Department of Computer Science

Submitted in part fulfilment for the degree of BEng.

Generation of Hardware Accelerators for an FPGA System

Jay Valentine

18th October 2021

Supervisor: Dr. Christopher Crispin-Bailey

To Katie, who supported me throughout this project and the last two years of my degree, and my parents, who made this all possible and helped me grow into the person I am today.

Acknowledgements

I would like to thank my supervisor, Dr. Chris Crispin-Bailey, for his guidance throughout this project.

Contents

List of Figures

List of Tables

Executive Summary

0.1 Motivation

As transistor densities in modern microprocessors increase, a breakdown in Dennard scaling [1] has been observed. This has led to the phenomenon of dark silicon, which describes transistors which, due to power or temperature limitations, cannot be used to their full potential [2]. This has severe consequences for the future of microprocessor architecture, especially in domains where energy consumption is of great concern, such as the rapidly growing mobile application domain.

A shift away from more traditional multicore architectures to heterogeneous platforms is seen as one solution to this problem. One such architecture is the *coprocessor-dominated architecture* (*CoDA*), in which one or more general-purpose processing cores are coupled with a large number of specialised hardware accelerators, which are able to perform very specific tasks faster and with greater energy efficiency than a general-purpose core.

0.2 Aims

This work aims to show that a coprocessor-dominated architecture can be utilized in an embedded FPGA platform. The use of an FPGA allows the accelerators to be designed specifically with the embedded application in mind. The aim is to produce a tool that can generate hardware accelerators from an application written for the Xilinx MicroBlaze soft processor [3], providing increases in performance and energy efficiency.

Because the accelerators are automatically generated by the tool, rather than designed by hand, the architecture can be re-generated for each version of the application, or even across different applications, very easily. The use of an FPGA allows the architecture to be very highly specialized, as the FPGA can be re-programmed for each architecture version. In this sense, the architecture itself becomes an extension of the application software, rather than a static platform as has traditionally been the case.

0.3 Methodology

To obtain measurements of the system's performance and energy efficiency, two benchmark applications were used, written in C and compiled using the Xilinx MicroBlaze GNU tools. These applications were then simulated using Vivado XSim with a range of coprocessor configurations to obtain performance measurements, including application speedup. Finally, the design was synthesised using Vivado to obtain power consumption estimates and measures of FPGA resource utilization.

0.4 Results

It was found that for some configurations, the architecture did provide a performance improvement, with speedups of up to 1.3x measured. This improvement was not as significant as was hoped. Power consumption estimates were poor, with the architecture offering no improvement over a base MicroBlaze system, and in many cases even resulting in increased power consumption. A range of causes of this poor performance were identified and solutions to these outlined as further investigation in this area of research.

0.5 Statement of Ethics

Neither the undertaking of this project, nor its end result, are envisioned to cause any harm. All testing was automated and performed entirely using software tools; no personal data was collected, nor were humans or animals participants in testing in any way.

Some portions of code used as test applications in the evaluation of this system were sourced from authors who had released the code under open source licenses. Great care was taken to ensure that all code used for this purpose was credited properly and that the licenses with which the code was provided permitted its use in this project.

1 Literature Review

1.1 Moore's Law and Dennard Scaling

In 1965, Gordon Moore predicted that the number of transistors in an integrated circuit will double approximately every two years [4]. Dennard scaling describes the way in which transistor power density remains constant as the transistors themselves shrink in size [1]. These phenomena combined allow for exponential transistor-density increases, and this has been exploited to produce exponentially higher performance in microprocessors year on year.

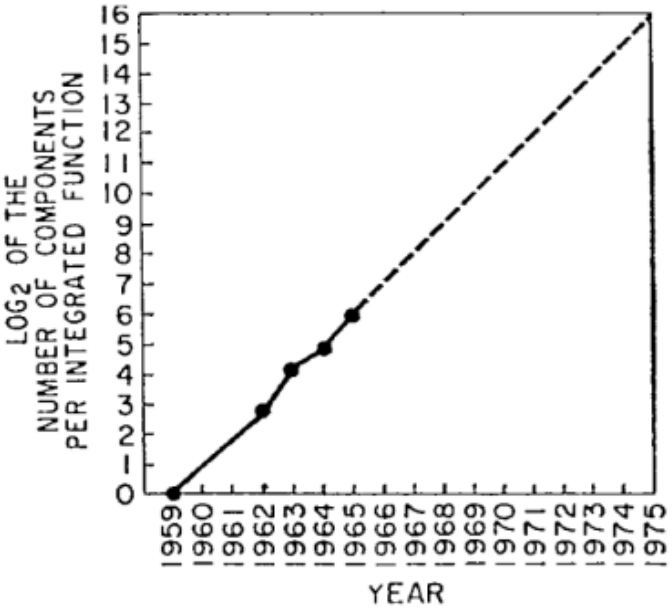

Figure 1.1: Moore's 1965 prediction. [4]

However, in more recent times, a breakdown in this scaling has been observed. In the past, microprocessor manufacturers have been able to offset the increased energy cost of faster transistor switching, resulting in higher and higher clock speeds. However, more recently, microprocessor clock speeds have remained relatively static. This is due to a breakdown in Dennard scaling for very small transistors, and has caused microprocessor

manufacturers to instead pursue increased performance by the use of multicore designs, as shown by figure 1.2.

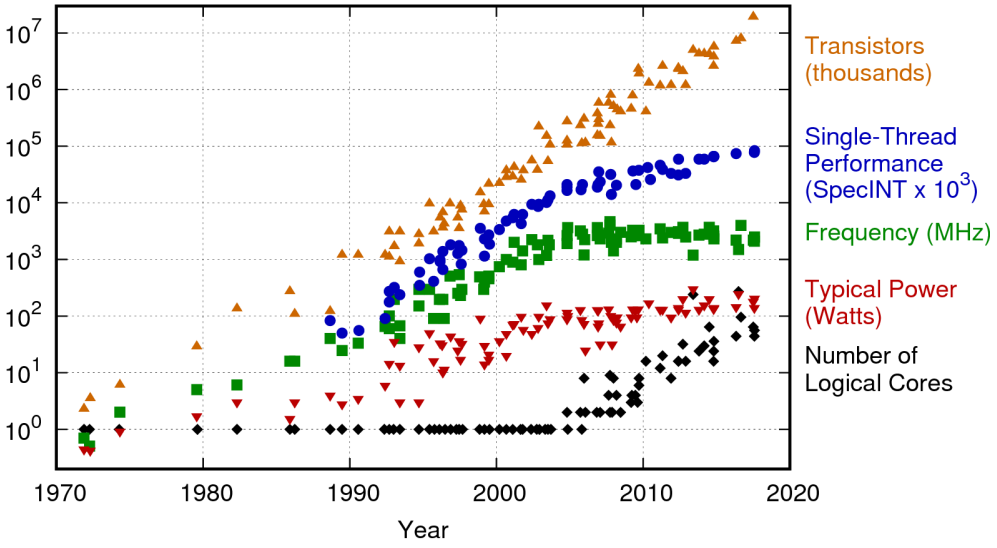

Figure 1.2: Trends in microprocessors since 1970. [5]

1.2 Dark Silicon

Dark silicon describes the way in which not all transistors in a microprocessor chip can be utilized simultaneously (usually because of power or heat limitations), leading to a percentage of the chip being under-utilized (or not utilized at all) [6]. This leads to a gap between the microprocessor's observed performance, and that predicted by extrapolating from historic performance gains.

