Abstract

The aim of this project is to investigate power consumption, at the software level, by the various instructions and creating a power profile that can later be used for estimating power consumption of the dynamic system workload. This project was carried out in association with XMOS using their XCore XS1-L1 device. This project was undertaken due to the increasing demand of identifying power optimization methods at the software level, where the predicted power saving potential is massive.

The background of this project explored various techniques of power optimization currently employed in the industry, the various components of power consumption, and the research being carried out to find new ways of optimizing power at the architectural level where the opportunities for power saving are the maximum. The project design included development of the framework, benchmark programs and the technique used to measure the power consumption. The findings of this project are quite interesting and some quite out of the ordinary. A list of contributions and achievements are summarized below:

- A method for measuring power consumption at the instruction level was successfully developed, see pages 27 -31
- This project resulted in some very interesting results, see Chapter 4

With some improvements, this project can be used as a basis for further study and the results from the power profiling and the combinational experiments can be used to derive an equation for estimating the dynamic power consumption of the system. This can also be integrated into a tool which can potentially do some sort of static power analysis and recompile the code for optimized power consumption.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my family, whose constant encouragement has kept me going for the whole year even though I'm so far from home. I also want to thank them for their financial support, without which this course would never have been possible.

I would like to thank my supervisor, Dr Kerstin Eder, for her involvement and supervision over this project. The inputs and material provided by her have proved more than valuable on several occasions.

Also a big thanks to Henk Muller and XMOS for providing the hardware setup and the information that enabled me to carry out this project successfully.

A big thank goes out to all my friends and class mates for their constant help and support at whatever time of the day (or night) it was needed. They have played a big part in my development over the past year and have always been by my side when most needed.

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Pranay Dewan, September 2010

Table of Contents

Chapter 1: Aim & Objectives

1.1 Introduction

Today some of the very powerful processors can consume almost 100W – 150W of power, with an average power density of 50W - 75W per square cm [25]. The immense power density creates problems for packaging, cooling, reliability and leakage current. This is a major issue for data centres and server farms where the energy consumption in 2006 was 60 billion kWh and it is estimated that by 2011, this would shoot up to 100 billion kWh with an approximate cost of $7.4 billion annually [22]. Even the handheld devices which are battery operated, the numbers may not be as large, but the problem persists when more and more features are added and the battery life keeps declining [25]. For all applications, the latest concern is reducing power consumption in order to keep up with the performance trends.

As optimization at the lower levels of abstraction reach their limits, they threaten a primary source of value in the computer industry, namely ongoing performance increases [1]. The need of the hour is to understand power management at the higher levels of abstraction, which offers greater flexibility and a larger impact as it eliminates the tedious low level manual interventions to make optimizations, and instead have a push button solution that would save time and engineering effort. Significant static and dynamic power reductions are possible, but require a good understanding of the system workload. This workload can be understood better by investigating performance (in terms of power consumed) of the system in the form of resources needed and the load profiles [21].

A recent tech talk by Mentor Graphics [21] showed that power optimization at the Register Transfer Level (RTL) and Gate level has a potential of around 20% whereas the optimization at the architectural (higher) level has a potential of almost 80% due to the flexibility provided by the higher level of abstraction. This investigation aims at bridging the gap between the power unaware software developer and the data that is available on the hardware side, thus helping the software developer make informed decisions when coding potentially power intensive applications. This project is carried in XMOS using their XCore XS1-L1 device. A particular focus of this project is to investigate, for XMOS, the effectiveness of their power saving features put in place on their XCore design. This investigation is expected to lead to the introduction of transparent power management at the earlier stages of the system development and hence form bedrock for formalizing methods in energy efficient and power aware system design. A research on this novel subject will contribute to the increasing demand for extremely high performance to execute many computationally intensive applications using only a small battery for power, making this an interesting area to work in.

1.2 Aim & List of Objectives

The main aim of this project is to investigate the relationship between the power consumption, which is usually done at the silicon level, and the application code, which is power unaware. This can be done by correlating functions of the application and power effectively obtaining an energy profile for different workload patterns, and with the help of a tracer that is available, we can see what was executed at what time and hence attribute power consumption to the respective instructions executed. The idea behind this is to do an instruction level power profiling which can later be used to calculate the overall power consumption of a piece software based on its dynamic workload. Once we have the basic power profile, this can be extended to the software level to see if this basic instruction level power profile holds good at the software level, where many instructions are clubbed together and executed.

The main objectives of this project are:

- Review the literature on low power design essentials
- Design the framework to be used for this project
- Design benchmark programs with various workload patterns
- Measure the power consumed for the various workload patterns
- Correlation of the data gathered, attributing the power consumption to the respective instructions
- Compile an interim and final dissertation report
- Develop a presentation of the achievements and also the complexities of this project

Moving on from the above mentioned wider context to a more specific context of this project, we intend to focus on the following main categories of tasks:

- Workload patterns
- Power consumption measurement
- Correlation of power consumption data based on workload

Workload patterns or the system workload refers to the stimulus that needs to be applied to the system, in this case the various benchmark programs, to simulate a usage pattern, in this case the power consumption at the output of the processor. Understanding what the system has to do and what are the performance criteria will enable us to compare the static and dynamic power profiles along with the system workload. Once a link can be established on doing the comparison, we can correlate the power data obtained to the workload and investigate how the software impacts the power consumption of the overall system.

1.3 Organisation of This Report

The main chapters in this report form the main contents of this dissertation complimented by a set of appendices that provide essential parts of the code.

Chapter 1 Aim and list of objectives of the project that were decided upon at the start of this project

Chapter 2 Background and context of this project and its relation to the work already done

Chapter 3 Project design methodology

Chapter 4 Results and their analysis

Chapter 5 Critical evaluation of the work done with respect to the positives and pitfalls of the project and a brief conclusion

Chapter 6 Possible improvements and further work

Appendix A Snippets of the framework source code designed

Appendix B Snippets of few of the benchmark programs designed

Chapter 2: Background & Previous Work

2.1 Introduction & Outline

Power consumption by a processor is the consumption of electrical energy which is used for either switching the devices internal to a processor such as transistors, or via the loss of energy in the form of heat dissipated due to the electronic circuits that are present in a processor. The main aim of the manufacturers today is to minimize this power consumption and yet be able to perform computationally intensive processing.

Moore's law [2] has been steady over the past few years, with the number of transistors on a single piece of chip doubling every eighteen months. The need for optimizations is on the brink because of this steady increase in the hardware capability, the desire for computationally intensive and faster processing has been ever increasing and now the demand is to perform these intensive functions with the same or less electrical energy that has been used in the past [1]. Plus users expect more and more out of their mobile devices, such as continuous access to the web, uninterrupted signal clarity, faster operating systems, and huge amounts of storage space and applications that necessitate large amounts of processing, all this just on a singular battery that is expected to last for long. To do all this, we need to rethink about the way we design systems and not just look at the performance-cost product but instead consider the power-performance-cost product. The trend of power requirement and power consumption, as shown in Figure 1 [5], shows us that based on this ITRS roadmap for SOC Consumer Portable Power Trend, the trend is going to become more and more severe going into the future. The chart is broken down into static and dynamic power, the left hand axis shows the power consumed in milli watts (mW), the trend is for more power to be used by these devices, but the requirement still remains constant at around 1W. Unfortunately, IC designers have this misconception that the power trend will fade away [7], but from the graph, it looks otherwise.

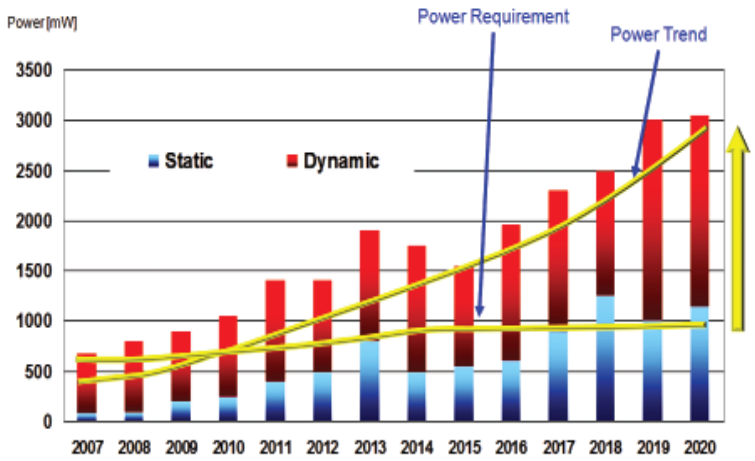

Figure 1: Power trends are outstripping power requirements [31]

For the last couple of decades, researchers have focussed their attention on optimization of power based on design features or the micro architectural level. As a result parallel processor architectures came into existence with pipelined and superscalar processors [3] with parallel decoding and features like shelving, register renaming and reorder buffers. Many branch predication algorithms and multi threaded architectures were being convoluted into the processors and marketed as a means of speed and power optimizations. A study [4] shows that the two main reasons for general purpose processor optimizations in the industry have been because of pipelining (resulting in increased clock speeds) and instruction level parallelism (resulting in a high instruction throughput). The improvements from both these techniques are now diminishing [1] since the current designs have very little logic to be implemented in the various pipeline stages and instruction parallelism has reached its limits. These micro architectural optimizations are reaching the limits resulting in a halt to the ongoing increase in performance.

Power introduces a number of unique challenges to IC designers as we work with technologies smaller than the 100nm mark [7]. Various methods are used by the designers to optimize power such as:

- Power gating – shutting down portions of the chip
- Power gating with retention and isolation – decision of retaining the state and how to restore the state once the power is back on
- Body bias – limit leakage and improve the performance [6]
- Dynamic Voltage Frequency Scaling (DVFS) – applicable to dynamic power

Most of the above now have diminishing effects on the leakage current as the processor technology gets smaller, and hence static power dissipation is of primary concern in designs today accounting for 50% of the power dissipation for the high end processors with 90nm technology [1].

As optimizations at the lower levels of abstraction reach their limits, we need to explore power management at the higher levels of abstraction which is in line with the main aim of this project, to investigate the consumption of power at the software level and finding ways to reduce the overall power dissipated by a processor.

To understand better the specific context of this project, we need to first understand the background of some of the key concepts like:

- What Is Power Consumption
- The Two Components Of Power Consumption
    - Static Power
    - Dynamic Power
- Current Techniques Of Reducing The Power Consumption
    - Hardware Techniques Of Power Optimization
- The Link Between The Various Layers Of Abstraction
- The Need To Optimize Power At The Architectural Level
- Ways To Optimize Power At The Architectural Level And The Research Done So Far
    - Software Techniques Of Power Optimization

In this chapter we will first understand the background of how power is consumed in a processor with the two components of power consumption. Once this is understood, the chapter moves on to explain the current common hardware techniques of power optimizations. Then before moving on to the software techniques of power optimization, we understand the link between the various layers of abstraction and also the need for optimizing power at the software or architectural level. These key concepts form the outline of this chapter and are discussed in detail in the background section of this report that follows.

2.2 Background, Context & Previous Work

2.2.1 Power Consumption & Its Components

The total power consumption in a processor due to in the internal circuitry is due to two components [8] of the power:

- Dynamic Component
- Static Component

Dynamic power depends on the internal switching, the instruction execution sequences, and the instruction execution rate and also the data operands involved. Static power depends on the temperature and voltage of the system and is consumed even though the circuitry is quiescent [9] which is a result of leakage current. When all the inputs are held at some logic levels and the state of the circuitry is not charging, that is when there is some leakage current [10] which results in static power consumption.

Power consumption of each device in the circuit should be known and minimized to the extent possible, hence power calculations is important from the point of view of power supply sizing, current requirements, device selection requirement, cooling and heat sink requirements [10]. Reduction of this power consumption has been the main goal of many designers till date and a good knowledge and understanding of what causes this power consumption and the ways it can be minimized is important from this project stand point. Hence in this report we will discuss the ways power is consumed (dynamic and static) and then move on to the techniques used to minimize this consumption of power.

2.2.1.1 Static Power Consumption

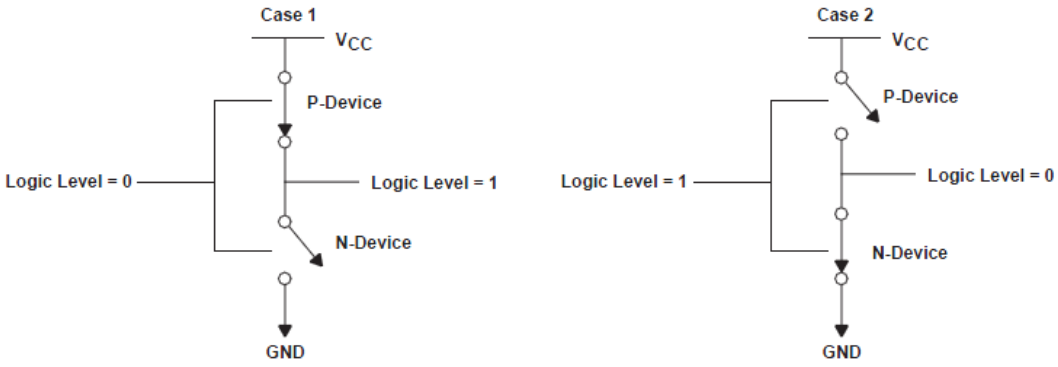

Figure 2: CMOS Inverter for Static Power Analysis [10]

Ideally in the scenario shown, the static power consumed should be zero [10]. As shown in Figure 2, for Case 1 if the input to the gates of the CMOS logic is at logic level 0, then the p-MOS device is ON and the n-MOS device is OFF. The output of the inverter is at logic level 1, thus causing an inversion of the input. For Case 2, if logic 1 is applied to the gates of the CMOS logic, then the p-MOS device is OFF and the n-MOS device is ON. Thus the output of the inverter is at logic level 0 causing an inversion. At any point one of the devices is always OFF, and since no current flows into the gate terminal, the resultant steady state current is zero.

However, a small amount of power is consumed (static power) due to the reverse bias leakage between diffused regions and the substrate. To understand this leakage better, we will look at the model shown in Figure 3 [10].

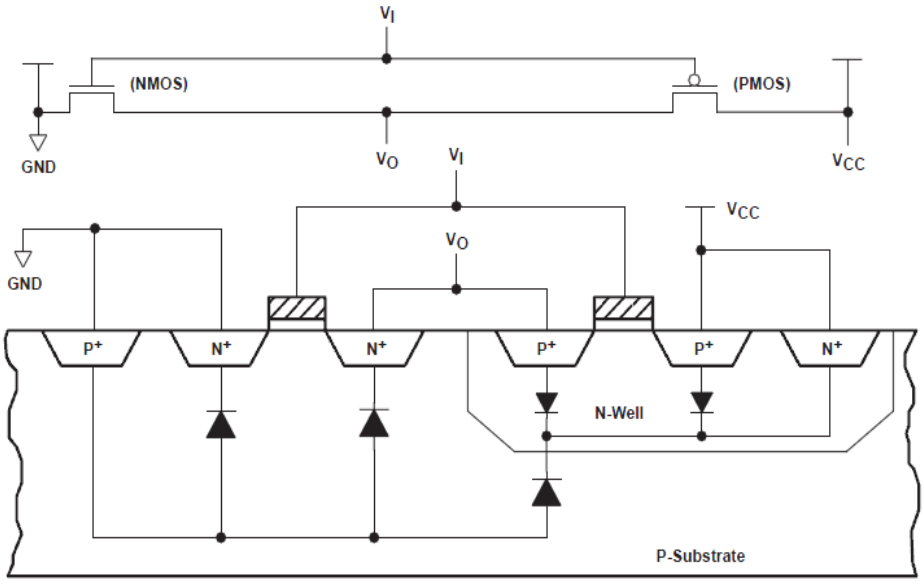

Figure 3: Model to show parasitic diodes in an Inverter [10]

This leakage current is due to the parasitic diodes shown in the model of Figure 3. These parasitic diodes are formed due to the source drain diffusion and the N-well diffusion. As seen in Figure 3, they appear in the N-well and the substrate and since these diodes are reverse biased, they are the main source for the leakage current (static current) and is given by:

$$I_{\text{leak}} = i_s \left(e^{\frac{qV}{kT}} - 1 \right)$$

Where:

i_s = Reverse Saturation Current
V = Diode Voltage
k = Boltzmann's constant (1.38 × e–23 J/K)
q = Electron Charge (1.602 × e–19 C)
T = Temperature (Kelvin)

Power, as we know is the product of current and the voltage, and in this case, total static power is the product of the leakage current and the supply voltage,

$$P_{\text{s}} = \sum (Leakage\ Current) x (Supply\ Voltage)$$

Ways to reduce static power [9]:

- Selectively used ratioed circuits
- Selectively use low threshold devices
- Reduce the leakage current by using stacked devices or Body Biasing
- Power Gating

2.2.1.2 Dynamic Power Consumption

Dynamic power of a circuit can be calculated adding up the transient power consumption and the capacitive load power consumption [10]. Transient power is consumed when the current flows due to the switching of the devices from one logic state to another and the frequency of switching, the rise and fall times of the input signal, as well as the internal nodes have a direct correlation to the transient power. Additional power is consumed to charge the external load capacitor which is dependent on the frequency of switching, called the capacitive load power.

