault on instruction fetch, misaligned memory access, memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch, misaligned memory access, memory protection violation
WB	None

Table 28: Exceptions that can occur in the MIPS pipeline (Hennessy and Patterson 2006)

Table 28 lists the possible exceptions that could occur on a simple 5-stage pipeline.

An example of a problem that can occur is where a load operation in a MEM stage and an ADD instruction in the EX stage causes a page fault and an arithmetic exception to arise at the same time (). To implement precise exceptions, the goal is to handle these in the correct order. However, the problem can be even more complicated.

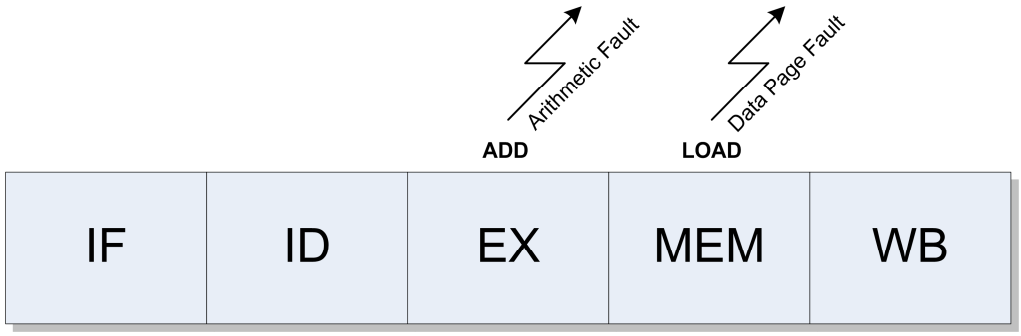


Figure 99: Problem when two exceptions occur at the same time

In some scenarios, exceptions can occur out of order. A new instruction may cause an exception before an older one. For instance, an ADD instruction might cause an instruction page fault in the IF stage before the previous LOAD instruction creates a data page fault when it reaches the MEM stage ().