It is predicted that with 22nm transistors, 21% of the chip will be dark silicon, with this rising to 50% at 8nm [2]. This prediction shows that dark silicon will become a serious limitation as transistor density grows, especially in areas where energy efficiency is a primary concern, as the transistors still consume resources, such as power and area, despite not being utilized.

Dark silicon also becomes a limiting factor with manycore devices. The limited parallelism of most applications results in a dark silicon gap when running with manycore devices. Again, [2] shows that beyond a certain number of cores the speedup achieved is negligible. This is another kind of dark silicon - the underutilization in this case is caused by the limited parallelism of the application being unable to exploit all of the cores of a device. This shows that while the shift to multicore designs is a solution to

the breakdown of Dennard scaling, it is not a long-term one if continued increases in performance are desired.

There have been several broad responses to this phenomenon. In [6], Taylor gives two pessimistic predictions regarding the future of silicon utilization. The first, the 'shrinking horseman', predicts that chips will begin to shrink as a result of the utilization wall. This would lead to an increase in cost per mm^2 of silicon, as design costs, test costs, marketing costs, etc. remain constant. As Taylor describes: "exponentially smaller chips are not exponentially cheaper".

The second, perhaps slightly less pessimistic, prediction in Taylor's paper is referred to as 'the dim horseman', and describes the under-clocking or infrequent use of general-purpose silicon in order to meet power budgets. While a better alternative than the shrinking of chips, this still causes issues as the chip is no longer operating at maximum capacity. Taylor outlines several options for the use of this 'dim silicon'. The first is the use of existing multicore architectures with some cores operating at a lower clock speed, or even being turned off intermittently. However, there are other 'dim silicon' approaches that make better use of the unutilized silicon area. One approach is to increase cache sizes, as cache memory is less power-dense than a processing core would be. This has a secondary benefit as well, reducing the likelihood of cache misses, thereby decreasing the number of power-hungry off-chip accesses.

Finally, Taylor also describes 'the specialized horseman'. This approach is to use the dark silicon area not for general-purpose computing, but for a large number of specialized cores, which would be much more energy efficient than a general-purpose core, allowing for increased energy efficiency at the cost of silicon area.

1.3 Heterogeneous Architectures

One such 'specialized horseman' approach is the use of heterogeneous architectures. These are computer architectures in which one or more general-purpose cores are coupled with special-purpose coprocessors, also known as hardware accelerators. A hardware accelerator is a specialised hardware circuit intended to perform a specific task more efficiently than a general-purpose processor. While historically hardware accelerators have been designed by hand for a specific application (e.g. encryption), this is infeasible when considering architectures with large numbers of accelerators. Thus an automated approach to generating hardware accelerators is required.

This is the approach taken in [7]. Here Razdan and Smith propose a

simple hardware accelerator architecture which avoids the need for memory synchronization and reduces the overheads involved in invoking a hardware accelerator. They describe a toolchain which is able to extract instruction streams to be 'outsourced' to an accelerator core (called a *programmable functional unit*, or *PFU*) after code generation.

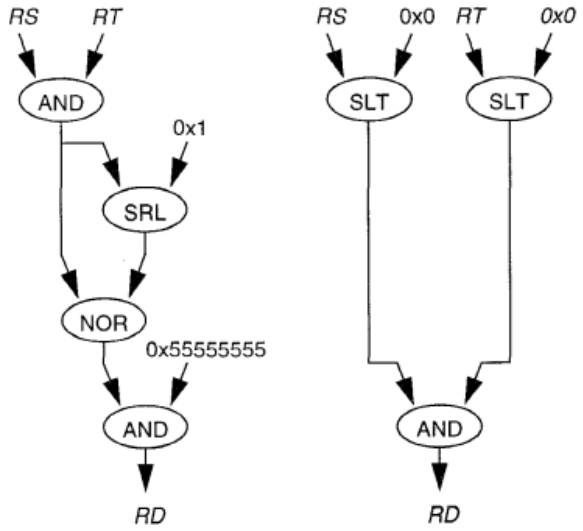

Figure 1.3: Two PFU optimization examples. Both sequences of operations can be evaluated in a single cycle, while the same sequences in MIPS R2000 instructions would take multiple cycles. [7]

Each PFU has at most two input operands and at most one output operand. In addition, the PFU-logic instructions (MIPS instructions that are candidates for translation into a PFU) cannot be memory-access or flow control instructions. This means that each PFU has an identical interface, and can be executed in a single cycle. This allows PFUs to be executed with a single instruction, *expfu*, in a single cycle, maintaining the fixed-format and single-cycle instructions of the MIPS processor's RISC instruction set.

1.4 Conservation Cores

While traditionally hardware accelerators have been used to speed up certain computations, they can also be used to achieve the same computational performance as a general-purpose processor at a fraction of the energy cost. For this reason, special-purpose cores are a subject of great interest to those trying to alleviate the dark silicon problem.

[8] introduces *conservation cores* (*c-cores*), which are hardware accelerators designed for this purpose. The paper outlines a method for generating c-cores for a given application. The first step is to identify 'hot' and 'cold'

portions of the application, using some form of profiling. 'Hot' code sections are those which are run frequently, and so are ideal candidates for c-cores. 'Cold' sections are run infrequently, and so are not ideal candidates, as the overhead involved in using a c-core would not be offset by the computation avoided by its use.

Once 'hot' portions are identified, a c-core can be synthesised for it. While previous hardware accelerators might have been hand-designed, such an approach is not viable if large numbers of c-cores are to be used, and so the paper outlines a method for automating the synthesis of these cores. A control-flow graph can be extracted from the code, and from this a state machine model can be constructed to perform the functionality represented in the code.