Transient power is given by,

$$P_{\text{T}} = C_{\text{pd}} * V_{cc}^{\,2} * f_{\text{I}} * N_{\text{SW}}$$

Where:

P_{T} = Transient Power Consumption
C_{pd} = Dynamic Power dissipation Capacitance
V_{cc} = Supply Voltage

f_{I} = Frequency of Input Signal
N_{SW} = Number of Bits Switching (for a single bit this value is 1)

Capacitive load power is given by,

$$P_{\mathrm{L}} = C_{\mathrm{L}} * V_{cc}^{2} * f_{\mathrm{O}} * N_{\mathrm{SW}}$$

Where:

P_{L} = Capacitive Load Power Consumption
C_{L} = External Load Capacitance (per output)
V_{cc} = Supply Voltage
f_{O} = Frequency of Output Signal
N_{SW} = Number of Bits Switching (for a single bit this value is 1)

If the capacitive loads are different for the different devices, the equation above can be re-written as,

$$P_{\mathrm{L}} = \sum(C_{\mathrm{Ln}} * f_{\mathrm{On}}) * V_{cc}^{2}$$

Where:

C_{Ln} = All different External Load Capacitances numbered 1 to n
f_{On} = All different frequencies of Output Signal numbered 1 to n

Therefore the dynamic power consumed is the sum of the transient and capacitive load powers and can be expressed as,

$$P_{\mathrm{D}} = P_{\mathrm{T}} + P_{\mathrm{L}}$$

$$P_{\mathrm{D}} = (C_{\mathrm{L}} * V_{cc}^{2} * f_{\mathrm{O}} * N_{\mathrm{SW}}) + (\sum(C_{\mathrm{Ln}} * f_{\mathrm{On}}) * V_{cc}^{2})$$

$$P_{\mathrm{D}} = [(C_{\mathrm{L}} * f_{\mathrm{O}} * N_{\mathrm{SW}}) + (\sum(C_{\mathrm{Ln}} * f_{\mathrm{On}})] * V_{cc}^{2}$$

Ways to reduce dynamic power [9]:

- Clock Gating
- Smaller transistors to reduce the device threshold
- Lowest suitable supply voltage
- Lowest suitable frequency of operation

The total power consumption is the sum of the static and dynamic power consumptions,

$$\boldsymbol{P_{\mathrm{tot}} = P_{\mathrm{S}} + P_{\mathrm{D}}}$$

2.2.2 Hardware Techniques of Power Optimization

2.2.2.1 Static Power Reduction Techniques

Some of the commonly used techniques for static power reduction are as shown in Figure 4, like the use of multi threshold transistors, or body biasing, or power gating to shut of parts of the circuit that are not active.

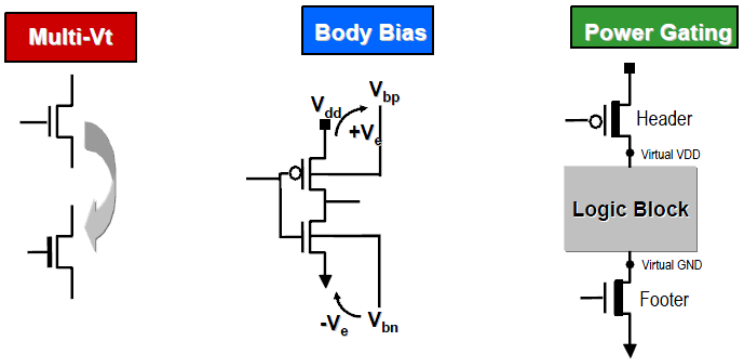

Figure 4: Techniques to Reduce Static (Leakage) Power [11]

Multiple Threshold Transistor circuits use a combination of two levels of thresholds [9], low threshold (low-Vt) transistors used for computation purposes and a high threshold (high-Vt) thransistor used as a switch in the idle mode to disconnet the power supply. As seen from Figure 5, the high-Vt transistor is connected between the the true V_{DD} and the virtual V_{DDV} which increases the impedance between the true and the virtual power rails, resulting in a higher level of power supply noise and gate delay.

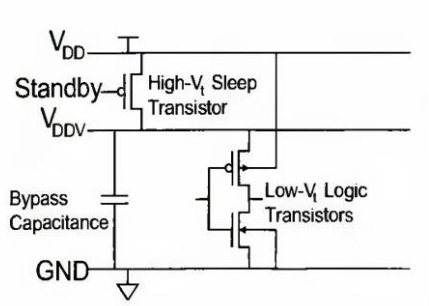

Figure 5: Multi Threshold Circuit [9]

The bypass capacitance stabalizes the the supply but when the capacitor discharges everytime V_{DDV} is disconnected, power is consumed. Since at any point, one of the computational transistors is OFF and the high-Vt sleep transistor too is off, this reults in two series OFF transistors. As per stack effect [12], the leakage through two series OFF transistors is much lower than a single OFF transistor, one being of high threshold and one being of low threshold. Thus the overall leakage current reduces, bringing in a positive change in the power performance of the system.

Body bias is another way of controlling the leakage current flowing through a transistor. A low threshold transistor can be reverse body biased to reduce the amount of leaksge current flowing through it and a high threshold transistor can be forward body biased to increase the performance during the active mode [9]. The technique usually applied is adaptive body biasing wherein one can achieve a rather uniform poerformance inspite the varations. In general the body bias voltage should be kept to less than 0.5V. Too much reverse body bias voltage leads to band-to-band-tunneling wherein there is a large amount of junction leakage and too much forward body bias voltage leads to a large amount of current through the body.

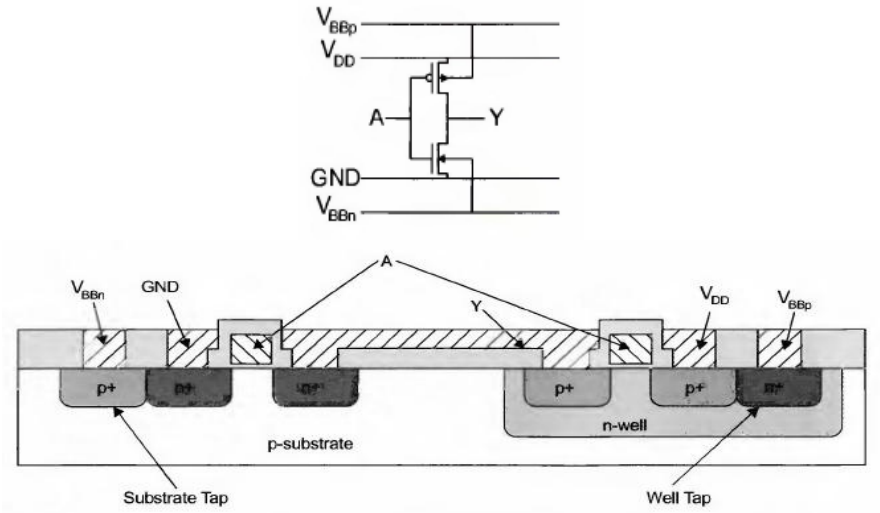

Figure 6: Body Bias [9]

As seen from Figure 6, we need additional power rails for for the various substrate and well voltages. Commonly used values for V_{BBn} is -0.4V and for V_{BBp} is 2.2V, where the power rail is of 1.8V. The n-well is at 1.8V and the p-substrate is kept at -0.4V because of V_{BBn} resulting in a reverse bias junction and hence nearly zer oleakage current flows. The n-well is at 1.8V and n-well at V_{BBp} is at 2.2V again causing a similar effect as in the previous case and hence reducing the leakage current.

Power Gating is one of the most effective ways of minimizing leakage current in a transistor. Power gating is achieved by using a suitably sized header or footer transistor, as shown in Figure 4, for a logic block that requires power gating [13]. When there is no activity for a long time, the logic will detect this and immediately apply a "sleep" signal to the gate of the header or footer. The header and footer sources act like virtual rails for power or ground. This is because they are directly connected to the main power or ground rail and if the respective transistor is activated (turned ON), then the sources of those transistors act like virtual rails. Thus on applying a sleep signal to the gate of either one of them, we turn of the power supply to the whole logic block. Once it is determined that we need to power back on the whole logic block, the sleep signal is de-asserted to restore the voltage to that logic block.

Power gating can be employed to cut off power to inactive units based on the workload of the system dynamically, this reduces both gate and sub-threshold leaksge of current, and results in a 20x to 2000x reduction in leakage with little or no cycle time

penalty [14]. Thus more power is available for scalar units and dedicated units available for intensive application perofrmance.

2.2.2.2 Dynamic Power Reduction Techniques

Some of the dynamic power reduction schemes are shown in the pie chart in Figure 7, where each section of the pie chart roughly shows the amount of power consumed by a particular entity. For example, the clock tree and latches account for almost 50% of the power consumption in a processor as per [11]. Memory accesses and I/O interactions would account for nearly 40% of the power consumption and the remaining is due to some random logic the chip has to account for. The methods for optimization are shown in the pie chart for each of the sources of power consumption, like optimization techniques for clock trees and latches are clock gating, clock tree synthesis and multiple supply voltage. Some of these will be discussed in further detail later.

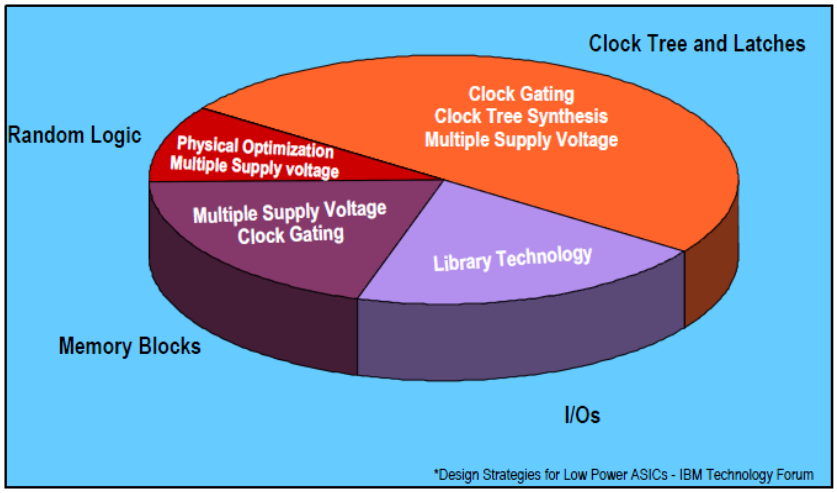

Figure 7: Techniques to Reduce Dynamic (Active) Power [11]

Clock gating is a technique that has been applied successfully in the custom ASIC domain [15]. If a portion of a circuit whose outputs do not count for, their clock and logic signal power is disabled temporarily. Some registers are clocked but their outputs are not considered, clock gating refers to activating the clocks in logic only if some work has to be done by that circuit. When the clock gaters are inserted, the delay between the blocks need to be taken into consideration to make sure no clock races occur [9]. In traditional synchronous designs the systems clock is connected to each and every clock pin of every flip-flop. Due to this setup, the power consumption has three main components [16], power consumed due to the combinatorial logic, the power consumed due to the flip-flops, and the power consumed by the clock buffer tree. Clock gating techniques not only optimizes power consumption by flip-flops, but also the power consumed by the clock buffer tree. Clock gating works by indentifying the flip-flops that share a common enable. Clock enable is used to control the select pin of the multiplexers connected to the D input of a flip-flop, or to control the clock enable pin on a flip-flop to allow the input to be clocked out to the output with a valid clock signal.

Hence if a bank of flip-flops, sharing a common enable signal, and have clock gating implemented, then the dynamic power consumption by these flip-flops will be nearly zero, as long as the enable is false.

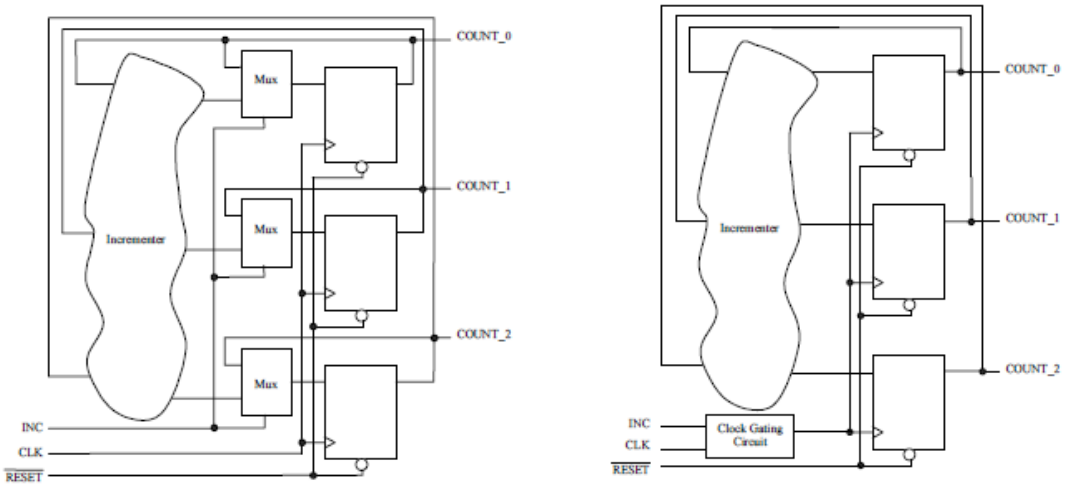

Figure 8: Clock Gating Implementation on a 3 Bit Counter [16]

A shown, the dynamic power consumption depends on the square of the supply voltage, thus lowering the supply voltage should lower the dynamic power consumed. But lowering supply voltage also lowers the performance of the system, hence to keep a tab on the lowering of supply voltage to maintain the same level of performance. Lowering V_{cc} for the entire chip will directly impact the performance of the chip; hence this is not an option. The other possibility is to lower V_{cc} at the module level, but this can be done only if the individual modules do not affect the overall performance of the IC and the gains in power reduction are limited. Finally, the V_{cc} can be lowered for each individual gate, such that the overall performance of the system has not reduced. Lowering the supply voltage for a particular device will increase the delay for that device but then this would be acceptable if the overall performance of the whole system is maintained [17]. Such paths where the delay can be increased by lowering the V_{cc} are called non critical paths.

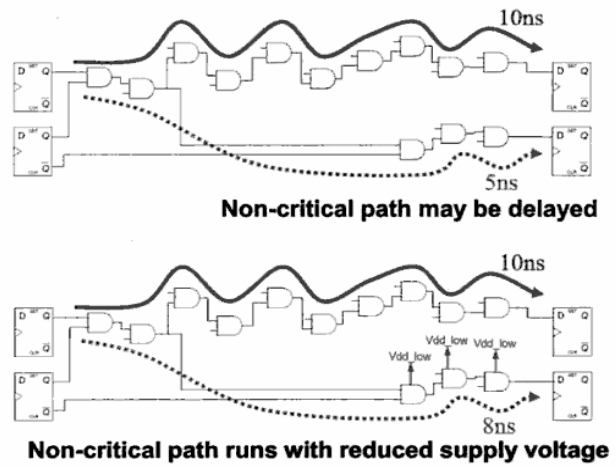

Figure 9: Multi Supply Voltage Design [17]

As seen from Figure 9, V_{dd_low} refers to the reduced voltage for the three AND gates, thus increasing the delay for each device, increasing the overall delay of the path, but since this is a non critical path the extra delay here will not affect the overall performance of the system, but it will result in a lowered overall dynamic power consumption [17]. The power reduction achieved in this method directly depends on the decrease in supply voltage from V_{dd} to V_{dd_low} and also to the number of devices to which V_{dd_low} is applied.

2.2.3 Link between Hardware & Software

Before me move to the big question of why we need to venture into the area of power optimizations at the higher levels of abstraction such as the application software level, we need to understand the flow of logic and how the software interacts with the hardware and where is it that the power consumption occurs.

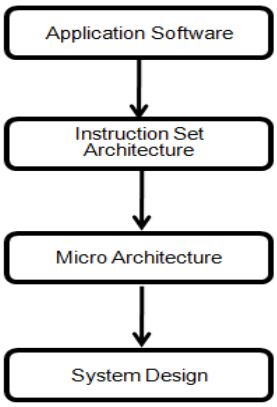

Figure 10: Subcategories of Layers

As seen in Figure 10, the flow of logic starts from the application software where the software developer has written the desired functionality to be performed. This code then gets converted to a stream of instructions specific to the instruction set architecture (ISA) [18]. The compiler and the assembler convert the source code into the object code that can be run on a processor. The ISA is that part of the processor that relates to programming and has information about the data types, instruction set, addressing modes and the registers. This information can be put together in what is known as opcodes. The layer below the ISA is the micro architecture, which is a more concrete description [18] of the system and is a set of processor design techniques used to implement the ISA. The micro architecture describes the various interconnections of all the constituent parts of the system and how they interoperate in order to implement the ISA. Some of the concepts of the micro architectural design are [3]:

- Instruction Pipelining
- Branch Predication
- Superscalar
- Out of Order Execution

- Register Renaming
- Reorder Buffers

Finally we have the system design that includes all the necessary hardware components such as interconnects, memory controllers and clock logic. Once we have the ISA and the micro architecture, the actual device needs to be fabricated onto a hardware chip and this process is often known as implementation [18]:

- Logic Implementation – blocks of the micro architecture are converted to logic equations
- Circuit Implementation – logic equations are implemented as transistor circuits
- Physical Implementation – chip with various circuits and interconnects is fabricated

2.2.4 The Need to Optimize Power at the Architectural Level

Improvements in materials technology, processor architectures and techniques have enabled transistor sizing to a mere 35nm regime, where the channels are made up of a handful of atoms [20]. This has led to more transistors in the same area as previously imagined, which means designers are allowed more functionality to be utilized by using all the transistors in a small space, thus pushing the performance beyond the previously assumed barriers. Handheld portable electronics today can integrate more and more functionality in the same amount of area as before, improving the performance trend, but also increasing the power consumption of such devices many fold. The main aim of designing such handheld electronic devices today is reducing the overall system power consumption. As seen in Figure 1, the trend of power consumption has ever been increasing as more and more functionality is being thrown into these devices, but the power requirement has stayed constant to around 1W. We are now seeing the diminishing effects of power optimizations at the lower levels of abstraction, and implementing the increased hardware on a single chip has the hit the well known “power wall” [20]. Research has shown that “the biggest low power design bottleneck is at the architectural or electronic system level (ESL), where the potential power savings are the greatest and where the methodology and tool support is weakest” [5].