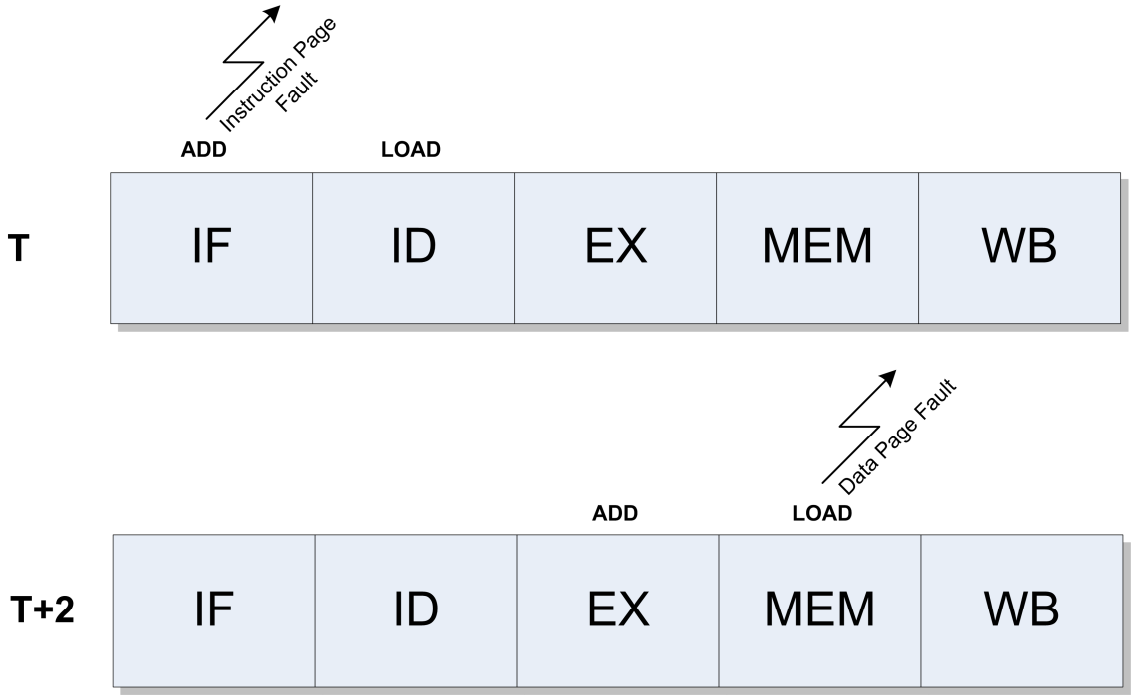


Figure 100: Problem when exceptions occur out of order

In this scenario, the pipeline cannot handle the exception as it occurs in time as this would lead to exceptions being handled out of order.

A common solution to this problem is to store the exception status of each instruction as it filters down the pipeline and only act upon it before it reaches the end of the MEM stage and before any state changes are committed to registers or memory. At this point, the

exception status flag is then checked and the exception is raised in the correct sequential instruction flow order. The drawback of this method is that most of the pipeline stages get flushed when an exception or interrupt occurs and performance is subsequently reduced.

This solution doesn't work for all processors as some have instructions that change the states during the middle of execution. In some cases registers or even memory are modified. Therefore if the instruction is aborted, there needs to be a way to undo any committed changes.

D.3.5.1 Re-order buffer

There can be problems when an interrupt or exception occurs on an out-of-order superscalar pipeline as several instructions may be in flux and state of the registers unknown. Therefore, one method to allow instructions to complete out-of-order and maintain the precise interrupt rule is to retain the results of each instruction in a reorder buffer. The results of executed instructions are only then committed once all the previous instructions are free of exception conditions.

A negative side to the reorder buffer method is that there can be a performance loss because the processor cannot issue an instruction which depends on a result currently being held in the reorder buffer until it writes that result into the register file.

D.3.5.2 History Buffer

The history file method provides a solution to the performance issues encountered in the reorder buffer method. Instructions are allowed to complete in any order and update the register file immediately upon completion. However, when an instruction is issued, the processor saves the previous state of any modified registers to a history buffer. This allows the reconstruction of the processor's sequential state in the event of an exception.

D.3.5.3 Future File

A similar technique to the history buffer solution is the future file method which contains two register files. These are known as the current working register file and the future file. Instructions will execute out-or-order and update the contents of the future file. When the previous instructions become clear of any exceptions, the working register file is updated

from the future file. When an exception or interrupt occurs, the future file is restored with the contents of the working file and thereby retains the precise exception status.

D.3.6 Modern CISC-RISC

Compared to previous computer history, memory is now both cheap and fast. However, the speed of memory has not kept pace with the speed of processors and manufactures are beginning to hit the bottlenecks on silicon speed. The problem is no longer how to fit the required functionality on a chip, but what can be done will all the available transistors. With the need to keep memory bandwidth up and execution fast, there has been a return to hybrid CISC-RISC architectures. In modern desktop processors the internal core is often RISC based but presents the programmer with a rich collection of CISC instructions. Compilers have also been advanced to make use of the CISC instructions. Advancements have also been made in the developments of memory systems to reduce the memory performance impact. This will now be described in the next section.

D.4 Memory

Another area of study that has undergone development in modern processors is the memory architecture. In computer systems memory is used for storing working data and program storage. These can be arranged into two commonly known models; the von Neumann or Harvard architecture.

D.4.1 Von Neumann

The von Neumann architecture - which is most commonly used in computing systems (Null and Lobur 2006) - stores both instructions and data in the same memory system (). This allows for practices such as self modifying code and dynamic loading of programs. However, a drawback is that without appropriate memory protection, accidental or malicious overwriting of code can have problematic and security implications. For instance, a buffer overflow attack can be used by Malware to overflow the call stack to overwrite or modify an existing program in order to achieve a higher privilege level to carry out the desired operation.