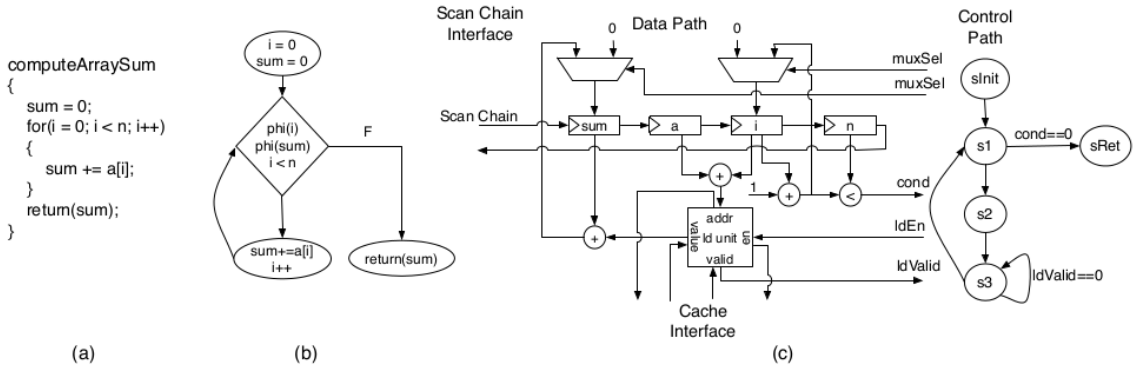

Figure 1.4: Conservation core example. [8]

One of the main issues with these c-cores is that memory synchronization restricts instruction-level parallelism and requires large numbers of pipeline registers (as each memory access marks a state-boundary). [9] attempts to alleviate this issue by introducing *selective de-pipelining* (SDP), in which memory accesses occur in 'fast states' while the rest of the processing done by the c-core proceeds as before, in 'slow states'. Signals can safely propogate through the 'slow states' without the need for latching on fast-state boundaries, while values loaded from memory are latched on fast-state boundaries as soon as they are available. An example of this process is shown in figure 2.4.

The GreenDroid project [10] takes the ideas of both [8] and [9] and attempts to apply them to the Android operating system and software stack. Mobile phone processors have vastly lower power budgets than traditional desktop processors, both to achieve long battery life and to reduce generated heat. Profiling of the Android software was used to identify the best portions of code to be converted into c-cores. As a result,

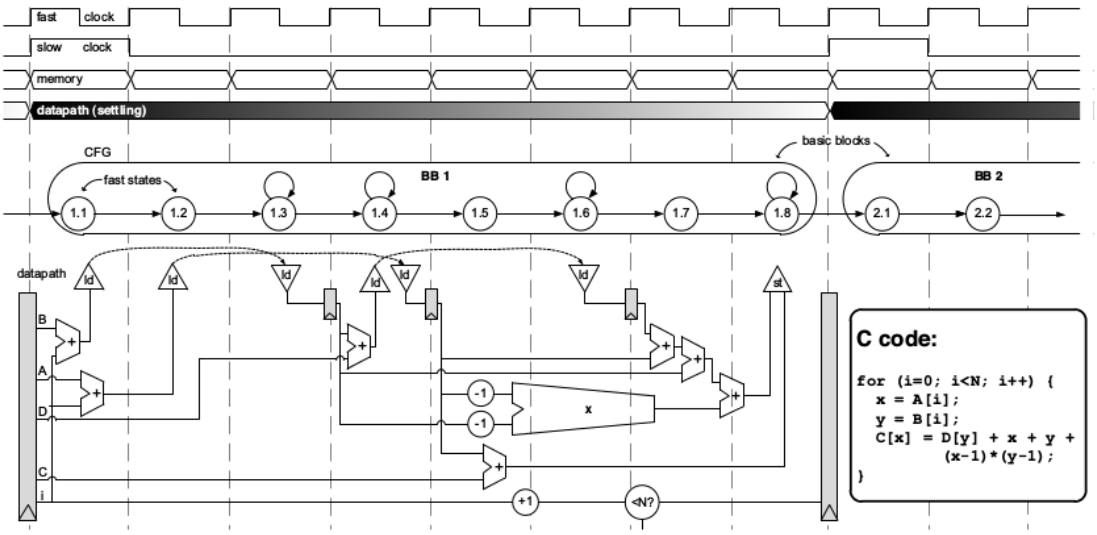

Figure 1.5: Selective de-pipelining example. [9]

c-cores account for over 90% of execution time, and if left idle when not in use, this leads to a significant reduction in energy consumption.

A similar approach is taken by Arnone in [11]. Here, Arnone outlines a method of generating accelerator cores for a stack architecture, resulting in both timing and power improvements. Two architectures for generated cores are described: composite and wave-core. Composite cores are simple state machines with instructions mapped to states, attempting to reduce logic area by reusing existing logic between states. Conversely, the wave-core architecture avoids the reuse of logic between states, reducing power density at the cost of increased logic area. In both cases the resulting accelerator core is more energy-efficient than the general-purpose processor is at the same task.

Here each core is a state machine generated from a basic block in the stack architecure's assembly code. States are divided between computation states and memory-access states. Each computation state is formed of a series of HDL statements which are direct translations of instructions in the stack architecture's instruction set. Between each computation state are one or more states in which memory is being accessed. Outputs from each state are latched in registers so that they are available for the next state to operate on.

Unlike conservation cores, these accelerators involve no flow control. This means that less use can be made of the cores (as the main processor must still perform flow control), but also that the implementation of state machines is less complex.

2 System Design

This chapter describes the architecture and design of the MicroBlaze system and the accelerator cores, and the methods of communication between them. It also describes the methods used to analyse the object code in order to extract accelerator cores, as well as the ways in which blocks to be extracted are selected automatically.

2.1 System Architecture

MicroBlaze [3] is a 32-bit RISC architecture, intended for implementation on an FPGA. It is highly configurable, as certain features (e.g. FPU, multiplier, barrel shifter) can be disabled if not required. This allows the architecture to be as minimal as is required by the application. Because of this, MicroBlaze is often used in embedded environments where power efficiency is a significant concern.

The architecture used in this project is a heterogeneous architecture consisting of a MicroBlaze processor connected to one or more hardware accelerator cores, via a control unit. Both the accelerators and the MicroBlaze core are connected to a local block RAM. Figure 2.1 shows this arrangement in detail.

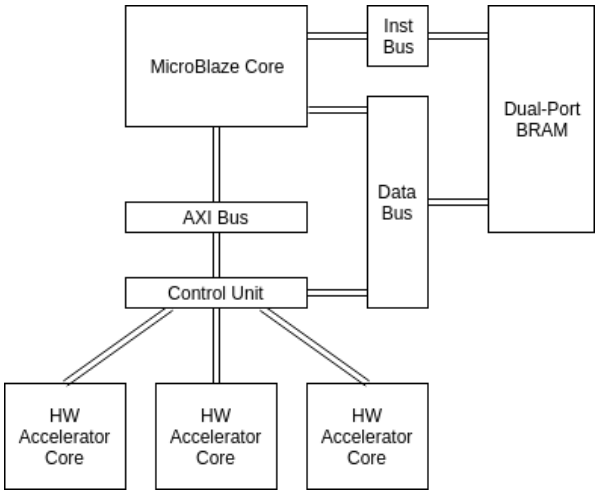

Figure 2.1: System architecture.

This reduces the number of signals required, as the accelerators themselves can use a simplified memory interface to communicate with the memory and the MicroBlaze core, rather than each having to separately implement the LMB and AXI protocols. The complexity of the LMB and AXI protocols, which provide for arbitration between competing master cores, are not required, as it is guaranteed that only one core (either an accelerator or MicroBlaze) will be executing at any one time.