The big question comes, as to why do we need to optimize power at the architectural level? Looking at Figure 11, we see that at the layout level there aren’t many choices in changing the design since the design is pretty much fixed. As we move up in the level of abstraction at the gate level, we have a little more choice and flexibility to change and optimize the design but it is still fairly constrained. At the RTL we can optimize the design by a fair amount of around 20% [21], but this is also limited since the system design and intent of what the system is doing is fixed and defined at the RTL. At the architectural level however, we see a much larger potential of optimizing the design for power consumption, almost up to 80% since there is much more flexibility, we can attack the problem in different ways, and solve it by delivering the performance the system requires in the most power efficient manner, and hence the highest degree of impact compared to any of the other levels of abstraction [21, 22]. The

power aware developer must keep in mind the amount of power that would be consumed for complex problems to get the best performance yet. It is the software that exercises the hardware and writing power aware software will be integral moving into the future.

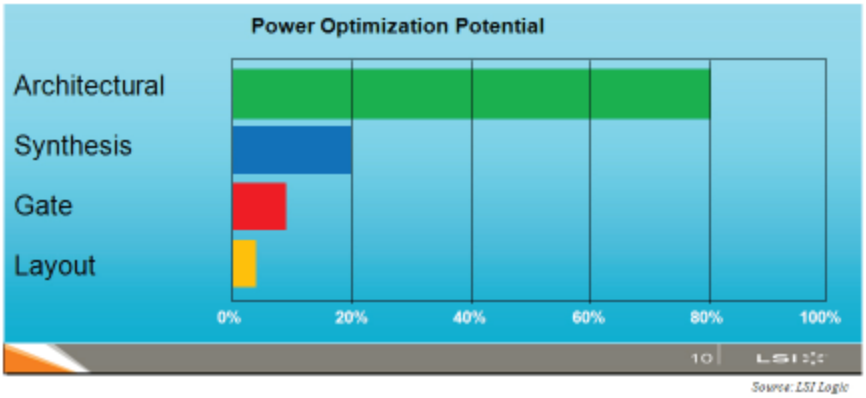

Figure 11: Power Optimization Potential at Various Levels [21]

Layout Level

Accurate simulation and estimation of power consumption can be done once the actual layout of the design is ready to be fabricated. The optimizations are done by transistor resizing and layout rearrangement at this level [19]. The possibilities for power optimization are very limited at this stage since the layout engineer can only play around with the sizes and placements of transistor blocks, but no major design changes can be done, like adjustments to the number of functional units since the design is fixed at this stage. Thus the potential for optimizations are pretty much fixed and limited as seen from Figure 11.

Gate Level

After the logic synthesis comes the gate level netlist, which can be simulated to find the power consumption fairly accurately [19], and this dynamic consumption would be based on the design activity. Designers at this point can optimize their designs by reducing the capacities the most active nodes in the design have to drive, plus optimizations of power can be done by balancing path delays, spurious transitions and retiming, with a fairly high level of accuracy but again in a very limited fashion. Numbers [21] estimate gate level optimizations to account for 10% reduction in power consumption.

Register Transfer Level

In contrast to gate level, the RT level is more abstract, power consumption can be estimated using event based simulation tools. The options to reduce dynamic power at this level are much higher when compared to the gate level. Certain parts of the design can be dynamically switched to low power mode [19] or the battery mode as it is known to improve battery life in portable electronic devices. Methods like reduced clock gating or full clock gating are applied to certain areas of the design to improve power consumption. Other methods like optimized resource sharing, optimized coding

of bus states further help in reducing the overall power consumption. In general architectural changes can be done with a higher degree of flexibility, for example changing the number of functional units to do a particular task, compromising on the overall performance, but reducing the power consumption none the less is a method to reduce the dynamic power.

Architectural Level

The fundamental advantage of power optimization at the architectural level is a result of higher degree of flexibility. A path that results in timing and power consumption at higher levels is the best path [23]. The understanding of architectural choices, software decisions and the impacts of data should be done before the RTL implementation begins. Research shows that, arguably up to 80% of the power management opportunity sits waiting to be exploited at the architecture and system level [21, 22, and 23].

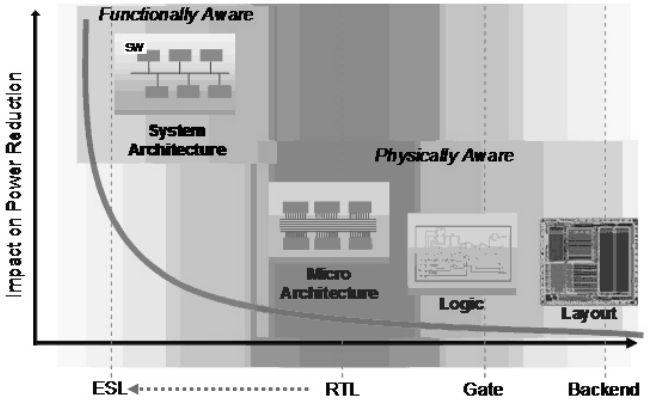

Figure 12: Known Power Optimization Techniques [23]

2.2.5 Software Techniques of Power Optimization

Power can't be optimized in isolation; we need to look at the system resources and the load profile, then we need to correlate that power with the peak power and its distribution. The first task should be to understand the workload of the system, where significant static and dynamic power reductions can be made based on the system workload. If we can understand the workload the system has to process, what work needs to be done and what should be the performance of the system, then we can work towards optimization of the overall system power consumption. If we compare the dynamic power profile [21] to the workload of the system, we can see that the dynamic power changes over time, but without any optimization, this dynamic power is going to be much higher than the actual dynamic power that is required to deliver the same level of performance. As seen in Figure 13, the dynamic power supplied to the system is always much more than the system workload and the worst case scenario is when the dynamic power is consumed even though the system workload is negligible, as depicted by the valley of the graph. There is a major dip in the workload valley, but not so much for dynamic power. If we can manage to optimize the system to tune the dynamic power profile to be as close to the workload as possible, this would result in great power optimizations. This is depicted in the Figure 13 as the optimized dynamic power profile.

Here the valley for the workload is as close to the optimized dynamic power profile, and to do this we need to understand the workload better.

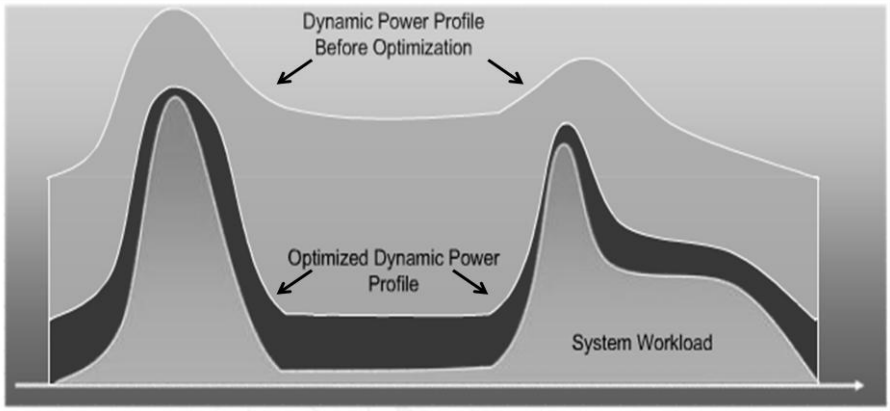

Figure 13: System Workload Dynamic Power Optimization [21]

We can do the same for static power. By better understanding the workload of the system, we can accordingly scale the clock frequencies or shut down portions of the design [21]. As seen from Figure 14, the static power profile before optimization is a pulse that stays constant irrespective of the system workload, whereas a better understanding of the workload allows us to quantize the static power in such a way that static power is consumed only when needed. The valley of the system workload is the region where maximum static power is consumed and wasted, but on tuning and optimizing the static power profile, we can truly achieve static power.

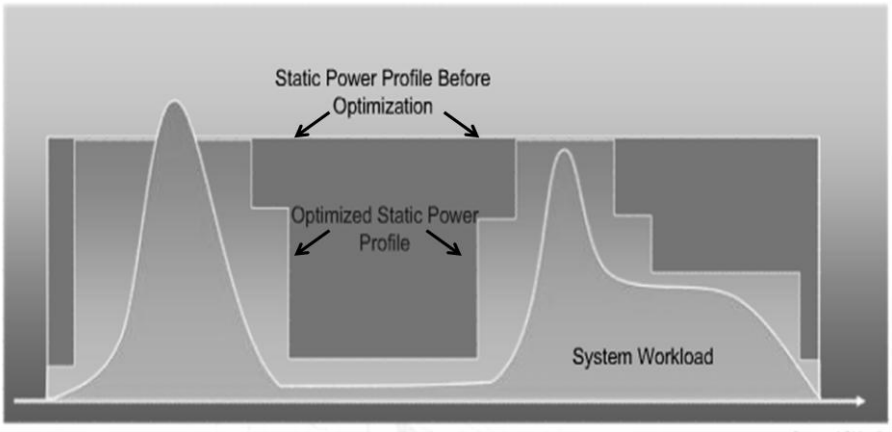

Figure 14: System Workload Static Power Optimization [21]

What we need is a power analysis framework [24] that is integrated into the development environment. It needs to include power models that can guide designers to develop software which is power aware. This framework enables early power analysis through simulations, which need to be fast and highly accurate allowing designers to analyze and make tradeoffs on the functionality to be performed. This sort of setup will provide a way of gathering the system activity information and this combined with a library of power characterized data will be able to compute the system power

consumption. Such an approach will allow designers to analyze data at a level which is typically not possible at the RTL. The simulations are run and the power dissipation over time waveform is plotted, which can be viewed either at the individual component level or the system level power consumption.

Power consumption of an individual component can be viewed as a finite state machine (FSM) [24]. The possibilities of various states could be either ON, OFF or an operational state. Figure 15 shows a power model of a CPU core with three states, ON, OFF and IDLE. Each state is allocated a leakage (as in the case of OFF and IDLE state) and a dynamic base power (as in the ON state). The base power (leakage or dynamic depending on which state you are in) is consumed by the individual component as long as it is in that state. Transitions and operations within a state account for the power consumption data [24]. For example, the ON state has two special operations, MULT and DIV, which have much higher power consumption than the base power of the ON state, hence they are explicitly modelled.

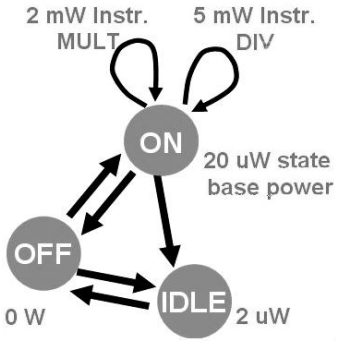

Figure 15: Assumed Power Model of a CPU Core [24]

The power values in the library of power characterized data can be made voltage and frequency dependent. For instance the baseline power of any state can be made dependent on the frequency and supply voltage of the processor to enable variable voltage or dynamic voltage and frequency scaling (DVFS) [24]. We also need to take into account the granularity of the power characterized data, because fine grained power models would require detailed trace information which would slow down the overall system analysis simulations. On the other hand a coarse grained power model would require only the baseline power data of the various states, hence making the overall system analysis simulations much faster, but with tradeoffs on the accuracy of performance and power estimation. An optimal solution would be to have a mixed strategy where both fine grained and coarse grained models are used; fine grained models can be applied to the high power consuming blocks whereas the coarse grained models can be applied to the less power hungry components of the system, thus giving accurate, yet speedy power estimations.

Study 1

This concept of power models is used as a background in the study [32] done at the University of California, where a functional level power estimation methodology for predicting the power dissipated by embedded software was developed. For a given

processor a "power data bank" [32] was built which, like a built in library function, contained the power information for the basic instructions. In order to estimate the power consumed by the embedded software, a power profile was first extracted with the help of tracing tools and then once this information was contained in the power data bank, the total energy consumption and execution time were evaluated based on this power data bank. It is predicted that the chip manufactures can supply the users with these power data banks in order to estimate the power consumption using simulators at earlier stages with an average error of 3%.

In order to implement the above mentioned technique, the project was divided into four main tasks:

- Power Measurements
- "Power Data Bank" Construction
- Function Level Power Modeling
- Function Level Power Estimation

It was observed that instruction level power models [33] depend on the power consumed by each instruction and the various combinations of instructions. A standard embedded software package contains thousands of such instructions and combinations. Through experiments [32] it has been observed that the majority of the power consumed is by the built in library functions, user defined functions, the main function body and others (e.g. hardware reset program). This led to the equation for power given by energy over time.

$$P = \frac{E}{T} = \frac{\sum_i E_{BF_i} + \sum_j E_{UF_j} + E_{main} + E_{others}}{\sum_i T_{BF_i} + \sum_j T_{UF_j} + T_{main} + T_{others}}$$

Since the user defined code will be different depending on the software application, it was proposed to find the energy consumption of the built in library functions and the individual instructions. The energy consumption and execution time for the built in library functions and basic instructions is known depending on the architecture being used. Also the power data bank should contain multiple entries for the same function or instruction to account for the cache misses and pipeline stalls that could possibly occur.

Once the power data bank is updated with all the possible combinations, the whole program is simulated using a power simulator either at the transistor level or the register transfer level. Lower the level, higher will be the accuracy of power estimations. A vendor can choose the best level possible depending on the desired accuracy and criticality of the application and at the same time not part with the information regarding the architecture by the use of the power data banks.

Study 2

According to the study done by LSI Logic, the power reduction capability at the ESL is almost up to 80% if it is exploited properly [31, 21, 7]. ESL application aware

designs make this reduction possible in various ways such as turning off idle parts of the chip, dynamic voltage frequency scaling, It is important to study the effects of the software on the hardware and power consumption since most of the opportunities to impact the power consumption are forgone once the architecture is fixed and the RTL is implemented [31]. Changing any part of the architecture will result in high costs and time. Hence the greatest opportunity lies at the ESL where the processor core, memory, peripherals and busses are defined and the software is integrated. According to the study done by Yossi Veller & Shabtay Matalon at Mentor Graphics [31] shows, there are four ways we can achieve this 80% power reduction through the ESL as discussed below.

Optimize Power Based On System Workload over Time

The best way to gain maximum optimization capabilities would be to correlate the power consumed by the workload on the system. These models are then capable of simulating the workload patterns and scenarios very quickly [31], thus enabling the designers to simulate system use cases two or three times faster than the RTL. It is only then that the great power optimization opportunities are identified with speedy computations.

Address Tradeoffs between Hardware & Software

Hardware functionalities often consume lesser power because they are more efficient than running the same functionality in software [31]. Transaction level models allow for designers to measure different power consumptions and evaluate the various hardware and software configurations to determine the best way in which the functionality can be partitioned with minimum power usage.

Optimize Area, Performance & Power Configurations

For the best power versus performance optimizations, the simulations for the system workload should be able to run at least two or three orders in magnitude faster than the RTL simulations. With quick simulation results, the designers can easily evaluate various design configurations and choose the best option according to the application.

Application Software Tuning

When certain functionality is in software, the power profiles will vary due to various reasons. This software can be tuned to consume the minimum power by comparing the various power profiles that were obtained by the simulations and choosing the best suited profile.

Study 3

A study [34] shows that the component of power consumption that occurs due a program running is the most difficult to simulate. This study uses the concept of instruction level power modeling to compute a particular piece of codes' energy consumption by summing the total energies of each individual instruction. This process is quick since the energies are summed to produce the final energy consumption of the program. But certain factors such as pipelining, memory hierarchy, constant switching due to data and different instruction encoding cause this energy consumption per

instruction to vary considerably [34]. This is a lot in line with what is intended in this project, to determine the individual instructions' power consumption and then try and find a correlation between the various instructions and the power consumed by them in a program.

The power consumed due to switching is considered an overhead over the base power, and this component is important since it will vary from instruction to instruction. Experiments done [34] concluded that this dynamic switching overhead was around the 15mA mark in most of the cases that were examined. This experiment was performed using a lab ammeter to provide stable readings over a period of time and the energy per instruction was used for analysis rather than the average current per instruction. In order to increase the switching activity, no-op's were added in between other instructions to generate the switching activity a normal program would otherwise encounter. Also instead of using a loop with enough instructions to minimize the jump at the end of the loop, a boot up case [34] that is essentially a shell of the main program loop was created. This avoided any potential errors in the measurements.