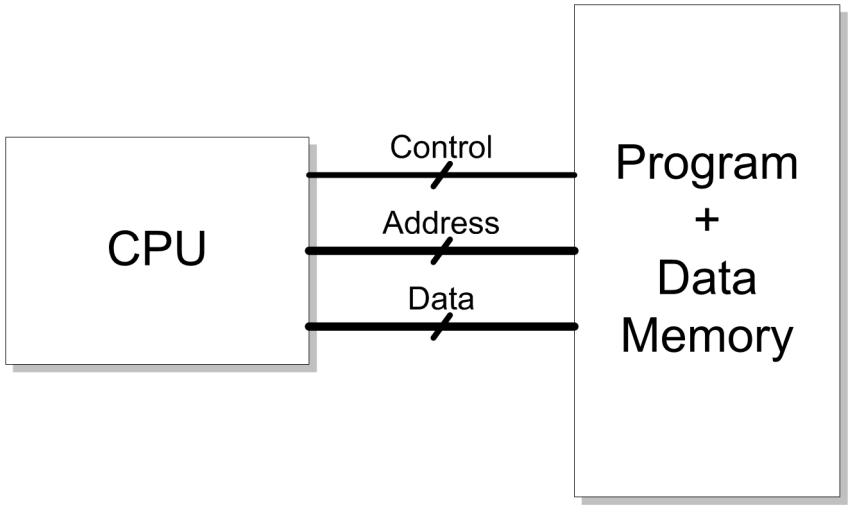


Figure 101: Von Neumann architecture

The von Neumann architecture also presents a serious bottleneck where both data and instruction fetches occur at the same time. This is common in pipelined processors where the CPU must be halted until both memory operations are carried out. On some memory systems this can be a lengthy process since the time for each access to memory may take a number of CPU cycles.

D.4.2 Harvard

The Harvard architecture utilizes a separate storage and bus medium for instructions and data (). Most computers which are documented as Harvard are in fact based on the modified Harvard architecture where it is possible to gain access to the program memory through the data bus.

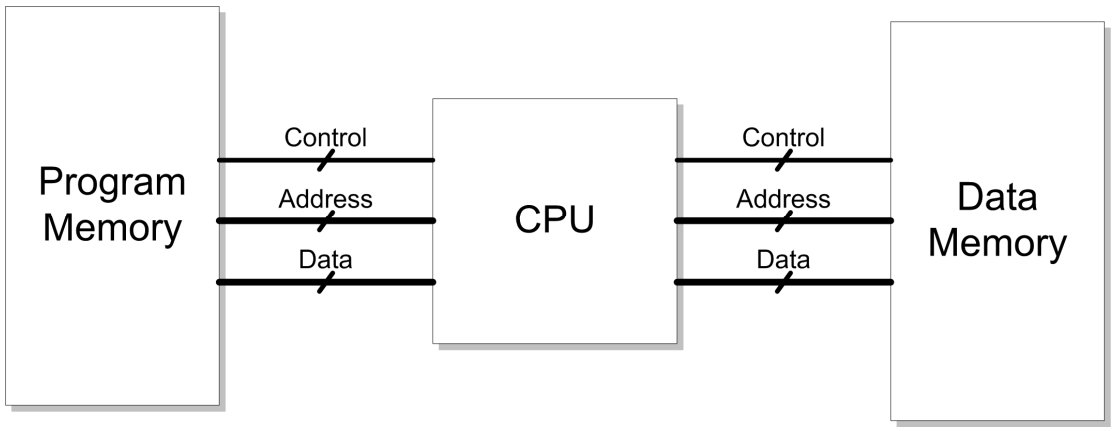


Figure 102: Harvard architecture

A disadvantage with the pure Harvard architecture is that programs cannot be loaded at runtime and then executed. Therefore in pure Harvard architectures the program is often preloaded in a read only memory so that the computer can begin execution as soon as the power is applied.