To further simplify the design, the more advanced MicroBlaze features, such as the hardware multiplier, divider, and barrel shifter, are deactivated. This not only simplifies the system architecture, speeding up synthesis and simulation times, but also reduces the pool of instructions that need to be translated (as hardware multiply etc. instructions no longer appear in application code).

2.2 Hardware Accelerator Architecture

Each hardware accelerator is modelled as a sequential state machine, with a sequence of states determined by the structure of the code that the hardware accelerator is intended to replace. Each state machine block has a similar interface, which is shown in detail in appendix A. All transitions between states are performed on the rising edge of the CLK signal.

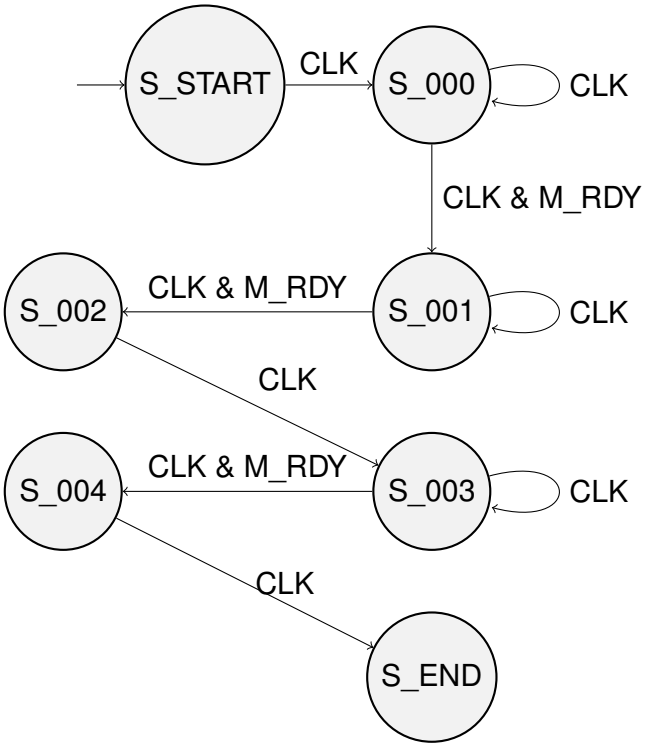

Figure 2.2: An example abstract state machine.

There are four kinds of states in the hardware accelerator model. The start state, which is the state the accelerator is in when activated, transfers inputs from the accelerator's register input ports into internal registers, for use during the computation. The end state is the state the accelerator moves to when the computation is complete. In this state, output values are transferred from the accelerator's internal registers to the register output ports, to be read by the MicroBlaze core. An example abstract state machine is shown in figure 2.2.

Between the start and end states are a series of computation and wait states. A computation state is derived from a series of non-memory-access instructions (e.g. `addi` (add-immediate) or `srl` (shift-right logical)). These instructions have translations in VHDL, allowing a circuit to be synthesised which implements the same functionality, but in parallel, as shown in figure 2.3. As such, while the instructions represented by a computation state might take several cycles to complete on a general-purpose processor, the hardware accelerator can always complete a computation state in a single cycle. The translations of MicroBlaze instructions used in the generation of accelerator cores can be seen in appendix B.

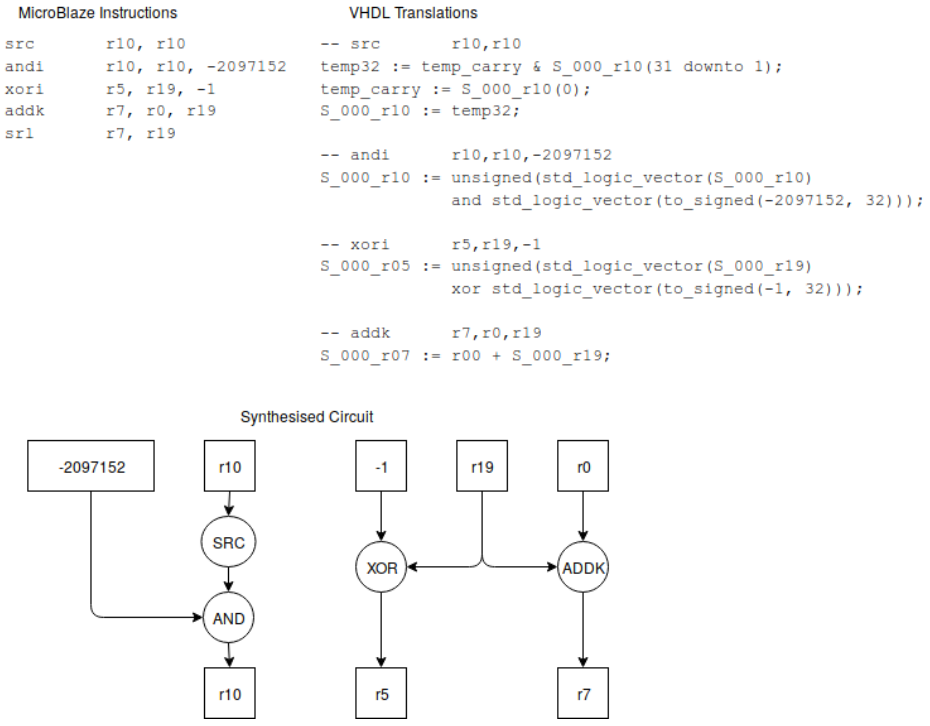

Figure 2.3: Comparison of original program instructions, VHDL translations, and final parallel circuit.

Each computation state is separated by one or more wait states, in which the accelerator either fetches data from or writes data to the local memory. Each wait state is associated with a single input/output instruction. On

entering a wait state, the accelerator sets up the necessary control signals for the memory access. Once the memory access is complete, the M_RDY signal is asserted and the accelerator continues into the next state.

2.3 Accessing Cores

In order for the generated hardware accelerators to be at all useful, a lightweight system to allow the MicroBlaze core to activate them and retrieve results is required. The accelerator cores are connected, via a controller module, to the general-purpose core via the AXI bus, which exposes the controller to MicroBlaze as a set of memory-mapped registers. These registers can then be written to and read from using the `sw` (store word) and `lw` (load word) instructions. Table 2.1 shows the locations and purpose of these registers.

Memory Location	**Description**
HW_ACCEL_PORT	Control register.
HW_ACCEL_PORT + 4	I/O for register R1.
...	...
HW_ACCEL_PORT + 120	I/O for register R30.
HW_ACCEL_PORT + 124	I/O for MSR register.

Table 2.1: Controller module memory-mapped registers. The symbol HW_ACCEL_PORT represents the memory location to which the controller is mapped.

There is no memory-mapped register for R31 because R31 is reserved by the compiler for storing the MSR system register when transferring it to and from the accelerator cores. This means that R31 will never be used by the application code, and so can be safely assumed to never be an input or output of a basic block (and by extension, an accelerator core). The transfer of MSR is necessary because this system register holds information required by some instructions, such as the carry flag, and so needs to be available to accelerator cores to use and modify.