For the results a WIMS controller was used and the measurements were done using the HP82000 D200 digital tester. A table showing the results from this experimentation is shown below and produced some interesting results which can be used as a tool for comparing the outcomes of this project as an optional objective.

Instruction Group	Energy (nJ)	Time (ns)
add-sub	0.2403	10
shift	0.1950	10
boolean	0.2127	10
compare	0.2082	10
multiply	2.7702	180
divide	2.7160	180
copy	0.2127	10
bit	0.6137	20
load abs	0.5249	20
load rel	0.3661	10
store abs	0.4427	20
store rel	0.3070	10

Instruction Group	Energy (nJ)	Time (ns)
win swap	0.1832	10
load imm	0.1961	10
branch-nt	0.1720	10
branch-t	0.5741	30
jmp abs	0.5372	30
jmp rel	0.4020	20
jmp abs sub	0.5658	30
jmp rel sub	0.3527	20
return	0.3700	20
swi	0.5585	10
noop	0.1931	10

Figure 16: Measured Energy for Each WIMS Instruction [34]

The various studies done are somewhat related to the objectives of this project in many ways. Thus understanding what others have done would be important in understanding the various ways in which this project can be carried out and also form a basis for comparison of results. These studies show that the overall aim of power aware software and optimizations at the ESL is the direction which needs further investigations and research, because that is where the key potential lies.

2.3 Summary

This chapter outlined the reasons for power consumption on a processor and its various components, static and dynamic power. The component that is of interest to us in this project is the dynamic power which is due to dynamic workloads and programs. We also looked into the current methods of power optimization at the lower layers of abstraction. Recent studies show that the potential to optimize power at the software level is enormous, but the techniques are novel. This chapter also discussed some research done in the area of power aware software design with the aid of a few studies and some interesting results obtained from those studies. The background researched is in line with the aim of this project, to investigate the consumption of power on the hardware due to execution of the software.

Chapter 3: Project Design

3.1 Introduction

The main aim of this project is to investigate the power consumption which is due to the execution of instructions at the software level and to understand how different workload patterns affect the consumption of power. The workload patterns or the system workload, as it can be described, is a set of benchmark programs that contain certain instructions or a group of instructions which can be arranged in a straight forward manner or in different clever ways to understand the various scenarios for power consumption. In this chapter we discuss the project design, including details of the technical problems faced and their solutions. To achieve the project goals, the task was divided into four main objectives:

Objective 1: Design framework

Objective 2: Design benchmark programs

Objective 3: Measure power consumption

Objective 4: Correlation of power consumed

3.2 Objective 1: Design Framework

The framework was designed in software using the XMOS Development Environment (XDE) and XC (which is an extension of C used by XMOS for their parallel architectures) which enabled the use of timers, ports, channels and parallel programming. The XDE is an eclipse based tool with 'Debug' and 'Timing' capabilities built into it. Before we move on to the main part of the software development, it would be interesting to know about the capabilities of the XC language and some interesting pointers about the XCore architecture that were used as a part of the hardware setup.

The XC language is an imperative programming language whose basic framework is designed around the C language [28] but with a few extensions to support:

- Concurrency – this is one concept along with the use of channel communications that was widely used in the scope of this project, the reason for which will be understood better on further reading
- Clocked Input and Output – this concept is used to ensure timely behaviour of the data that is sampled and driven on specific edges of the clock, thus allowing the processor to control, on which edges each input and output operation occurs
- Port Buffering – this concept is mainly used to improve the performance of the functions that use the concept of clocked I/O on ports

A note on the XCore architecture, the device used was the XS1-L1 device whose processor has a multi threaded capability for concurrent execution and a high level programming language (XC) design flow. Figure 17 shows the internal architecture of the XS1-L1 device that was used in this project provided by XMOS. Some of the key features that are worth noting are:

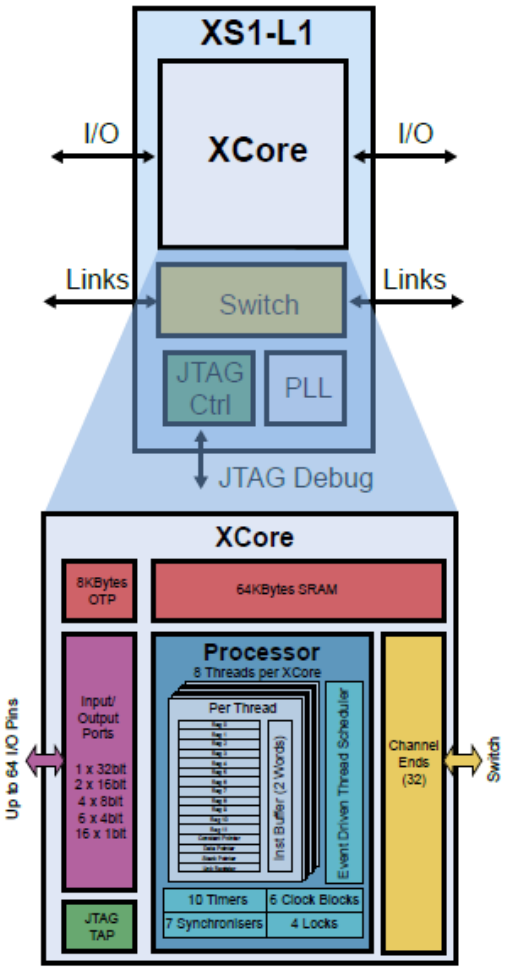

Figure 17: Internal Architecture of the XS1-L1 Device [35]

- Each XCore has
  - ✓ 8 threads for parallel processing
  - ✓ 32 bit processor
  - ✓ 64KBytes SRAM memory
  - ✓ I/O ports for communicating with external components
  - ✓ 32 channel ends for inter thread communications
  - ✓ 10 timers used for synchronisation
- High performance switch used for synchronous communication between threads on different XCores

The XS1 integrates a number of XCore processors on a single chip [25], and are general purpose in the sense they can work on the C programming language and also

support parallel processing. There also exists a high performance switch that can be used to communicate between the XCores for increasing the performance. These interconnects allow communication between all the XCores on the chip or a system if there is more than one chip. Each of these XCores has the capability to execute a number of threads concurrently and include a set of registers for each thread, a scheduler to dynamically select a thread for execution, a set of synchronisers for synchronisation and channels for communication. A set of ports for input output, a set of timers for real time execution.

Some of the main features of the instruction set are that the instructions are short (over 80% of the instructions are 16 bit) [25], making it efficient access to stack and other compiler allocated regions, memory is byte addressable, the input and output instructions allow for fast access within an XCore and also between XCores.

The concurrency of the threads can be used to parallel implement input output transaction along with the software application, allows communication in parallel with processing [25], and to some extent latency hiding, The instruction set allows for, event driven communications, streamed and synchronized communication between threads in any XCore, and also allows for idling processor with clocks disabled to save power.

As for the instruction issue and execution, a short pipeline is implemented to increase the responsiveness of the instruction. No forwarding or speculative instruction issue is implemented. As far as scheduling is concerned, it is guaranteed that for n threads executing, each will get at least 1/n processor cycles, and it can be said that one thread cycle is equivalent to n processor cycles. The pipeline has a unique memory access stage, which is available to all the instructions and the rule is that if any instruction requires any data for execution, it is fetched during the memory access stage. Brach instructions fetch their targets during this stage.

The hardware was setup as shown in Figure 18 below. The master was connected through two ADC's reading the voltages (V1 and V2) across the two ends of the 1Ω resistor, one end corresponding to the variable voltage source and the other end corresponding to the slave (load). Thus if there was increased activity at the load end, it would result in a decreased voltage across the variable voltage source since more current was being drawn through the 1Ω resistor. Also since we would know the voltage difference V1 ~ V2, and since we know the resistance of the resistor (1Ω), we can calculate the current flowing through the 1Ω resistor. Thus we can calculate the power that was consumed due the load being executed.

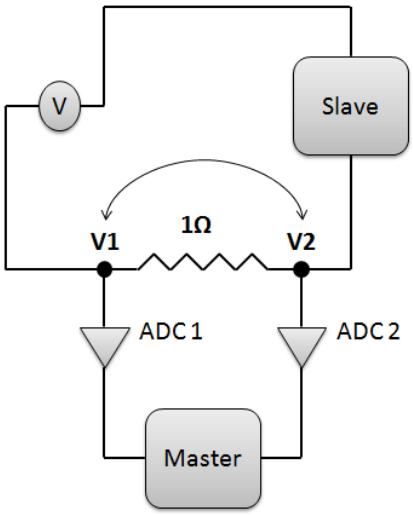

Figure 18: Hardware Setup & Interconnects

Now that we have an idea about the design environment and the hardware used, we can move on to the software setup developed for the framework. As seen in Figure 19 below, the whole software setup was divided into two main functions, one for measuring the power consumption and the other for executing the load. The XDE was used for the software development and testing activities. The initial phase of this project was dedicated to understanding the working of the XDE and studying the XC programming language which was used for the software framework.

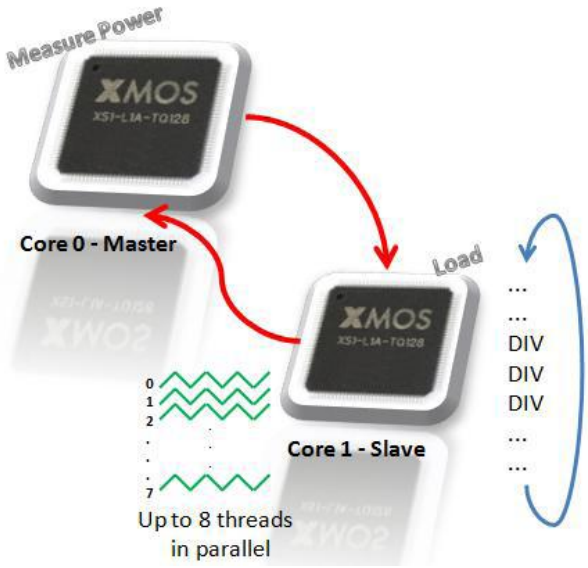

Figure 19: Software Setup & Interconnects

Figure 20 below shows the layout of the various functions that needed to be developed for the framework. Since two modules were being used, one for measuring the power consumed, and the other for executing the load that would consume power, there were two branches from the main function of the program. One branch represents

execution on Core 0 (Master), and the second branch represents execution on Core 1 (Slave). The master module in turn repeatedly calls 'measure_task' function that was actually responsible for reading the outputs from the two ADC's as shown in Figure 18. The slave module can execute from one, up to eight threads in parallel. Each thread could execute the same workload or different ones depending on the experiment that was being carried out. Since the tool and the programming language were used for the first time, one of the main challenges was completing the framework on time. But this was accomplished by a lot of research done on the tool and the company provided resource materials. The overall time taken for this part of the project was around a month, which included learning the tool and language, extensive research on the programming constructs and constant interactions with the supervisor. Each of the functions developed are explained in detail below.

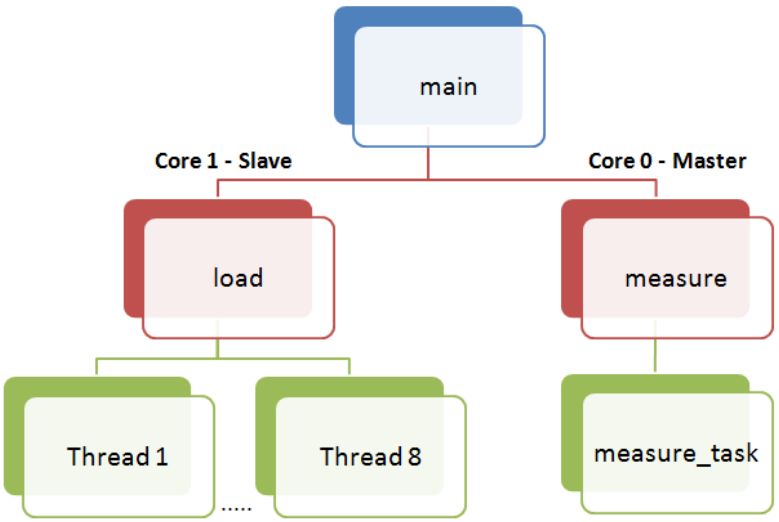

Figure 20: Arrangement of Software Functions

measure ()

In the measure function, the control would go into a while loop that would continuously execute until it receives a signal from the load (slave module) signalling that the load has reached the end of its execution and the master module can now stop taking any further measurements. But till this point, as long as the master module is executing the while loop, it is taking continuous measurements of the voltages (V1~V2) as explained above and recording them into an array at regular intervals. The major challenge here was to figure out how the master and slave module would communicate with each other and signal each other the end of certain events. This was accomplished by the use of channel communications which provide synchronous point to point connections between pairs of threads over which data can be communicated [28], and in this case between a thread executing on the master module and a thread executing on the slave module, and the data used was a control signal indicating the start or stop of the program execution. Each thread must have a channel end declared which would identify one end of the channel.

Another challenge encountered here was that, the interval at which power measurements were to be taken had to be large enough in order to execute the code that was necessary to read the data output from the two ADC's. This time interval, which is defined as 'DELAY_ITERATION' in the code was calculated by a trial and error basis until it was found that the loop time was large enough to accommodate the measurement function. This was a rather crude method for deducing the minimum time period, but due to lack of time, calculations were not the best approach. At the end of the timer expiring for every loop iteration, a variable would be updated with the new timer value indicating when the next loop time would expire and at the same point call the 'measure_task' function.

The 'measure_task' function measures signals on the control board and was provided by XMOS for measuring the outputs of the two ADC's and hence the voltage difference (V1~V2) across the 1Ω resistor. This function initializes the IIC port and setups the two ADC's to sample all its channels, and then in a loop, gathers voltage data from the two ADC's, which is then scaled to mV and saved into an array. The values from these arrays are later printed out onto the console after the execution of the load program. A big challenge here was to integrate the framework designed so far and the 'measure_task' function that was provided by XMOS. Initially, the simulator was used to gather the timing information of the various functions, especially the timing information of the 'measure_task' function. Since the simulator was used, no actual voltages were being measured, and hence the 'measure_task' function simply incremented a counter and printed out its value for every iteration on the console, allowing for any changes to be made with the framework design. Finally, once the framework was working fine with the simulator, the hardware was connected and the same 'measure_task' function would now gather voltage information from the ADC's and store them in an array.

Yet another challenge under this section of the framework was trying to figure out, when to stop the power measurements and how. The obvious answer for when was, exactly when the load has finished execution of the benchmark workload programs. This landmark was indicated to the master module, which is doing the power measurements, via the control channel that was established between the master and the slave. To get a better understanding of the flow of control, refer to Figure 21. The start of measurements was indicated by the start of the execution of the load on the slave (indicated by the green dots in Figure 21), and the end of measurements was indicated by the end of execution of the load on the slave (this time indicated by the red dots in Figure 21). Thus channels were used very efficiently for synchronous communication between the two devices. Once the control channel would mark the end of execution of the load on the slave (red dots), the master would update a variable which was the condition for executing the while loop on the master in the measure function. To be more precise here, the condition used was:

while (!i)

Here the variable 'i' was initially set to 0, and once the control channel marked the end of execution of the load, the channel value would change, and correspondingly change the value of variable 'i' to 1. The moment that happens, the control would exit the while

loop and stop the measurements, since the power measurements were being done in the while loop. The details of this function are provided in Appendix A.