D.4.3 Performance

The performance of a computer not only depends on the processor architecture but also the speed of the memory system. Ideally the memory would be as fast as the processor and accessible in a single clock cycle. However, this is only true for very expensive memory which is often only available in small quantities due to its cost. The problem is that memory speeds are increasing at a slower rate than processor speeds (Harris and Harris 2007), as shown in .

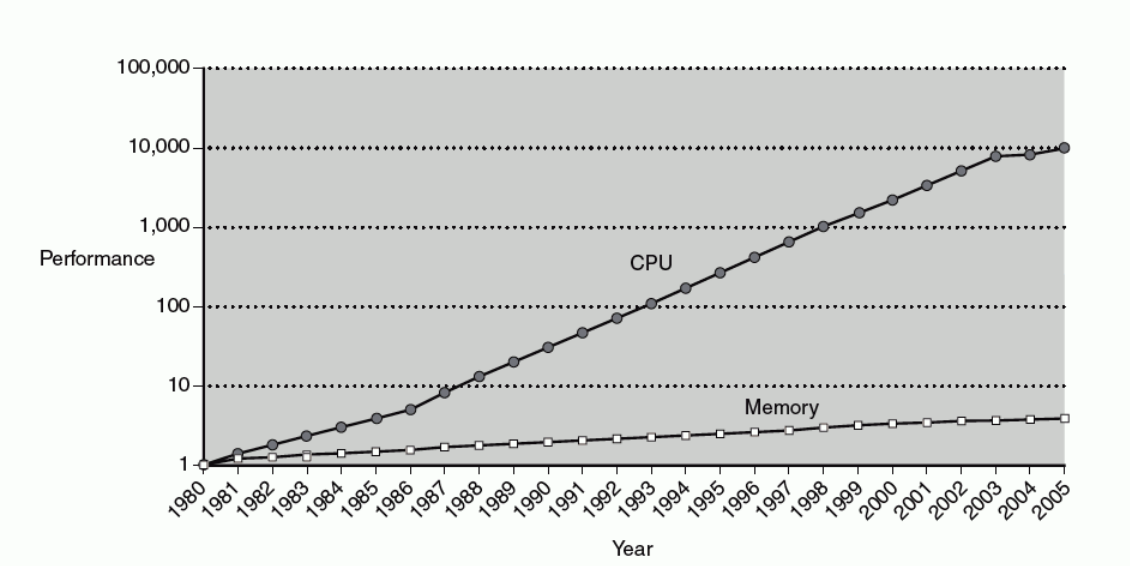


Figure 103: The gap in performance between processors and memory over time (Patterson and Hennessy 2005)

In modern computers main memory is often implemented using dynamic random access memory (DRAM), while faster memories such as cache are implemented using static random access memory (SRAM). DRAMs are typically made from large banks of capacitors and have better density than SRAM. This means that DRAM can have a larger

capacity for the same amount of silicon and are therefore cheaper (Patterson and Hennessy 2005).