2.3.1 Transaction Overview

Before activation, the controller module is in the READY state, waiting for the MicroBlaze core to pass inputs and select a core to be activated. The controller module remains in this state as the MicroBlaze core transfers inputs by writing to the appropriate controller module register. Once all

inputs have been transferred, the required core is activated by writing a numeric ID to the special control register. The MicroBlaze core then goes into a 'sleep' mode (by executing a `mbar` instruction) while the controller module activates the core. The controller then goes into the WAITING state.

Once the core has finished its computation, it signals the controller, which goes into the DONE state, and transfers any outputs to the controller module. The controller then asserts the MicroBlaze core's wakeup signal, and the MicroBlaze core reads the appropriate outputs from the memory-mapped registers. Finally, it reads from the special control register to reset the controller and the accelerator core, and the controller again goes into the READY state. The result of this read contains the MSR system register, the final output from the core. If necessary, the MicroBlaze core then writes the new value of MSR into the system register.

2.4 False Output Pruning

In order to ensure that register coherency is maintained when an accelerator core is activated, the registers that form the inputs and outputs of the basic block from which the core is derived must be known.

The naive approach to identifying the inputs and outputs of a block is to assume that any register that is written to is an output, and any register read before it is written to is an input. However, this leads to very conservative results, as temporary registers used in the basic block are assumed to be outputs, even when they are not used in subsequent blocks. This creates a large amount of unnecessary overhead, as temporary registers are transferred from a core when doing so is unnecessary.

In order to solve this problem, the analysis needs to take into account both how the registers are used (as defined by the ABI) and the context in which the block exists (i.e. what blocks come before and after it). The register usage convention, taken from the ABI, is shown in full in appendix C. R3 and R4 are used for returning values from subroutines, and so must be outputs from a basic block (if written to) if that block is the last block in a subroutine (i.e. the last instruction in it is a `rtsd` instruction). However, as R5 to R12 are volatile, they are not preserved when returning from a subroutine (as they will be restored by the caller). Therefore, if they are written to in the last block of a subroutine, they can be discarded as outputs.

Once the true inputs and outputs of the last block of a subroutine are known, the outputs of any block directly preceeding it in the program-flow graph in the same subroutine can be 'pruned' in a similar way. If a volatile register is an output of a preceeding block, but not a return-value register

(R3 or R4), nor an input of the following block, it is not a 'true' output and can be assumed to be temporary. This approach can then be applied recursively until the start of the subroutine is reached.

2.5 Hardware Accelerator Selection Heuristics

There may be a limited amount of resources available for use as hardware accelerators on any given die. Therefore, the system needs to be able to decide which code blocks should be implemented as hardware accelerators and which should be left to execute on the general purpose core. A set of heuristics were devised to allow the system to make this decision in an automated fashion.

2.5.1 Input/Output Overhead

Each accelerator has a number of input and output registers, and these registers must be written to (in the case of inputs) and read from (in the case of outputs) the accelerator core by MicroBlaze when a core is used. Each read or write is a transaction on the AXI bus, which has an associated overhead. The more inputs or outputs a particular core is, the more expensive its overheads will be. However, because the overhead for a block is a fixed cost, it must be considered relative to the size of the block itself. For example, a block with 10 cycles of instructions and 2 cycles of overhead has a higher relative overhead than a block with 100 cycles of instructions and 10 cycles of overhead.

Thus, relative I/O overhead is represented as below:

$$\frac{cost_{in}+cost_{out}}{cost_{in}+cost_{out}+\#cycles} \tag{2.1}$$

Where $cost_{in}$ is the cost (in cycles) of transferring inputs from MicroBlaze to the core, $cost_{out}$ is the cost (in cycles) of transferring outputs from the core to MicroBlaze, and $\#cycles$ is the number of cycles of instructions in the core.

If this value is greater than 0.5, this indicates that the given block would have more I/O overhead than it has instructions, making it a poor candidate for translation. Therefore, relative overheads >0.5 are undesirable, and a core becomes more optimal the closer to 0 its relative overhead is.

2.5.2 Potential Parallelism

The purpose of the hardware accelerator cores being generated is (in part) to extract parallelism out of sequential code. This is limited however by the maximum 'width' of any one sequence of computation instructions (i.e. instructions that operate only on registers, and not on memory). In the worst case, a single computation instruction bounded on both sides by memory-access instructions has a 'width' of 1, and cannot attain any parallelism whatsoever.

The core selection process should seek to maximise potential parallelism, as represented below:

$$\frac{\sum_{c \in C} \#c}{\#C} \tag{2.2}$$

Where C is the set of computation sequences in a basic block, with each $c \in C$ being an individual sequence. $\#c$ is the number of instructions in a given sequence, and $\#C$ is the number of sequences in a given block.

Intuitively, this heuristic represents the 'average width' of a basic block, and is a predictor of the level of parallelism that can be extracted from the block. As such, it is expected that this heuristic be proportional to the instruction-per-cycle (IPC) count of the resulting core.

There is no theoretical limit to the potential parallelism heuristic, because there can be arbitrary-length sequences of computation instructions in a basic block. Hence, to normalize this value, the maximum 'average width' in the program must be known. The potential parallelism values for all blocks can then be divided by this maximum, giving a value in the range [0, 1].

2.5.3 Memory Access Density

As a core can only access one location in memory at a time, each memory access instruction is converted into its own state. In this state, the core sets up control signals, address, and data (if writing) and waits for the memory to respond with the 'ready' signal, at which point it reads data (if reading) and continues to the next state. Because each of these wait states must be completed in sequence, a lower bound on the number of states a core can have (and therefore the number of cycles it can complete in) is imposed by the number of memory accesses in a basic block.

In order to take this into account when selecting basic blocks to be selected, the 'memory-access density' of each block needs to be calculated, as follows:

$$\frac{\#instructions_{io}}{\#instructions_{total}} \tag{2.3}$$

Where $\#instructions_{io}$ is the number of memory-access instructions in the block, and $\#instructions_{total}$ is the total number of instructions in the block.

A memory-access density of >0.5 indicates that the block has more memory-access in it than it has computation, and a memory-access density of <0.5 indicates the reverse. As the amount of memory-access limits the parallelism that can be extracted from a block, a lower memory-access density is desirable.

2.5.4 Combining Heuristics

The three heuristics described in this section describe different attributes of a basic block, and when selecting a block to be transformed into an accelerator core it is necessary to combine these into a 'hybrid' heuristic. Furthermore, this combination can be weighted to allow the preference for blocks of a certain kind to be controlled, e.g. if it is preferred that all selected blocks have low I/O overheads.

Because each heuristic is normalized, a simple weighted sum can be used to combine them:

$$a \cdot H_{io} + b \cdot H_{par} + c \cdot H_{mem} \tag{2.4}$$

Where H_{io} is the I/O overhead heuristic, H_{par} is the potential parallelism heuristic, and H_{mem} is the memory access heuristic. $a + b + c = 1$.