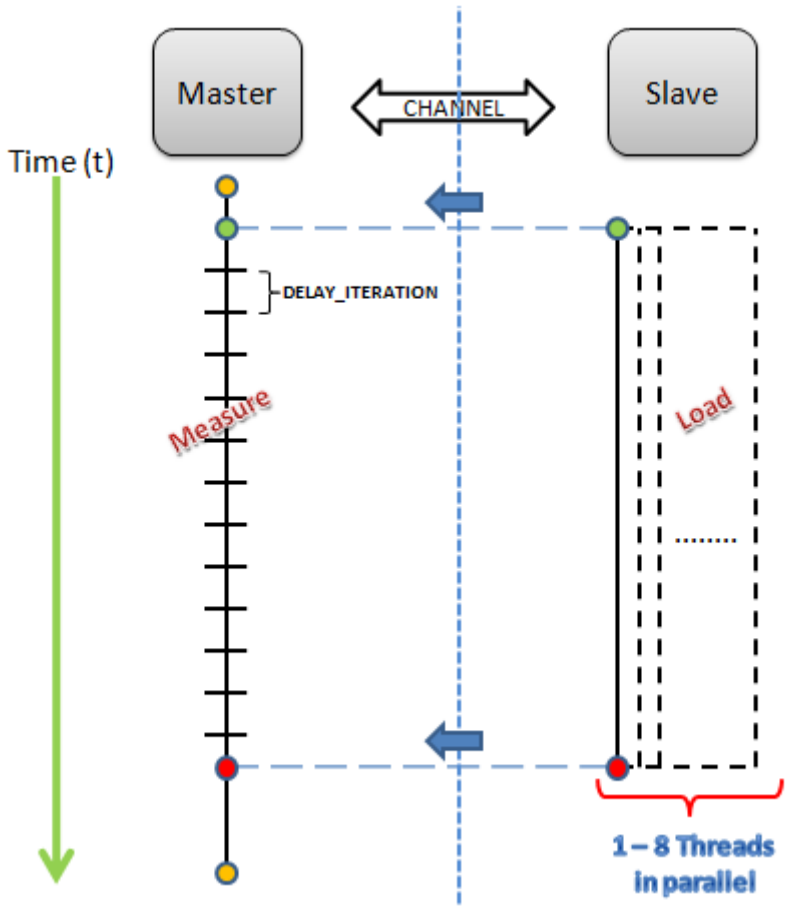

Figure 21: Flow of Control of Software Framework

load ()

The load function was straight forward in a way that it could call, from one, up to eight threads in parallel doing a certain piece of work. Before the start of execution of the load, this function would send a signal over the channel indicating the start of execution of the load, and towards the end of execution of the load, this function would again send another signal over the channel, this time indicating the end of execution of the load. This is done in order for the measure function to know precisely at what instants of time should the power measurements begin and at what instants should the power measurements end. Also, each thread could call the same workload, or different workloads, depending on the experiments being carried out. The number of threads that had to be executed simultaneously was done using the 'par' statement [28], which executed all the statements within its braces concurrently as multiple threads. The exact way in which this function is coded is shown in Appendix A.

```
par
{
        Statement 1;  /* Thread 1 */
        Statement 2;  /* Thread 2 */
        Statement 3;  /* Thread 3 */
        ....
        ....
}
```

The above example would run three threads concurrently, one thread corresponding to each of the statements. This would also ensure that at the closing brace '}' the parent function waits for all the statements to complete execution before moving on. These 'par' statements can be used anywhere in the program and can have a maximum of eight threads executed concurrently. Hence in the experiments carried out, the power consumption due to the workload is measured for different number of threads at a time, with the aim of understanding how threads affect the power consumption of the system.

3.3 Objective 2: Design Benchmark Programs

The design of the various benchmark programs (workload) was divided into three main categories, each of which is explained in detail in the following paragraphs.

Part 1: Programs with basic instructions being executed

Part 2: Programs with various combinations of instructions being executed together

Part 3: Programs with complex instructions being executed

Each category of benchmark programs were designed in an incremental order, the first being the basic set of instructions being profiled individually to make a sort of baseline for some of the most common type of instructions (Part 1). The next set of benchmark programs were combinations of the basic instructions grouped together to understand how the architecture reacts to this type of a workload (Part 2). The final set of benchmark programs were intended to understand the energy consumption of the processor better (Part 3).

A challenge here was to design programs in such a way that the power measurements were done for exactly the instruction under investigation, i.e. if the investigation aimed at studying the effects of the ADD instruction on the power consumed, then other additional statements should be ignored. As in the case of adding two numbers (c=a+b) using C programming language, this would imply, load the value of variable 'a' from memory, then load the value of 'b' from memory, add the two values, store the result back into memory. This simple addition in a high level language would introduce many other overheads and instructions whose power consumption is not of interest at the moment. To overcome this challenge, a solution was proposed to use linear assembly, i.e. using a combination of C and assembly in the same program. Hence all of the above categories of benchmark programs were designed using a combination of C, XC, and assembly languages.

```
for (i = 0; i < 500000; i++)
{
        __asm__ ("add r2, r0, r1");
        ..................
        ..................
        ..................
}
```

One way of accomplishing the above task is by using a loop shown above. The values of registers r0 and r1 can be initialized before entering the 'for' loop. The number of assembly instructions inside the loop should be large enough to nullify the affects of the branch that would occur towards the end of the loop [34]. Also the number of loop iterations is large enough to ensure that enough number of readings can be taken by the measure function described earlier. Such a scheme would ensure that very precise and accurate power measurement readings were being taken corresponding specifically to the instruction under investigation.

3.3.1 Part 1

The first set of benchmark programs aimed at basically executing instructions like the addition, subtraction, multiplication, division, no-operation, load and store. This helped in forming a baseline or a power profile that could be used to understand the working of the architecture and how various instructions affects the power consumption. This procedure took around a week to complete, which included designing the programs and integrating them into the software framework already developed. The method used in designing these programs was as explained earlier, by using a large loop with some assembly code running inside the loop.

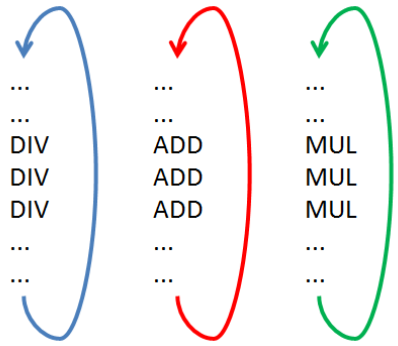

Figure 22: Design of Benchmark Programs (Part 1)

3.3.2 Part 2

The second set of benchmark programs aimed at establishing a link between the combinations of instructions such as combinations of ADD's and MUL's in order to investigate the power consumption in such scenarios. The aim of doing this is because in our day to day lives, the programs that we write are never one set of instructions in a row, but in fact are a combination of various instructions. Running the same instruction

repeatedly does not model the typical program execution as the same instruction generates very little switching [34]. Hence understanding how these instructions working in conjunction with each other and the power consumed by them in combinations would be an interesting scenario. None the less, executing the first set of benchmark programs was important from this study point of view in order to establish a baseline power profile of the basic instructions. The design of these programs, including execution took around a week to complete. The combinations were grouped in different ways. One way was to group them alternately 1 ADD and then 1 MUL in a very large loop. Another combination used was all ADD's first and then all MUL's. Yet another combination was a group of ADD's and then a group of MUL's. These are summarized in Figure 23 below.

Combination	Repeated	Loop counter
1 ADD and 1 MUL	20	500,000
20 ADD and 20 MUL	1	500,000
20 MUL and 20 ADD	1	500,000
10 ADD and 10 MUL	2	500,000
10 MUL and 10 ADD	2	500,000

Figure 23: Various Combinations Used in Part 2 Experiments

3.3.3 Part 3

The third set of benchmark programs aimed at finding out the power consumption, of complex instructions. In CISC (Complex Instruction Set Computers) architectures, we usually find instructions that do complex computations of two or more operations in a single instruction. Here it would be interesting to see how these complex instructions affect power consumption compared to their component basic instructions. For example, a long multiply (LMUL) is equivalent to using a single multiplication (MUL) and two additions (ADD) combined together.

1 LMUL = 1 MUL + 2 ADD

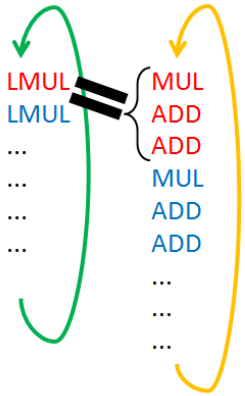

Figure 24: Design of Benchmark Programs (Part 3)

The method used for designing these programs was again the same as the one described earlier, wherein a combination of C and assembly languages was used. The loop content, i.e. the number of instructions in a loop, was large enough to nullify the effects of the branch at the end of the loop and the loop counter was large enough to ensure multiple readings can be gathered. The design of this part of benchmark programs again took around a week to implement and experiment with. A few sample benchmark programs are shown in Appendix B.

3.4 Objective 3: Measure Power Consumption

To understand better, the way in which the power consumed is actually measured, refer to Figure 21 above, which shows the flow of control between the master and slave modules. The figure can be considered to have a timeline where the time is increasing along the negative y-axis as shown by the time axis. There is a channel for synchronous communication between the master and slave modules which makes sure that the starting point for measuring the power consumption is exactly matching up with the starting off the benchmark program execution indicated by the synchronised green dots in the figure below. The slave module can execute from one up to eight threads in parallel. This is done in order to understand better the effects of parallel architectures on the power consumption and the performance of the overall system.

Once the program begins execution, readings for the voltages V1 and V2 are taken by the master module at regular intervals as shown in the figure above. The minimum value for the time interval was decided based on trial and error and had to ensure that this time was large enough to accommodate the execution of the 'measure_task' function that reads the values of the voltages V1 and V2 from the two ADC's. This process goes on until the load program has finished execution. To make sure a good number of readings are taken, the benchmark program is run in a long loop with multiple iterations. Towards the end of the benchmark program a signal is again conveyed over the channel to the master module indicating that the master can now stop taking further readings and can terminate. This is indicated by the synchronised red dots in the Figure 21 above.

The data gathering task was quite manual in the sense, once the data was printed out onto the console of the XDE (XMOS Development Environment), the data was copied into an excel workbook. The data was first gathered for one thread, followed by two and so on. Selecting the number of threads that were to be executed was done by simply commenting the other threads.

```
par
{
        Statement 1;          /* Thread 1 */
        Statement 2;          /* Thread 2 */
        // Statement 3;       /* Thread 3 */
        ....
        ....
}
```

Thus if all but two threads are commented as shown above, only two threads will be allocated on the slave module that executes the load. Each of the threads could execute the same function or different ones depending on the type of experiment being carried out. But in general, choosing the number of threads that had to be executed was done manually as explained above.

The method for gathering data, being quite crude, can be further improved with an automated system in place to extract the values stored in the array. A database management system can be put in place to handle the various benchmark programs and versions, since every iteration resulted in some modifications to the benchmark programs.

3.5 Objective 4: Correlation of Power Consumed

This step basically involved correlating the power consumption measured and the various instructions. This section is explained in detail in Chapter 4, the 'Results and Analysis' part of this dissertation. In general, the data that was collected and saved into the excel workbook was compared. The comparison involved:

- The instruction or combinations of instructions executed
- The power consumed
- The number of threads that were active
- The time taken for execution
- The operands used in some cases

Graphs were plotted for the power consumed by various instructions versus the number of threads that were active for each instruction separately. Also in the case of Part 3 benchmark programs, the time taken was given particular importance in analysis.

3.6 Summary

Since the goals of this project were divided into multiple objectives, each objective could be considered as a task leading to the success of this project. In this chapter we discussed the way in which the work was done to accomplish all the objectives, including the limitations and benefits of the various approaches. The various objectives were divided in a way which leads to the incremental progress of the project, starting from the design of the framework, then design of the benchmark programs, measuring the power consumption and finally the analysis of the data gathered in the previous step.

The benchmark programs were divided into three parts, each part handling a particular aspect of the project. The method for measuring the power consumed and the working of the framework are also explained in detail. Each part under the benchmark

programs has lead to various conclusions, some related to the power consumption, some generic conclusions.

Chapter 4: Results and Analysis

4.1 Introduction

In the previous chapter, in the 'Design Benchmark Programs' section, three different categories of workload patterns were introduced,

Part 1: Programs with basic instructions being executed

Part 2: Programs with various combinations of instructions being executed together

Part 3: Programs with complex instructions being executed

The results from each of these parts will be individually discussed in this chapter in the sections to follow.

4.2 Part 1: Results and Analysis

In this part, the basic instructions' power profiling was done in order to establish a base that can further be used in analyzing more complex workload patterns. The first thing done here is the power profiling followed by some interesting results and their analysis based on the power profile gathered and some background knowledge on the processor architecture.

Power Profile

The first step to power profiling was to get a set of readings corresponding to the basic instructions, comparing them and understanding why they consume the amount of power they consume. The best way to do this would be to first collect the data for the various basic instructions; this data collected would be in mV, from which the power consumed by the slave module can be calculated and plotted in the form of a graph.

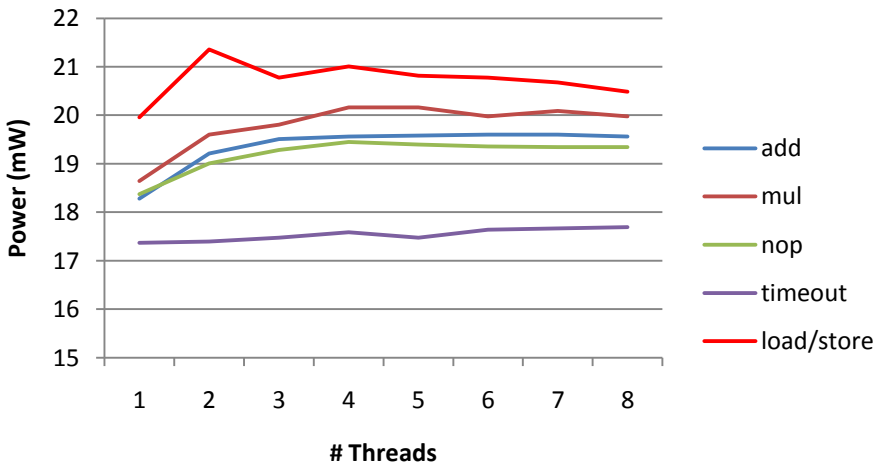

Figure 25: Power Profile Graph for Various Instructions

The graph of Figure 25 above shows the power consumption of the various instructions plotted on the same graph for comparison. The plot is the power consumed by the slave module while a particular instruction was being executed versus the number of threads being used at any instant of time. Each reading for a particular thread corresponds to the average reading obtained over a period of time. For example in the graph of Figure 26 shown below, while executing the addition instruction, each point plotted for a particular number of threads, the readings were taken over a long period of time and an average of all these readings were evened out and plotted on the graph.

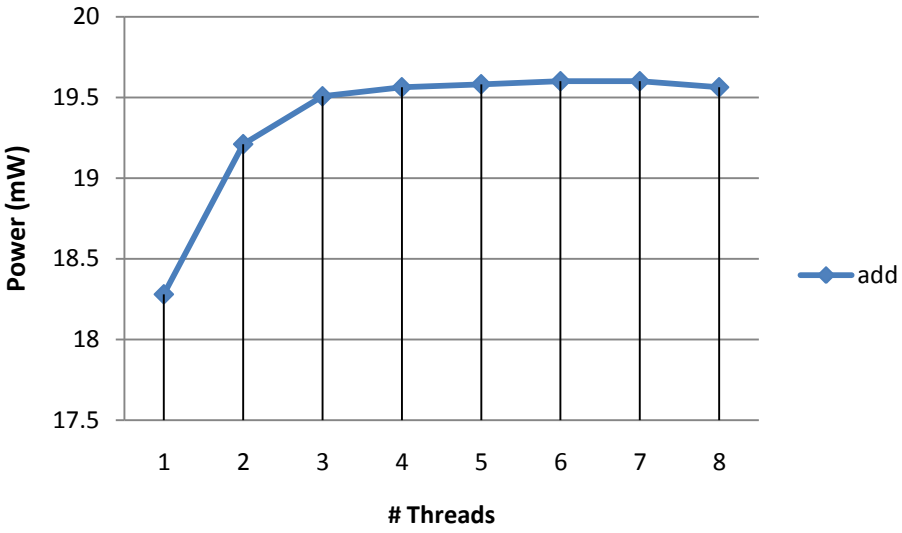

Figure 26: Power Consumption Graph for ADD Instruction

Thus as seen in the graph of Figure 26 above, the power consumed by the addition instruction for one thread was first collected and calculated in terms of power (in mW) and then an average of multiple readings was averaged out to get just one point on the graph corresponding to the power consumption by an 'ADD' instruction for execution on a single thread. Then the next set of data was collected to get the next point on the graph corresponding to the power consumption by an 'ADD' instruction for execution on two threads. The same is done for three to eight threads with the execution of the 'ADD' instruction.

Threads	Voltage (mV) (V1~V2 across 1Ω resistor)	Power (mW)
1	135.2000	18.27904
2	138.6000	19.20996
3	139.6667	19.50678
4	139.8667	19.56268
5	139.9333	19.58134
6	140.0000	19.60000
7	140.0000	19.60000
8	139.8667	19.56268

Figure 27: Table Showing Power Consumption Values for ADD Instruction

The table of Figure 27 above shows the form in which the data was collected, the number of threads along with the voltage difference V1~V2 across the 1Ω resistor. From the voltage difference readings, the power consumed can be established:

Voltage => V = (V1 ~ V2) mV

Resistance => R = 1Ω

Current => I = V/R = (V1 ~ V2) mA => since resistance is 1Ω

Power => P = V*I = (V1~V2) * (V1~V2) = $(V1{\sim}V2)^2$ mW

The graph of Figure 26 above suggests that the power consumption increases as the number of threads used increases up to four threads and beyond four threads, the power consumption becomes a plateau and flattens out. Beyond four threads, there may be a slight variation in the power levels, but in general it flattens. This happens for most of the instructions as seen in the graph of Figure 25 above. The reason for this is the architecture of the pipeline for the XCore devices.

The processor is implemented using a short pipeline to maximize the responsiveness to external stimuli and is structured to provide deterministic execution of multiple threads. The pipeline does not support forwarding between its various stages, branch prediction or speculative instruction issue making it a rather simple design. A set of n threads can be considered to be a set of n virtual processors, each with a clock rate 1/n that of the processor clock, an exception being when the number of threads allocated is less than the pipeline depth. The pipeline depth on the XS1-L1 devices is four stages deep and hence if up to four threads are allocated, then 1/4th of the processor cycles are allocated to each of the threads, but as the number of threads being allocated increases beyond four, then each thread is allocated 1/n processor cycles, n being the number of allocated threads (greater than four). Since for a single thread, only one pipeline stage will be active per processor cycle, the power consumption is the minimum, but as the number of threads being allocated increases to four, all the four pipeline stages are active per processor cycle, thereby increasing the power consumption from one to four threads being active.