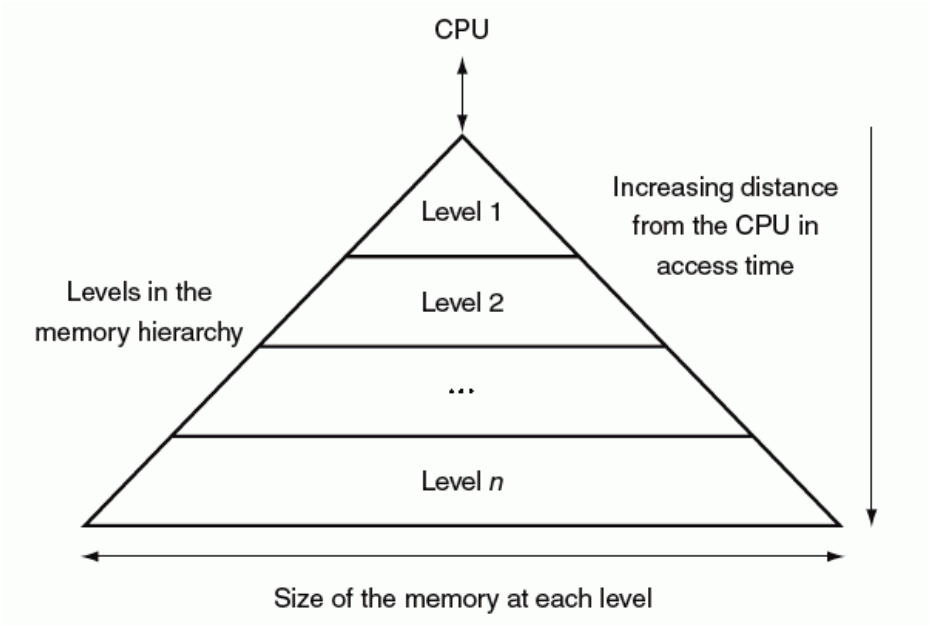


Figure 104: Memory hierarchy model (Patterson and Hennessy 2005)

Due to the difference in price and access times it is often advantageous to build memory as hierarchy of levels (). In this way by using the principles of locality it is possible to provide the quantity of cheaper memory at the speed closer to that of the fastest memory (Patterson and Hennessy 2005).

D.4.4 Cache

Cache is a small amount of fast memory located near the top of the memory hierarchy which can temporarily hold various sections of code and data based on the rules of locality (Hennessy and Patterson 2006).

A cache hit occurs when a CPU performs a memory access and finds the item stored within the cache. Conversely a cache miss occurs when the CPU does not find the item in cache.

If a cache miss occurs the processor can be paused while a block (fixed collection of data) containing the required word is fetched from main memory. The rules of temporal locality assume that it is likely that the CPU will require the word again in the near future. Spatial

locality makes the assumption that it is highly probable that the rest of the data in the block will be needed soon.

The process of cache hits and misses are handled in hardware. During a cache miss the time required to fetch the first word in a block is known as the latency and the time to receive the whole block is known as the bandwidth (Hennessy and Patterson 2006).

The variation between the cache hit and miss times and the difficult nature of predicting which items are currently held in cache make determining tight BCET's and WCET's highly complex.

D.4.5 Direct memory Access

Direct memory access (DMA) is a hardware unit that allows data to be moved between IO and memory without the processor needing to intervene. This mechanism frees up the processor from doing tedious data transfers to generally slower IO. A negative aspect is that the DMA shares the data bus with the processor and therefore they can interfere with each other as they gain control of the bus.

D.4.6 Memory management unit

Memory management units (MMU) are a hardware mechanism that provides a number of useful functions such as paged memory, memory protection and translations between virtual and physical addresses. With paged memory, the address range is divided into manageable chunks known as pages. These pages are then used by memory allocation schemes. When a program is loaded into memory it may consume a number of pages whose locations may be non-consecutive. The program will then operate based on a virtual address which makes the memory region appear consecutive to the processor even though the addresses may not reflect the real addresses where the program is stored. A virtual address contains a page number and offset address which is translated through a page table to convert the virtual addresses to physical addresses. A translation lookaside buffer (TLB) is sometimes used to cache page table entries to help increase performance. Memory protection is sometimes included and can allow areas to be set with different privilege levels or be flagged as execute, read or read/write only.

A MMU can benefit full scale operating systems greatly where dynamic memory allocation and separation of tasks are concerned. However they can impose a significant overhead on memory access times especially if the page table entries are not held within the TLB.

D.5 Conclusions

This chapter has provided an overview of computer architecture and its developments to some of the features found in modern processors. It has been shown in the past that a drive for miniaturization, reliability and performance has resulted in modern computer architecture exhibiting some highly complex features. Although this has benefited performance, the increase in complexity makes it harder to predictably make precise estimates of the systems behaviour.