This gives a cost in the range [0, 1] for each basic block, allowing blocks to be compared when making selections for translation.

3 Experimental Methodology

This chapter outlines how the performance of the system described in the previous chapter was measured. In addition, this chapter describes how an estimation of the energy charateristics of the system are obtained. All testing was performed in the Xilinx Vivado toolsuite.

3.1 Benchmark Applications

A range of applications were selected for use as benchmarks in the evaluation of the MicroBlaze system. These applications were selected with the intention of being representative of embedded workloads.

- An implementation of the SHA256 algorithm [12].
- A 64-sample FFT application [13].

It is worth noting that there is a very small number of benchmarks - nowhere near enough to understand how this system will behave in the real world. However, it is hoped that even with the small quantity of tests, a case can be made for the benefits of such a configuration in an embedded system.

Each application was compiled using the Xilinx MicroBlaze GNU tools to produce a series of assembly instructions. The accelerator generation tool was then run to produce a number of state machines, generated as VHDL files. These were then used in a simulation, along with the MicroBlaze system, to run the compiled application. A range of core counts were used, ranging from 1 to 45. In addition, the application was run on the MicroBlaze core alone, to provide a baseline to which other measurements could be compared.

3.2 Measuring Application Speedup

Speedup is a measure of the difference in execution time between two configurations, and is calculated as below [14]:

$$\frac{cycles_{new}}{cycles_{old}}$$

The number of clock cycles taken for each application to complete was measured during the simulation, and from these values a measurement of speedup could be calculated relative to the baseline execution time. This was repeated for each configuration (number of cores, method of analysis, etc.).

3.2.1 Cycle Breakdowns

In addition to measuring the overall execution time of the application, the execution time spent performing specific tasks was also measured. This provided four values, giving a breakdown of the execution time of the application as below:

- Execution time spent on the MicroBlaze core (not interacting with accelerator cores).
- Execution time spent on accelerator cores.
- Execution time spent transferring values over the AXI bus.
- Execution time spent with the MicroBlaze core asleep after an accelerator core had finished executing.

3.3 Measuring Power Consumption

From the simulation, a Signal Activity Interchange Format (SAIF) file could be produced. This format describes the switching rates of signals in the system during the course of the simulation. This was used with the Vivado power estimation utility to provide a more accurate estimate of power consumption, taking into account the behaviour of the system. This estimate provided two measurements: one for dynamic power consumption, and one for static power consumption.

3.3.1 Dynamic Power

Dynamic power is the power consumed as capacitances are charged and discharged, when a transistor switches on or off [15]. As such, it is expected that the dynamic power of a circuit be related to the overall activity of a

system. A system with a high rate of transistor switching will consume more dynamic power than one in which transistor switching rates are low. It is expected that dynamic power consumption will decrease as accelerator cores are added (assuming the right blocks are selected for conversion), as the MicroBlaze core will be inactive while the cores are performing computations, reducing the overall amount of transistor switching occuring in the system.

3.3.2 Static Power

Static power, or leakage power, is power consumed by transistors even when they are in a steady state, due to the leakage of current through transistor gates or due to electron tunnelling [15]. Because static power consumption is unrelated to the actual activity of the circuit, and only its size, it is expected that static power consumption will increase as more accelerator cores are added, as each core adds more components to the system.

3.4 Measuring Parallism (Instructions-Per-Clock)

As one of the aims of this project is to extract parallel circuits from sequential code, it makes sense to measure the parallelism of the generated hardware accelerators. One such metric is instructions-per-clock (IPC). To measure IPC for each generated core, the number of cycles taken to execute the core was measured, as well as the number of instructions in the basic block from which the core was derived. IPC can then be calculated:

$$\frac{\#instructions}{\#cycles} \tag{3.1}$$

This gives the average number of instructions executed per clock cycle. For comparison, a RISC processor like MicroBlaze will generally have an IPC of 1, indicating that a single instruction is executed every cycle.

4 Results

This chapter outlines the results obtained from the benchmark applications used. Section 4.1 is a brief overview of the performance and power consumption measurements observed. Section 4.2 then shows execution-time breakdowns for the SHA256 benchmark application across all four selection heuristics, as this was the benchmark on which the system performed the best. Finally, section 4.3 identifies trends in the attributes of generated accelerator cores.

4.1 Speedup and Power Consumption

The SHA256 benchmark showed significant speedup, as can be seen from figure 4.1. The highest speedup was obtained by both the hybrid and overhead heuristics, and was over 1.3x, however even with as many as 28 cores, the speedup gained using the overhead heuristic remained above the baseline, at 1.1x, while the hybrid heuristic dropped below the baseline at 21 cores. This indicates that the overhead heuristic is the best of the four for producing a system that offers a performance increase with a large number of cores.

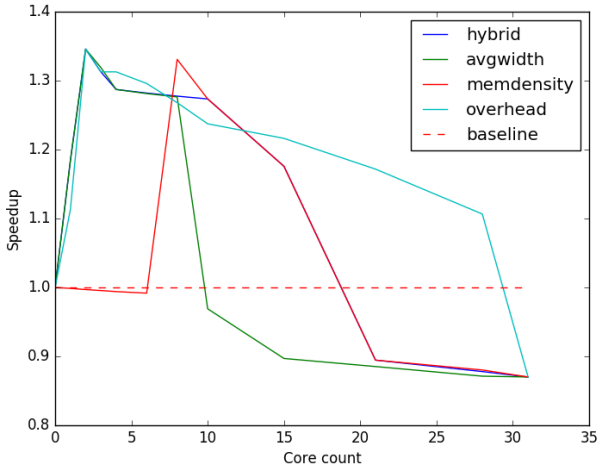

Figure 4.1: Speedup provided by accelerator cores for the SHA256 benchmark, by selection heuristic.

In contrast, accelerator cores in the FFT benchmark either slowed down the system or provided no change. When using the hybrid, overhead, or avgwidth selection heuristics with the FFT benchmark (figure 4.2), only a small speedup (1.1x) was observed at the highest point. The speedup then fell below the baseline with 10 cores. The memdensity heuristic performed worse, not offering any speedup whatsoever, and even falling to 0.5x with 45 cores.

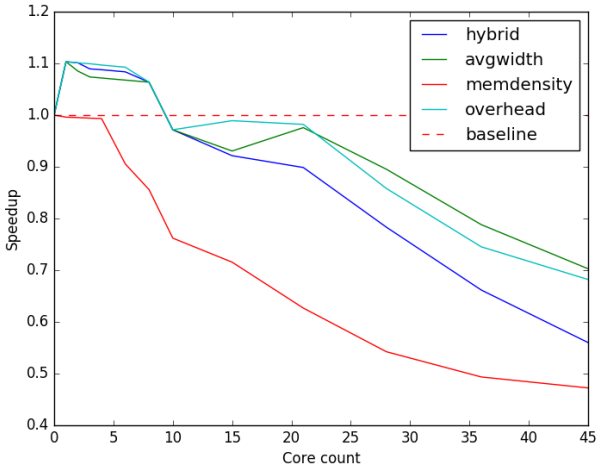

Figure 4.2: Speedup provided by accelerator cores for the FFT benchmark, by selection heuristic.