1 Thread active	Pipeline Stages			
	Stage 1	Stage 2	Stage 3	Stage 4
1 processor cycle	Thread 1			
		Thread 1		
			Thread 1	
				Thread 1
1 processor cycle	Thread 1			
		Thread 1		
			Thread 1	
				Thread 1

Figure 28: Execution Cycle for 1 Thread

As seen from the Figures 28 - 32, for one thread being active, just one quarter of the processor cycle is active, for two threads being active, half of the processor cycle is active, for three threads being active, 3/4 of the processor cycle is active and finally for four threads, the full processor cycle is active. For five threads and beyond, again the full processor cycle is active, but the only difference here (compared to four threads being active) is that, the amount of time spent on each thread when four threads are active is maximum when compared to the amount of time when more than four threads are active, in which case the time spent on each thread per processor cycle reduces correspondingly.

2 Threads active	Pipeline Stages			
	Stage 1	Stage 2	Stage 3	Stage 4
1 processor cycle	Thread 1			
	Thread 2	Thread 1		
		Thread 2	Thread 1	
			Thread 2	Thread 1
1 processor cycle	Thread 1			Thread 2
	Thread 2	Thread 1		
		Thread 2	Thread 1	
			Thread 2	Thread 1

Figure 29: Execution Cycle for 2 Threads

3 Threads active	Pipeline Stages			
	Stage 1	Stage 2	Stage 3	Stage 4
1 processor cycle	Thread 1			
	Thread 2	Thread 1		
	Thread 3	Thread 2	Thread 1	
		Thread 3	Thread 2	Thread 1
1 processor cycle	Thread 1		Thread 3	Thread 2
	Thread 2	Thread 1		Thread 3
	Thread 3	Thread 2	Thread 1	
		Thread 3	Thread 2	Thread 1

Figure 30: Execution Cycle for 3 Threads

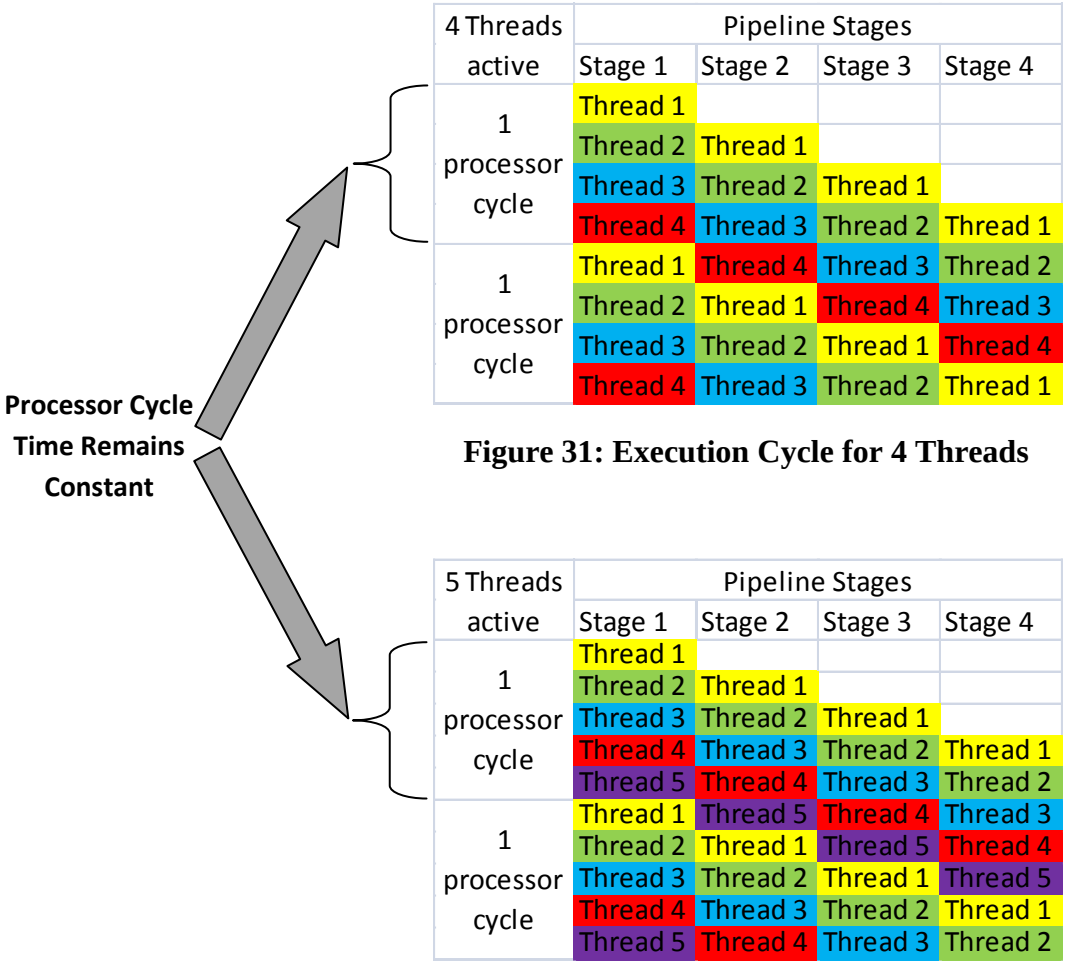

Figure 31: Execution Cycle for 4 Threads

Figure 32: Execution Cycle for 5 Threads

Up to four threads, the time spent on each thread is a quarter of the processor cycle time. But for five threads, the time spent on each thread is now 1/5 of the processor cycle, and for six threads the time spent for the execution of each thread has reduced to 1/6 of the processor cycle. From the above explanation, two conclusions can be drawn; one is that the power consumption increases from one thread to four threads being active. Beyond four threads, the power consumption stays constant since the part of the processor cycle used for all the threads put together is always constant.

The other conclusion drawn is that the amount of time taken to execute a particular workload by one thread is the same as that taken by four threads, each executing the same workload per thread. But as we start using five or more threads, each executing the same workload per thread, the time taken to complete that workload increases accordingly. This is because the amount of time spent per thread during each processor cycle reduces as seen in Figure 32 above for five or more threads.

Timers Consume Minimum Power

Another interesting result that can be drawn from the graph of Figure 25 is that the timers consume the minimum power in the system to execute, i.e. waiting for a timer to expire. This experiment was carried out with one timer per thread when more than one thread was used at a time. The power consumed by the system waiting for a timer to expire is a minimum at 17.37 mW to 17.68 mW for one to eight threads. As we can see, there is not much of a difference in the power consumption from one thread to eight threads, and looking at the graph of Figure 25, we see a rather flat line for power consumption versus the number of threads.

Threads	Voltage (mV) (V1~V2 across 1Ω resistor)	Power (mW)
1	131.8	17.37124
2	131.9	17.39761
3	132.2	17.47684
4	132.6	17.58276
5	132.2	17.47684
6	132.8	17.63584
7	132.9	17.66241
8	133.0	17.68900

Figure 33: Table Showing Power Consumption Values for 'time out'

From this we can conclude that the power consumed by the timer can be considered to be a baseline power consumption, which means that it would be the minimum power that would be consumed even if the processor is not doing anything for a respective clock frequency. All other instructions and workloads will consume a power level higher than the power consumed by a timer waiting to expire. A timer waiting to expire is considered to be doing nothing at all, it has a thread allocated and hence can be considered as the baseline power in this setup. The ideal scenario would be to have a thread allocated and doing nothing, but that was a limitation of this project, since allocating a thread and not doing any sort of work on it was not possible. But the closest one can get to form a baseline is using the timer's timeout feature which has a thread allocated and does nothing at all while waiting for a timer to expire.

NOP's are Additions

Another interesting result that can be gathered from the graph of Figure 25 above is that the amount of power consumed by an 'ADD' is almost the same as the amount of power consumed by a 'NOP' instruction. The graph plots for these instructions are overlapping one and another. A point to be noted here is that the plot for 'NOP' is slightly lower than that of the 'ADD' instruction by taking a closer look at the graph of Figure 34 for these two instructions.

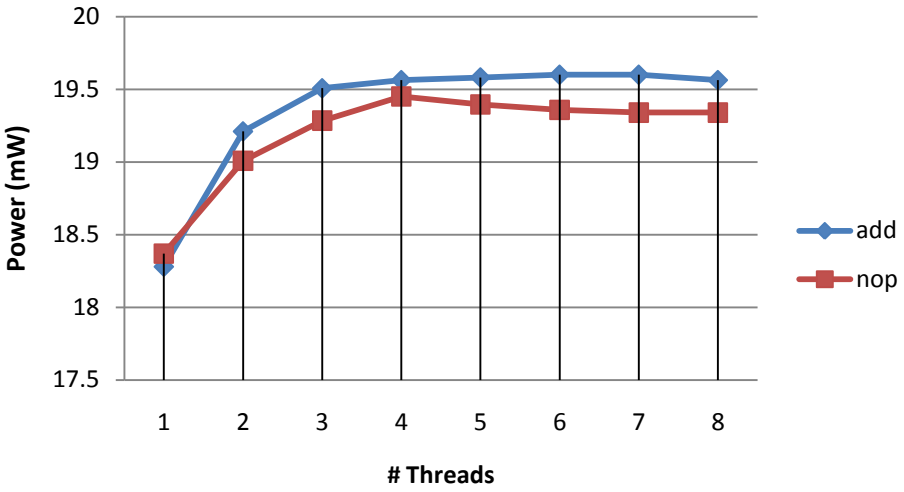

Figure 34: Graph Comparing ADD and NOP

The reason for these two plots to be almost overlapping is because in the XS1 Assembly Language Manual [29], the definition of a 'NOP' is:

ADD r0, r0, 0

This implies that a 'NOP' in reality is just adding the contents of register r0 with 0 and storing it back in r0. Hence since all 'NOP' instructions get decoded as additions, the amount of power consumed by this instruction will also be almost the same as that of an 'ADD' instruction.

The reason for the plot for 'NOP' to be slightly lower than that of the 'ADD' instruction is because of the contents of the operands. In the case of a 'NOP' a 0 is being added which requires minimum switching in the bits since all the constant operand bits are 0, whereas the 'ADD' instruction which was carried out by adding 0x5555 with 0x5555, with alternate bits being 0's and 1's, resulted in a slightly higher power consumption with respect to the 'NOP'.

Data Dependencies are Minimal

The power consumption due to the change in data operands for a particular instruction is minimal. For example adding 0x0000 with 0x0000 in one case and the adding 0x5555 with 0x5555 in another case causes minimum power consumption differences for the two cases.

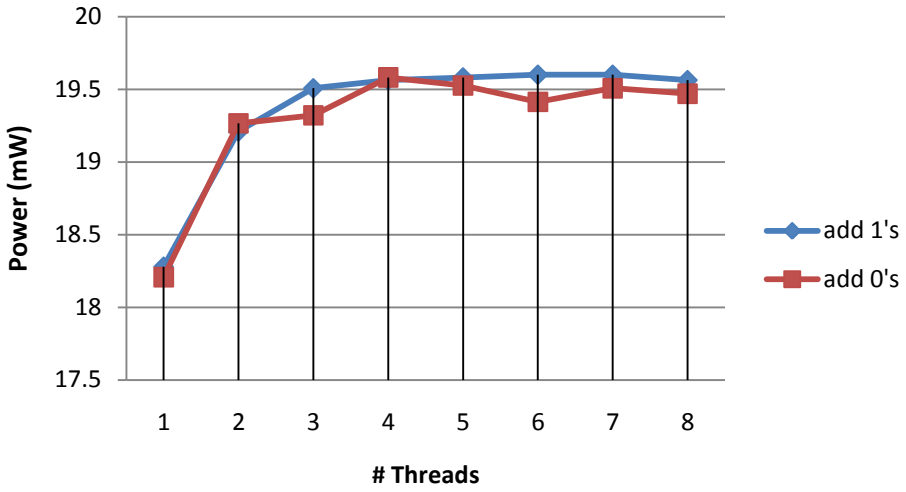

Figure 35: Graph Comparing Data Dependencies

In fact in some cases the power consumed is exactly the same where the graph plots overlap with each other.

Using Threads is Very Efficient

If observed closely, the difference in power consumption from one thread, to two, to three, and more is very minimal. In fact, the power consumption increases from one to four threads and beyond that it stays constant. The reason for this has already been discussed in the above sections. But the point to be taken away from this statement here is that threads not only provide the capabilities for parallel execution, but also allow for 100% increase in workload for only 5% increase in power consumption, and 200% increase in workload for only 6.7% increase in power consumption.

Threads	Power (mW) (ADD)	% Increase in Power Consumption from 1 Thread
1	18.27904	0
2	19.20996	5.092828
3	19.50678	6.716643
4	19.56268	7.022494
5	19.58134	7.124541
6	19.60000	7.226638
7	19.60000	7.226638
8	19.56268	7.022494

Figure 36: Increase in Power Consumption Per Thread

Thus if we had to use 8 threads to do 8 times the amount of work that would ideally be done by one thread, it takes only ~ 7% increase in power consumption but the amount of work done has increased by 700%. Yes of course the time taken also correspondingly increases but if we had to use 4 threads, we still use up an increased

7% power consumption but do 300% more work and consume the same amount of time as 1 thread. Hence it is sometimes a trade off depending on the application if 1 thread is to be used, or 4 threads to do more work in the same time, or 8 threads to do a lot more work with the same power consumption as 4 threads but with longer time duration. In either case the concepts of threads and parallel processing can be exploited to its best with the knowledge of the amount of power that would be consumed in each scenario and informed decisions can be made to choose the best suited approach.

More Resources More Power

The XCore has various resources that need to be managed such as threads, synchronizers, channels and channel ends, timers and locks [26]. Each of these resources consumes power. For example adding more threads clearly increases the power consumption as has been discussed earlier. Also it has been noted that adding more channels and channel ends increases the power consumption too. This is because allocation of such resources costs the processor, in terms of power consumption, in order to maintain an active state of the resource. Allocating a resource and keeping it in the active state consumes energy.

4.3 Part 2: Results and Analysis

Once the power profiling of the common instructions was done individually, it would be an interesting aspect to see what would happen if we clubbed different instructions into the workload programs. Ideally in real life scenarios this is the situation we are bound to encounter, where in the assembly program consists of different instructions at every stage. Running the same instruction over and over again does not give an accurate idea about a programs' power consumption because the program execution generates very little switching activity [34], and hence understanding their working in various patterns would be an interesting aspect.

Orthogonality Does Not Hold Good

When an 'ADD' instruction is coupled with a 'MUL' instruction, one would expect that the total power consumption of the system is an average of the 'ADD' and 'MUL' individually, but, this is not that case that was observed on the XCore. Various combinations of 'ADD' and 'MUL' instructions were coupled together, some with alternate ADD's and MUL's on a single thread, some with alternate groups of ADD's and MUL's on a single thread, some with first half ADD's and second half MUL's, and many others. All the combinations resulted in the same outcome, i.e. the power consumed when these instructions are coupled together is not an average of the two instructions' individual requirements.

All these combinations resulted in almost the same power consumption with a few minute variations, but overall the power consumption is as listed in the table of Figure 37 below. It should be noted that the work done in either case is the same since the number of instructions of each category executed is the same, i.e. 20 ADD's and 20 MUL's in one loop repeated 500,000 times.

Threads	Power (mW) (ADD)	Power (mW) (MUL)	Power (mW) (average)	Power (mW) (combo)
1	18.2790	18.6413	18.4602	18.8055
2	19.2099	19.6000	19.4049	19.7121
3	19.5067	19.8058	19.6563	20.0505
4	19.5626	20.1640	19.8633	20.1261

Figure 37: Table for Power Consumption Comparison for Part 2 Experiments

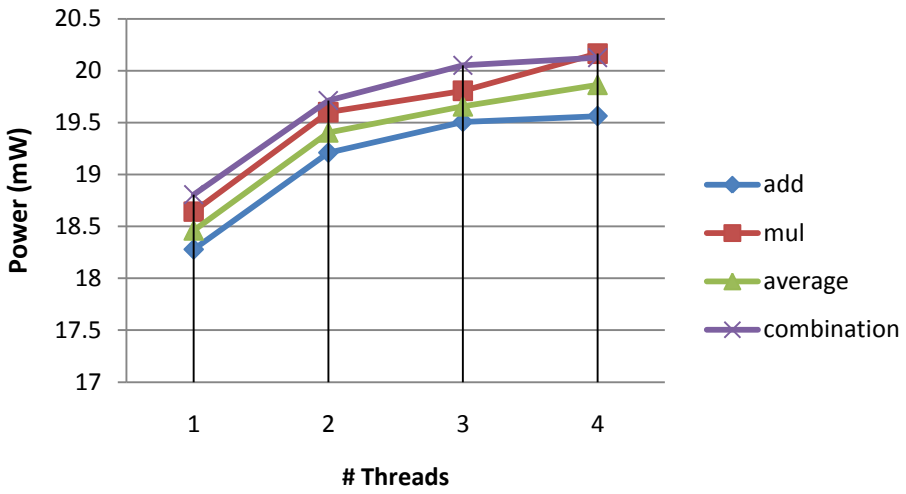

Figure 38: Graph for Power Consumption Comparison for Part 2 Experiments

We would expect the combinations' power consumption to fall around the average (green) plot. But as seen in the graph the combinations' power consumption is greater than the individual readings. The reason for this behaviour can be because of higher *Hamming Distances* in the encoding of the opcodes. Hamming distance between two strings x and y is the number of positions in which x and y are different [30] and in this case in terms of the different bits of the encoded opcodes. Thus if we have lower hamming distances, there is minimum switching in the transistors resulting in lower power consumption. Higher hamming distance results in more switching at the transistor level resulting in higher power consumption. This is because the power consumption is directly proportional to the product of the capacitance and the switching frequency.

$$P \propto C*f$$

Thus with higher hamming distances, we can expect higher switching frequency and hence higher power consumption. For a workload with all MUL's or all ADD's, the hamming distance would be minimum except where the constants change, but in

general the encoded opcodes are the same resulting in lower power consumptions when compared to an alternate set of ADD's and MUL's.