It was initially hoped that the use of hardware accelerators would reduce dynamic power consumption, but figures 4.3 and 4.4 show that this was not the case. The dynamic power consumption of the standalone MicroBlaze system in the SHA256 benchmark was around 500mW, while the consumption with the maximum number of cores (33) was 1900mW, regardless of selection heuristic used.

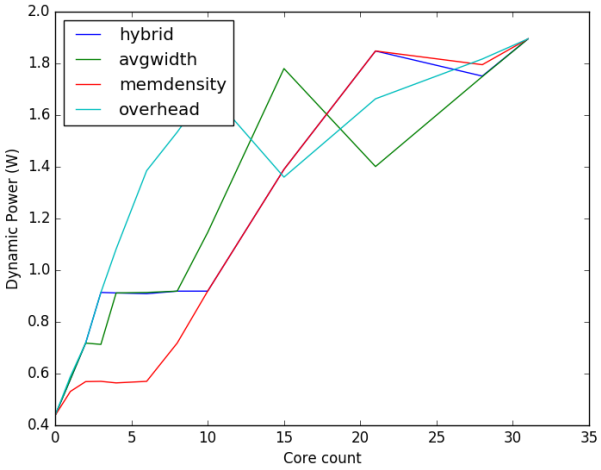

Figure 4.3: Dynamic power consumption for the SHA256 benchmark, by selection heuristic.

Interestingly, unlike with the SHA256 benchmark, there was a difference between the performance of the heuristics with the FFT benchmark. The overhead heuristic, which offered the best speedup, showed the worst increase in power consumption. Conversely, the memdensity heuristic offered the worst speedup, but the least increase in power consumption. This indicates that, for some applications, the core selection criteria are different depending on whether performance or energy efficiency is the primary concern.

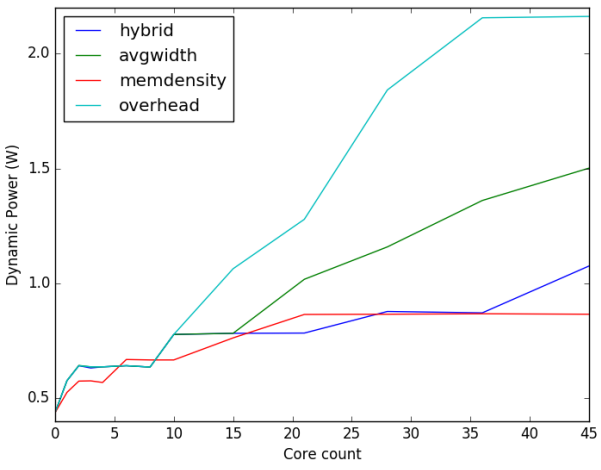

Figure 4.4: Dynamic power consumption for the FFT benchmark, by selection heuristic.

As expected, the static power consumed by the system increased as

more cores were added, as a larger system was being constructed. Figure 4.5 shows that for the SHA256 benchmark, the static power estimates increased generally, regardless of selection heuristic. The static power estimations for the FFT benchmark (figure 4.6) showed a similar variance between selection heuristics as the dynamic power estimations, with the overhead heuristic again showing the worst increase in power consumption, and the memdensity heuristic showing the least increase. This indicates that the cores selected by the memdensity heuristic (preferring cores with fewer memory accesses) are more power-efficient.

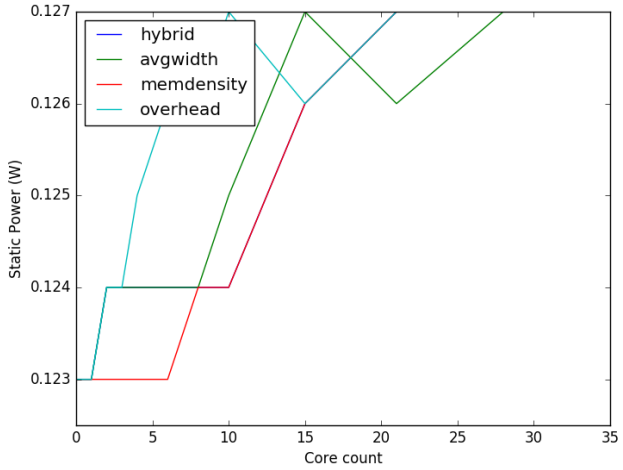

Figure 4.5: Static power consumption for the SHA256 benchmark, by selection heuristic.

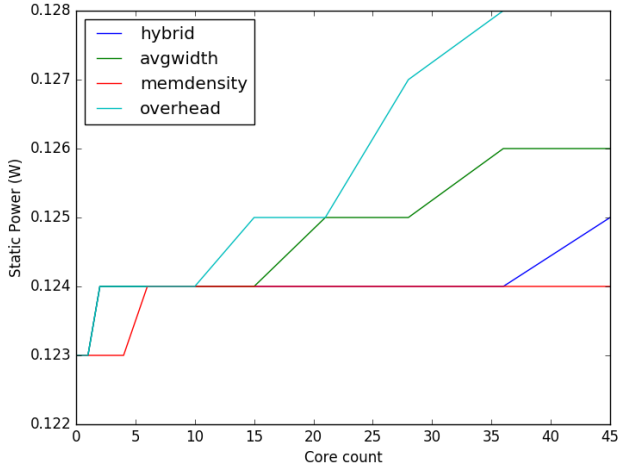

Figure 4.6: Static power consumption for the FFT benchmark, by selection heuristic.

4.2 Effect of Selection Heuristics on Performance

While initially the avgwidth heuristic (figure 4.7) halved the execution time spent on the MicroBlaze core with 2 to 8 cores, this performance improvement stopped once the 10-core mark was reached. This heuristic selects the initial cores well, but makes poor selections in the long run.

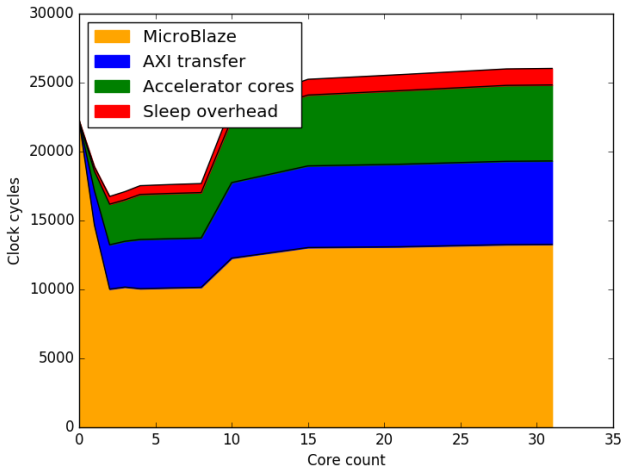

Figure 4.7: Execution-time breakdown for the SHA256 benchmark using the avgwidth selection heuristic.