Design for Predictability

As seen in the previous section, the power consumption for the overall system with a category 2 type of workload, where different instructions are grouped together, is more than the isolated power consumed by the individual instructions. This sort of behaviour can be explained to some extent but is unpredictable in a sense. If a static power analysis tool had to be developed, which is the ultimate motive of studies like this project, and then one would expect a somewhat predictable behaviour when it comes to power consumption. Thus in the future, would it be more appropriate to design processor architectures that are power predictable? This is something which can be debated for long, but the real question is, would such designs be feasible?

4.4 Part 3: Results and Analysis

The third set of experiments carried out involved workloads with long instructions such as the long multiply. This was done in order to understand how the power consumption is affected when we use complex instructions to do a certain piece of work, when compared to its RISC counterparts. More importantly we understand the energy (power over time) consumption of the system with the help of this experiment.

Complex Instructions are Power Efficient

Complex instructions like the 'LMUL' (long multiply), multiplies two words to form a double-word and adds to single words [26]. If the same were to be done using RISC instructions, then one would have to have a single multiply and two additions. So in order to understand the energy (power over time) consumed in performing complex operations, the experiments aimed at gathering two sets of data. One set would be the power consumed for the long multiply instruction with the time taken to execute a certain number of instructions. The second set of data to be gathered would involve doing the same amount of work, but in terms of basic RISC instructions. This means that if we had to execute 100 LMUL's, then the second set of programs will require 100 MUL's and 2*100 (200) ADD's to perform the same amount of work as a 100 LMUL's.

1 LMUL = 1 MUL + 2 ADD

100 LMUL = 100 MUL + 200 ADD

The experiment was carried out for one thread. The results obtained from this experiment were quite interesting and are summarized in the table of Figure 39 below. The workload for the first set used 50 LMUL in a loop of 500,000. The workload for the second set used 50 MUL and 100 ADD in a similar loop of 500,000.

Instruction	Power (mW) average	Time (ms)
LMUL	18.70512	310
MUL, ADD, ADD	19.31753	810

Figure 39: Table Comparing Power Consumption & Time Taken for Complex Instructions & Simple Instructions

The results show that the complex instruction not only took lesser time to execute which is expected, it also consumed 0.612 mW less power compared to the equivalent RISC instructions. The main reason for this is again due to the increased hamming distances due to the multiple instructions being executed one after the other. In the case of the LMUL, the hamming distance is low since the opcodes encoding will not vary from one line of code to the next. But for the RISC equivalents, the encoded opcodes are changing for every single line of code resulting in higher hamming distances and hence greater transistor switching on the hardware, resulting in greater power consumption.

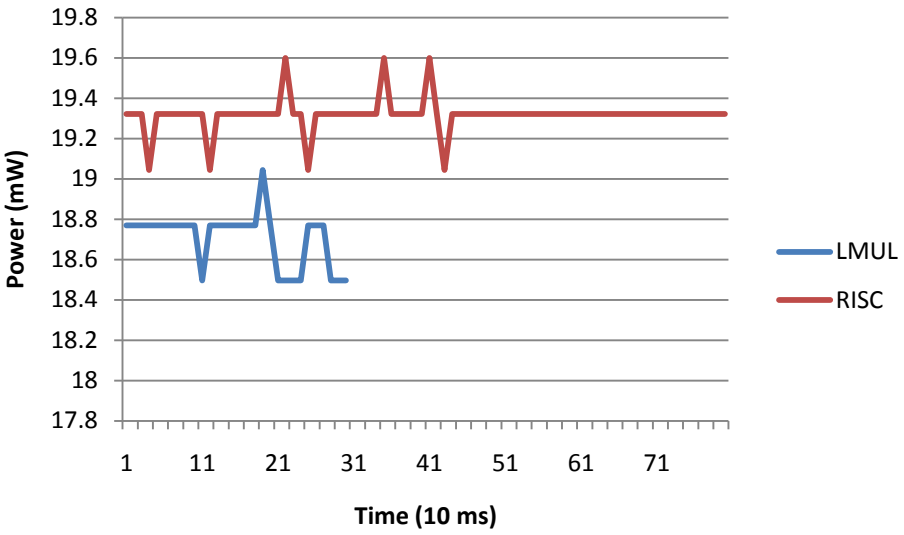

Figure 40: Graph Comparing Power Consumption & Time Taken for Complex Instructions & Simple Instructions

Based on the graph of Figure 40 above, it can be concluded that knowledge of the architecture and a good understanding of the instruction set can help in making informed decisions while designing software or compilers. This would help in not only developing power aware software, but also better performance in terms of time taken for execution of the same amount of workload.

4.5 Additional Results

On completion of the three part experiments that were intended, further analysis was done on branches in order to improve on the power profile obtained so far. Branches proved to be tricky instructions to program as workload benchmarks, but were accomplished none the less. Ideally, one would encounter a scenario wherein based on a condition, we jump to a particular location or carry on with the program execution. In order to simulate both these scenarios two forms of branch instructions were tested for:

Branches Taken

Branches Not Taken

The instruction used for branches taken was **BRBU**, branch relative backwards unconditional [26]. This instruction implements a backward jump relative to the immediate operand that specifies the offset. This instruction changed the program counter (pc) value based on the immediate operand for a relative backwards jump resulting in

pc = pc – operand*2

Now if we made the immediate operand as 0, then the pc would not be updated and as the pc points to the next instruction to be executed, this would mean that we are actually performing a jump, but a jump to the next location. This can be added into a large loop to gather the data for power consumed when the branch is taken.

Similarly the instruction used for the branch not taken experiment was **BRBT**, branch relative backwards true [26], which meant that this instruction would implement a conditional relative jump backwards, condition being the contents of a register. If the contents of the register were non zero, then the branch would not be taken and the program execution would continue. Thus in order to achieve this, the contents of the register were filled with a non zero value ensuring that the branches were never taken and move in a similar fashion as before in a large loop.

The results from the two experiments are summarised in the graph of Figure 41 shown below.

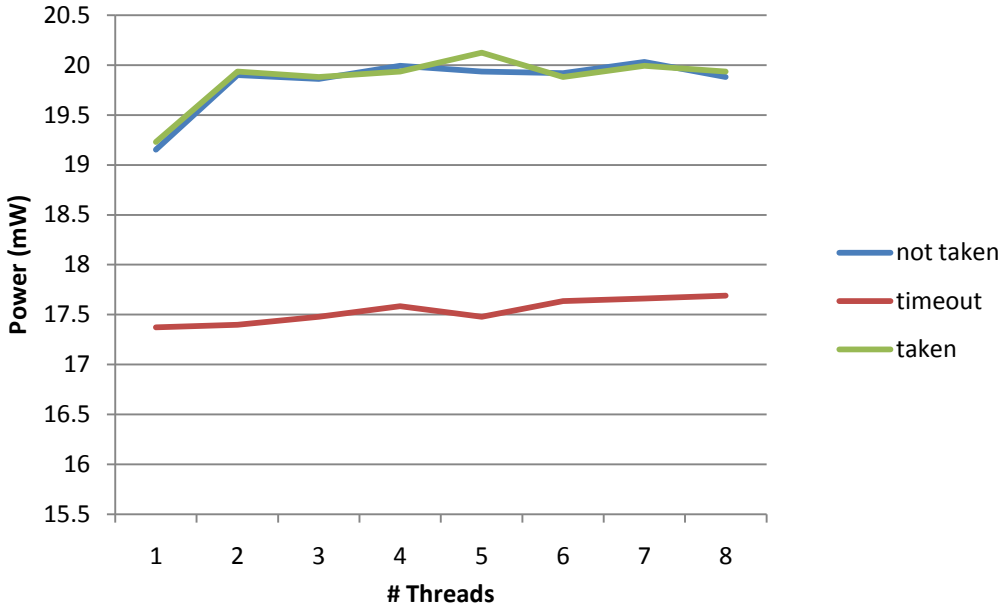

Figure 41: Branches Taken & Not Taken

As seen, the branch instruction, whether taken or not taken, consumes almost the same amount of power and is quite high, second to the load instruction. This is quite what is expected because branches in general require more resources to handle and are quite expensive, in terms of power. The pattern followed is similar to the previous experiments wherein the power consumed for a single thread is minimum and the increases up to four threads. Beyond four threads, the power consumption is constant up to eight threads being active at once.

4.6 Summary

After lots of experimentation and refinement of the benchmark programs, a basic power profile was obtained as a result of the Part 1 experiments. Part 2 and Part 3 experiments also resulted in some very interesting results. The overall conclusions drawn from this project are summarized as:

- Threads are very effective
- Timers consume minimum power in the system
- NOP's are additions in reality
- More resources used means more power consumed
- Data dependencies are minimal
- Power consumption for one thread is minimum and increases up to four threads and beyond that remains almost constant
- Orthogonality of instruction execution does not hold good
- Complex instructions are efficient and consume less power
- The software developer should be power aware of the target architecture

Chapter 5: Conclusion & Critical Evaluation

The main aim of this project was to create a power profile for the various instructions of the XCore architecture which was used for carrying out this project. In order to achieve this aim, various objectives were initially laid out as in **Chapter 1** and in this section we will discuss the extent to which each of these objectives were achieved, and whether the choices that were made initially with hindsight were, the best ones at the time.

To undertake a project of this novel nature, a lot of background work had to done, in the form of understanding the technicalities of power consumption and the two components of power consumption, static and dynamic power as described extensively in **Chapter 2**. A lot of research was found with respect to the hardware techniques of power optimization and the need to optimize power at the software level. But the research carried out on power optimization methods at the software level was very limited and sparse. It was quite difficult in identifying articles and work done in the field of this project and this can be attributed mainly due to the novel nature of this project. Initially, during the write up of the interim report, there was very less background mentioned on the optimization methods at the software level, but in this report, with a lot of time spent on finding articles in this field, some more techniques have been identified. Overall this objective of background research was carried out with a reasonable degree of success over the entire project cycle.

The next important objective was to design the software framework of this project which is extensively described in **Chapter 3,** section on '**Design Framework**' of this report. One major difference in the current project framework compared to the one initially planned was that way in which the power consumed was to be measured. Initially it was decided that the power measurements would be done by, first executing the program and measuring the power consumed by the processor over a period of time, and then manually correlating the power consumed to the instruction that was being executed with the help of the XMOS Timing Analyser (XTA). The XTA would identify the instruction being executed at a particular instant of time and then this would be compared with the output of the power consumption over time, the common factor between the two being time, making time a critical factor in the correlation. The procedure to carry this out is shown in the impression of Figure 42 below, and as seen, first we need to use the timing analyser to see at what point in time which instruction is being executed, and then on another graph, view the power consumption over a period of time, and then correlate them to each other. This is quite a tedious and manual task, not to mention time consuming. In the technique used in this project, this procedure was done away with and a more automated version of the same was used as described in Chapter 3, section on 'Design Framework'. This made experimentation much easier and faster. The manual procedure for measuring power consumption initially planned was one choice that was not the best one initially before the start of this project. Hence the current method for data gathering was quite effective and met with a good degree of success.

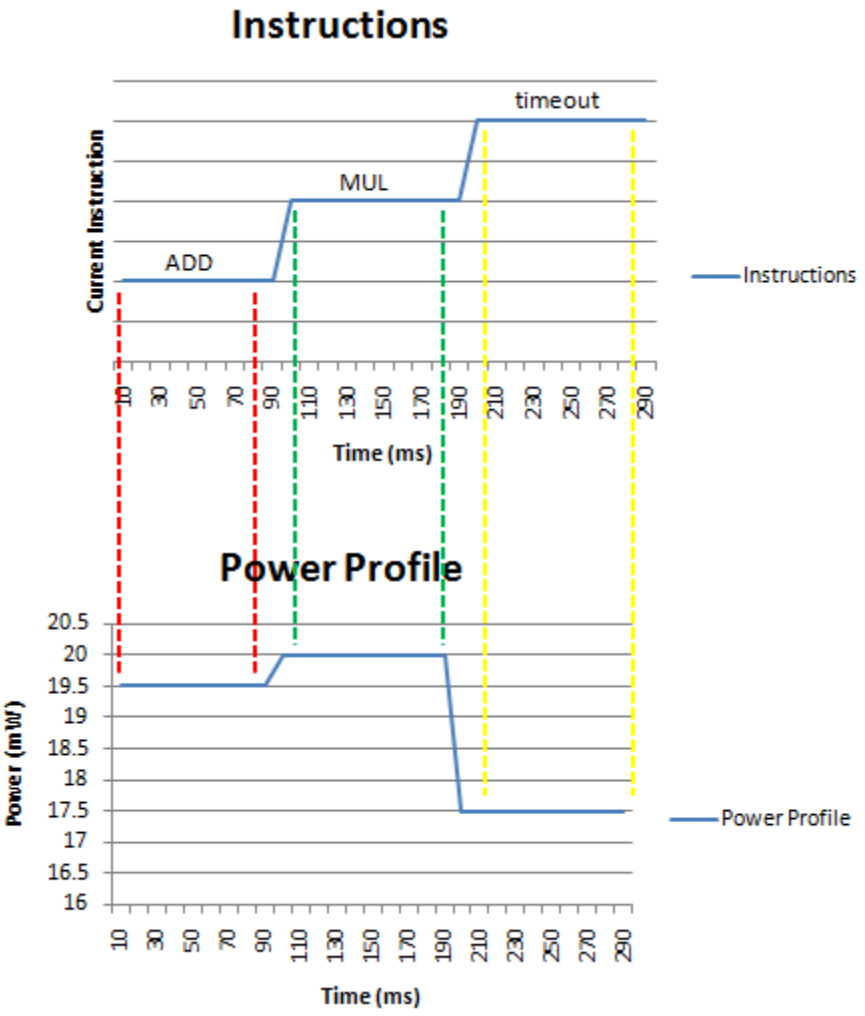

Figure 42: Impression of Manual Correlation of Power Consumption

The framework designed on the whole is not a very elegant one, as the way in which the power consumed was measured could still have been refined and bettered a bit more. In the current method, once the workload is executed and the power consumption is measured, the data which is saved in an array is printed out on the console and later copied onto excel for analysis and graphical comparisons. This could have been further bettered and automated in ways mentioned in the improvements section of this report. The main reason for this can be attributed to the lack of in depth knowledge in the XC programming language and certain concepts like parallel programming and channel usage.

The method used to input the benchmark programs into the main program was quite simple and efficient. As described in detail in **Chapter 3**, section on '**Design Benchmark Programs**', a switch case statement was used to feed in various types of workload patterns, and once all the workload patterns were programmed, just by changing the value of the case expression, one could change the workload. This method, besides making the data gathering quick and efficient, also ensured a clean way of coding, wherein all the functions corresponding to the various workloads were called from a single location and defined elsewhere. The breakup of the project benchmark

programs in three different parts was a systematic way of working on the project, first by running programs for power profiling, and then using the data gathered from this power profile to verify the outcome of the second part programs that considered mixed combinations of workloads, and then finally moving on to some more interesting outcomes of the power profiling by studying the effects of the complex instructions on the power consumption of the processor.

The data gathering procedure was quite crude, where the voltage values were copied from the console on the development environment and pasted into excel for analysis and comparisons. This could have been improved with a better knowledge of the development environment and the XC programming language. But considering the time duration of this project, the method used was simple and easy, and the analysis of the data gathered could be done on time. The way in which the data was collected was quite clever and quick, and as described in **Chapter 3**, section on '**Measure Power Consumption**', instead of using the timing analyzer which would have been quite time consuming and tedious, an automated approach was used which made the data gathering quick, easy and generic to use with various workload patterns as inputs.

The results section of **Chapter 4** explained in detail, all the results that were obtained, and their analysis in a very systematic manner. The analysis followed the same structure as the design of the benchmark programs, in three parts. This enabled a good understanding of the basic power profile obtained followed by the analysis of the combinational instructions and finally the analysis of the complex instructions. Most of the analyses done were with a good understanding of the background and the working of the XMOS XCore architecture in mind. There were some results which could have been explained better. For example the reasons for the power consumption due to the combinational instructions (Part 2 benchmark programs) were analysed to be due to the larger hamming distances. The concept of hamming distances and how exactly does that affect the power consumption could have been explained in depth with some data on the encoded bits and the actual calculation of hamming distances. The reason for this is because if one were to develop an equation for finding the power consumption of a combination of more than one instructions running one after each other, then understanding how these hamming distances correlate to the power consumed would be a good basis. This can be considered as some future work on this project, wherein knowing the actual distances would help in forming an equation that a compiler could use to do some static power analysis. But overall the results were demonstrated well in a very readable graphical manner with the use of tables where necessary for comparison of values. Good use of figures was made to explain the various experimental setups and in understanding the working of the project execution flow.

The number of experiments carried out for Part 1 of the benchmark programs was quite extensive and varied, resulting in a good power profile. Part 2 and Part 3 had a limited number of experiments that were carried out, especially Part 2 which had the potential for more experimentation. But Part 1 covered the main aim of this project that was initially planned for and was accomplished with a great deal of success. Since this was completed well before the scheduled time, there was room for Part 2 and Part 3 experiments enumerating to the study of this project.

To conclude, the project achieved most of its objectives as expected and with a great deal of success and more. There is room for improvements in some areas. The overall conclusions drawn from this project are:

- Threads are very effective
- Timers consume minimum power in the system
- NOP's are additions in reality
- More resources used means more power consumed
- Data dependencies are minimal
- Power consumption for one thread is minimum for one thread and increases up to four threads and beyond that remains almost constant
- Orthogonality of instruction execution does not hold good
- Complex instructions are efficient and consume less power
- The software developer should be power aware of the target architecture

If the findings of this project can form the basis of further work in developing low power software, it can impact the way in which the current power versus performance trends follow. Power aware software would be an essential feature in the future which would offer software developers and system architects a means to be able to start optimizations much higher up in the abstraction level. Also, in a world where more and more features are to be included on handheld devices, a world where a server farms' energy consumption is a matter of concern, we need to look into methods to drastically improve the power dissipation, both static and dynamic. To ensure that the power-performance-cost factor is kept in mind, we need to look at ways to keep up with the trend of ever increasing performance. This project will offer a practical platform for multiple architectures and, because of its novel nature, it can be used as a good starting point for further research. Also further to this project, research can be done on exploring different techniques and automation of inclusion of power optimized structures that will tune the application software by correlating the power with the system workload.

Chapter 6: Improvements & Future Work

Though a good deal of work was carried in the short time this project was executed, there is still room for improvements and future work in the same area. This section describes the improvements and future work that can be done on this project.

6.1 Improvements

The first improvement that needs to be done is the re-design of the software framework, since the current design, though simple, is crude and not very elegant. The various functions that are used in the project can be partitioned in a well defined manner, and also with the use of multiple files, each one representing a task, can make the code more readable. The way in which the benchmark programs are input to the framework is simple, but can be improved further by having a method that can execute all the programs in a single execution.

The data collection definitely needs a great deal of improvement, as the current method for viewing the results is copying from the console and pasting it into an excel workbook. This method can be automated such that the data gathered is automatically fed into a file and the graphs automatically plotted along with the power calculations. Currently the power is also calculated manually using formulas in excel, but if the design can be integrated with the power equation, the power consumed can be directly viewed based on the voltage values that are actually obtained.

Another area for improvement is the contents of the benchmark programs. Part 1 benchmark programs were quite good, but not exhaustive enough, the power profile should ideally contain the power details of all the instructions in that particular instruction set architecture. The benchmark programs under Part 2 and Part 3 should also be improved further, made more exhaustive with various combinations executed.

An essential area for improvements would be to develop an equation for estimating the power consumption of the system based on the baseline power profile. As seen in Part 2 experiments, the power consumption of two different instructions is not an average of the two, but is most likely a function of the two based on hamming distances and the amount of switching that occurs between the two opcodes.

6.2 Future Work

One good area of research can be to build a power profile library. This library should have the capability of initially updating itself based on the target architecture and then allow the user to use the same power profile library in tools that are capable of static power analysis. This would make the library generic and usable across various platforms.

Then the next step of development can be to create a tool that would use this power profile library and provide an estimate of the overall power consumption along with the timing information. An interesting feature of this tool would be to provide the

power information of specific functions that are part of a larger program. Another interesting feature that could possibly be integrated into the tool is the traffic light system, wherein, if a programmer comes across two instructions, whose combination will drastically increase the power consumption of the system, the traffic light integrated into the tool could turn red indicating to the developer the potential power hazard. This could be a dynamic process where each instruction is compared with the previous and the next while the software is being written.

The final step to the tool development can be to build a compiler that can use the library function and the tool to estimate the power consumption for various functions at various instances of time and then re-compile the program for optimized power consumption.

Bibliography

[1] “Mobile Supercomputers, Embedded Computing”, Todd Austin, David Blaauw, Scott Mahlke and Trevor Mudge, University of Michigan, Chaitali Chakrabarti, Arizona State University, Wayne Wolf, Princeton University, 2009.

[2] “Moore’s Law and its Implications for Information Warfare”, Carlo Kopp, BE(Hons), MSc, PhD, PEng, Computer Science & Software Engineering, Monash University, Clayton, 3800, Australia, 2000, 2002, Carlo Kopp.

[3] “Advanced Computer Architecture, A Design Space Approach”, Dezso Sima, Terence Fountain, Peter Kacsuk, ISBN 0-201-42291-3.

[4] “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures”, Vikas Agarwal et. al., appears in the Proceedings of the 27th Annual International Symposium on Computer Architecture.

[5] “Low Power Design Essentials”, Jan Rabaey, September 2008, Issue 1, SCD Source.

[6] “Active Leakage Control with Sleep Transistors and Body Bias”, Zhengya Zhang and Zheng Guo

[7] “Power Efficient Design: Challenges and Trends”, Barry Pangrle, Solutions Architect, Mentor Graphics (http://www.mentor.com/events/tech-talk/)

[8] “Estimating Power for ADSP-BF561 Blackfin® Processors”, contributed by Joe B, Revision 2, June 27, 2007

[9] “CMOS VLSI Design: A Circuits and Systems Perspective”, Weste and Harris, Third edition, ISBN 0-321-14901-7

[10] “CMOS Power Consumption and Cpd Calculation”, Abul Sarwar, Texas Instruments

[11] “Low Power Design Techniques for Leading Edge Chip”, Chin-Chi Teng, Ph.D., Engineering Director, Cadence Design Systems, Inc.

[12] “A New Technique for Standby Leakage Reduction in High Performance circuits”, Y. Ye, S. Borkar, V. De, Symposium of VLSI Circuits Digest Papers, 1998, pp. 40-41.

[13] “Microarchitectural Techniques for Power Gating of Execution Units”, Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, Pradip Bose, IBM T. J. Watson Research Center.

[14] “Keep Hot Chips Cool”, Ruchir Puri, Leon Stok, Subhrajit Bhattacharya, IBM T.J. Watson Research Center, Yorktown Heights, NY.

[15] "CLOCK GATING ARCHITECTURES FOR FPGA POWER REDUCTION", Safeen Huda, Muntasir Mallick, Jason H. Anderson, Dept. of ECE, Univ. of Toronto, Toronto, ON Canada.

[16] "Power Reduction Through RTL Clock Gating", Frank Emnett and Mark Biegel, Automotive Integrated Electronics Corporation.

[17] "Ultra Low-Power Electronics and Design", edited by Enrico Macii. Boston, Mass, London, Kluwer Academic Publishers, c2004. xvi, 273 p.

[18] http://thepcweb.com/?p=199

[19] "Design for Low-Power at the Electronic System Level", Frank Schirrmeister, ChipVision Design Systems.

[20] "ISSCC 2010 Trends Report"

[21] "How to Optimize Power through Transaction Level Analysis and High Level Synthesis", Jon McDonald, Technical Marketing Engineer, Mentor Graphics (http://www.mentor.com/events/tech-talk/)

[22] "System Approach to Low Power Design", Walden C Rhines, Chairman & CEO, Mentor Graphics, 1 April 2009.

[23] "How High Level Modeling Speeds Low Power Design", Glenn Perry, Mentor Graphics, (http://www.scdsource.com/article.php?id=293)

[24] "Analyzing IC Power at the Electronic System Level", Lars Kruse, ChipVision Design Systems, (http://www.scdsource.com/article.php?id=82)

[25] "Low Power Methodology Manual for System on Chip Design", Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, Kaijian Shi, ISBN 978-0-387-71818-7, 2007, Chapter 1.

[26] "The XMOS XS1 Architecture", David May, 19/10/2009, XMOS.

[27] "Tools User Guide", Huw Geddes, Version 9.7, 22/07/2009, XMOS.

[28] "Programming XC on XMOS Devices", Douglas Watt, 24/09/2009, XMOS.

[29] "XS1 Assembly Language Manual", Douglas Watt, Version 9.9, 15/10/2009, XMOS.

[30] "Hamming Distance for Conjugates", Jeffrey Shallit, Discrete Mathematics Volume 309, Issue 12, 28 June 2009, Pages 4197-4199

[31] "Why You Should Optimize Power at the Electronic System Level", Yossi Veller & Shabtay Matalon, Mentor Graphics, www.mentor.com

[32] “Function-Level Power Estimation Methodology for Microprocessors”, Gang Qu, Naoyuki Kawabe, Kimiyoshi Usami, and Miodrag Potkonjak, Computer Science Department, University of California, Los Angeles, CA 90095, USA, Design Methodology Dept., Semiconductor Company, Toshiba Corporation, Kawasaki, 210, Japan

[33] “Power Analysis of Embedded Software: A First Step Towards Software Power Minimization”, V. Tiwari, S. Malik, and A. Wolfe, IEEE Transactions of Very Large Scale Integration Systems, Vol.2, No.4, pp. 437-445, 1994.

[34] “Methodology for Instruction Level Power Estimation in Pipelined Microsystems”, Robert M. Senger, Eric D. Marsman, University of Michigan, Richard B. Brown, University of Utah

[35] “XS1-L1 128TQFP Datasheet”, Version 2.1, XMOS, Publication Date: 2010/05/20

Appendix A: Framework Source Code

Only the essential parts of the framework are listed here, some declarations have been removed due to space constraints

```
/*
 * Core 0 -> Master
 * Core 1 -> Slave
 */

/*
* Need a minimum of 1000000 (10ns x 1000000 = 10ms)
* delay to process measure_task function
*/
# define DELAY_ITERATION 1000000

void measure (chanend);
void load (chanend);
int measure_n (void);

// Global arrays for storing voltage values
int array1 [1000], array2 [1000];
```

The main function that calls measure and load in parallel

```
int main (void)
{
par
        {
                // Call measure and load functions in parallel
                on stdcore[0] : measure(ctrl); // Master, needs 1 thread
                on stdcore[1] : load(ctrl);    // Slave, can use up to 8 threads
        }

        return 0;
}
```

The load function that calls from one, up to eight threads in parallel

```
void load(chanend ctrl)
{
        // set the channel by default to 0 to make sure it is asserted low
        ctrl <: 0;

        // run n threads in parallel to make sure optimum use of processing
        par
        {
                a = measure_1(); // thread 1
                //b = measure_1(); // thread 2
                //c = measure_1(); // thread 3
                //d = measure_2(); // thread 4
                //e = measure_n(); // thread 5
                //f = measure_n(); // thread 6
                //g = measure_n(); // thread 7
                //h = measure_n(); // thread 8
        }

        // once the load program is complete signal the channel to end the program
        ctrl <: 1;
}
```

The measure function that measures the power consumed

```
void measure(chanend ctrl)
{
        ctrl :> i; // get the 0 for start from ctrl channel

        t :> time;
        timeStart = time;
        time += DELAY_ITERATION;

        while(!i)
        {
                select {

                // wait for i to be 1 and then exit loop
                case ctrl :> i:
                        timeEnd = time;
                   break;
```

```
                // wait for timer to expire and then call measure_task
                case t when timerafter(time) :> void:
                        time += DELAY_ITERATION;
                        measure_task(j, scl, sda);
                        j++;
                        break;

                }

        }
}

void measure_task(int index, port p_iic_scl, port p_iic_sda)
{
        // Initialise the I2C port
        iic_initialise ( p_iic_scl, p_iic_sda );

        // Setup the ADC to sample all channels
        wrData[0] = 0x30;
        iic_write ( p_iic_scl, p_iic_sda, iic_address_adc, wrData, 1 );

        // Loop getting data
        for(int i = 0; i < 16; i++)
        {
                // Read the data from the ADC
                iic_read ( p_iic_scl, p_iic_sda, iic_address_adc, rdData, 4 );

                // Get the 8-bit values from the data received and scale to mV
                // ( adc_value * (3300 / 256) )
                a0 += (((rdData[0] & 0xf) << 8) | (rdData[1] & 0xFF)) ;
                a1 += (((rdData[2] & 0xf) << 8) | (rdData[3] & 0xFF)) ;
        }

        // Write out the values to the debug
        array1[index] = ( ((a1-a0) * 52800) >> 20 );
        array2[index] = ( ( a0 * 52800 ) >> 20 );

        return;
}
```

Appendix B: Benchmark Programs Source Code

Only the essential parts of the benchmark programs are listed here, some declarations have been removed due to space constraints. Since the work was carried in 3 parts, the benchmark programs here are also divided into 3 different parts.

```
int measure_n (void)
{
	int load = 17; // can take values from 0 -> 17

	switch(load)
	{
		case 0 : addition(); break;
		case 1 : subtraction(a, b); break;
		case 2 : multiplication(); break;
		case 3 : division(a, b); break;
		case 4 : nop(); break;
		case 5 : timer_waiting(); break;
		case 6 : load_instr_1(); break;
		case 7 : load_instr_2(); break;
		case 8 : alternate_1(); break;
		case 9 : alternate_2(); break;
		case 10 : alternate_3(); break;
		case 11 : alternate_4(); break;
		case 12 : alternate_5(); break;
		case 13 : long_instr_1(); break;
		case 14 : long_instr_2(); break;
		case 15 : alternate_operands(); break;
		case 16 : branches_not_taken(); break;
		case 17 : branches_taken(); break;

		/* keep adding more cases */
		default : break;
	}

	return;
}
```

Part 1: Basic Instructions

```
void addition()
{
	int i;
	__asm__ ("ldc r0, 0x5555");
	__asm__ ("ldc r1, 0x5555");

	for (i=0;i<500000;i++)
	{
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");

		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
	}
}

void multiplication(void)
{
	int i;
	__asm__ ("ldc r0, 0x0000");
	__asm__ ("ldc r1, 0x0000");

	for (i=0;i<500000;i++)
	{
```

```
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");

		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
	}
}

void timer_waiting()
{
	timer t;
	int time;

	t :> time;
	time += (DELAY_ITERATION*10);
	t when timerafter(time) :> void;

}
```

Part 2: Combinational Instructions

```
void alternate_1() /* 1 add and then 1 mul repeated 20 times */
{
	int i;
	__asm__ ("ldc r0, 0x5555");
	__asm__ ("ldc r1, 0x5555");

	for (i=0;i<500000;i++)
	{
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");

		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("mul r2, r0, r1");
	}
}

void alternate_2() /* 20 add and then 20 mul */
{
	int i;
	__asm__ ("ldc r0, 0x5555");
	__asm__ ("ldc r1, 0x5555");

	for (i=0;i<500000;i++)
	{
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");

		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
```

```
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
	}
}

void alternate_3() /* 20 mul and then 20 add */
{
	int i;
	__asm__ ("ldc r0, 0x5555");
	__asm__ ("ldc r1, 0x5555");

	for (i=0;i<500000;i++)
	{
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");

		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
	}
}

void alternate_4() /* 10 add and then 10 mul repeated 2 times */
{
	int i;
	__asm__ ("ldc r0, 0x5555");
	__asm__ ("ldc r1, 0x5555");

	for (i=0;i<500000;i++)
	{
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");

		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");

		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
		__asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");

		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
	}
}

void alternate_5() /* 10 mul and then 10 add repeated 2 times */
{
	int i;
	__asm__ ("ldc r0, 0x5555");
	__asm__ ("ldc r1, 0x5555");

	for (i=0;i<500000;i++)
	{
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
		__asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
```

```
        __asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");

        __asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
        __asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
        __asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
        __asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
        __asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");

        __asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
        __asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
        __asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
```

Part 3: Complex Instructions

```
void long_instr_1()
{
    int i;
    __asm__ ("ldc r0, 0x5555");
    __asm__ ("ldc r1, 0x5555");

    for (i=0;i<500000;i++)
    {
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
```

```
            __asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");
            __asm__ ("mul r2, r0, r1"); __asm__ ("mul r2, r0, r1");

            __asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
            __asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
            __asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
            __asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
            __asm__ ("add r2, r0, r1"); __asm__ ("add r2, r0, r1");
        }
}
```

```
                __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
                __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
                __asm__ ("lmul r5,r4, r3, r2, r1, r0"); __asm__ ("lmul r5,r4, r3, r2, r1, r0");
        }
}

void long_instr_2()
{
        int i;
        __asm__ ("ldc r0, 0x5555");
        __asm__ ("ldc r1, 0x5555");
        __asm__ ("ldc r2, 0x5555");
        __asm__ ("ldc r3, 0x5555");

        for (i=0;i<500000;i++)
        {
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4") ;__asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");

                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4") ;__asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");

                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4") ;__asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
                __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4"); __asm__ ("mul r5, r3, r2"); __asm__ ("add r4, r0, r1"); __asm__ ("add r5, r5, r4");
        }
}
```