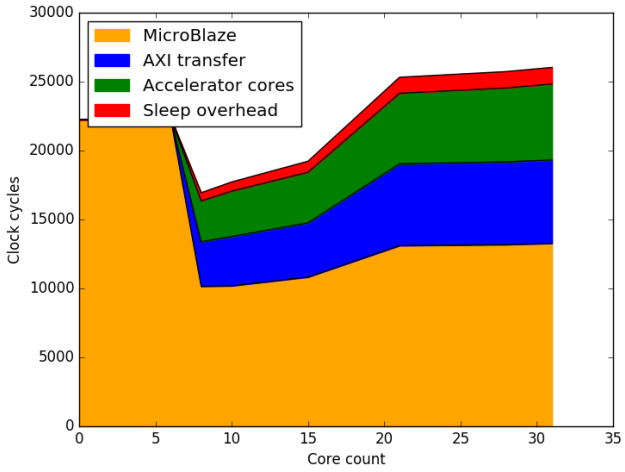

Figure 4.8: Execution-time breakdown for the SHA256 benchmark using the memdensity selection heuristic.

Conversely, the memdensity heuristic (figure 4.8) did not show a perfor-

mance improvement until the 8-core mark, with the cycle count sharply dropping at this point. This shows that, unlike the avgwidth heuristic, this heuristic does not initially select the cores that offer the best

Of the three individual heuristics, the overhead heuristic (figure 4.9) performed the best, with a sharp drop in the cycle count with 2 cores, and speedup not dropping below the baseline until 33 cores. This indicates that I/O overhead is a significant limiting factor in the accelerator core performance, and that optimising core selection to reduce this overhead yields significant performance improvements.

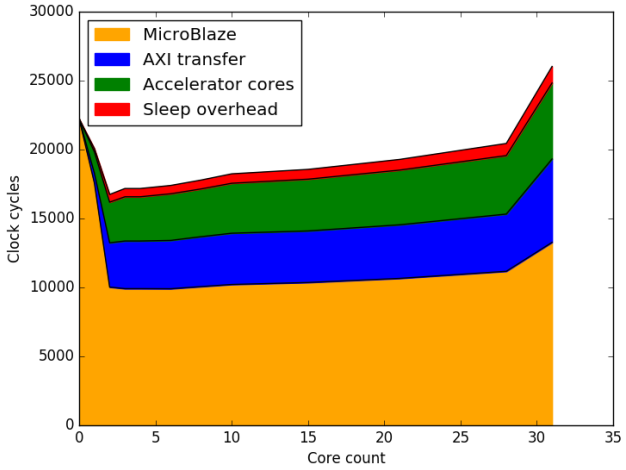

Figure 4.9: Execution-time breakdown for the SHA256 benchmark using the overhead selection heuristic.

As expected, the hybrid selection heuristic (figure 4.10) generated a configuration which performed well initially, halving the number of cycles executed on the MicroBlaze core. However, unlike the overhead heuristic, speedup fell below the baseline at 21 cores, rather than 33. Interestingly, this indicates that the I/O overhead factor dwarfs all other factors in core selection, making it really the only factor determining the performance of a given system.

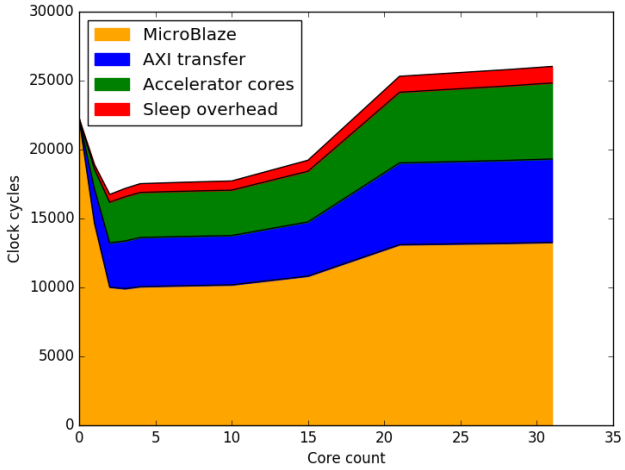

Figure 4.10: Execution-time breakdown for the SHA256 benchmark using the hybrid selection heuristic.

4.3 Trends in Generated Cores

This section describes trends observed in the generated hardware accelerator cores. As the overhead selection heuristic performed better than the others at selecting blocks to be translated, only the cores selected by this heuristic are considered here.

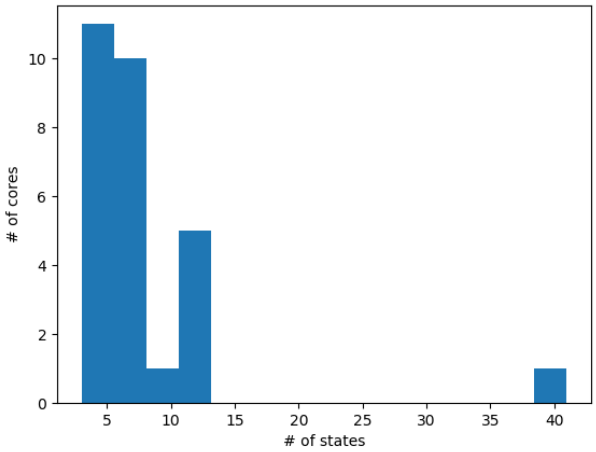

Figure 4.11: Distribution of states in hardware accelerator cores for the SHA256 benchmark application (28 cores).

Figure 4.11 shows that the state machines produced are generally smaller, with almost all of the cores having between 3 and 15 states.

Figure 4.12 shows the average instruction-per-clock measurements for the SHA256 benchmark as the number of cores selected increase. The IPC of initially-selected cores is high, but the average drops to 1 as the number of cores increases. This indicates that there are cores being selected with an IPC <1.

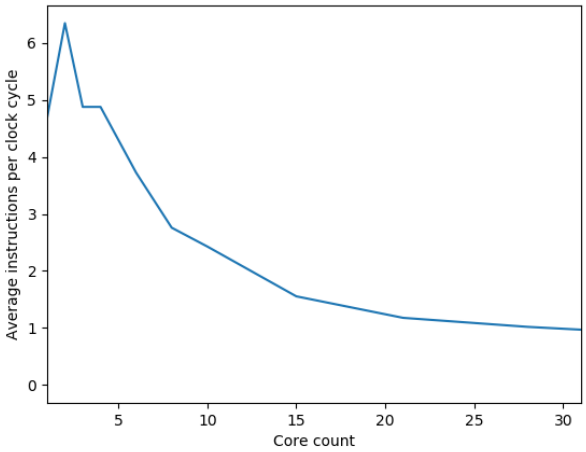

Figure 4.12: Average instruction-per-clock for the SHA256 benchmark application, with the overhead selection heuristic.

Figure 4.13 shows the average IPC for the FFT benchmark. Here the average IPC with 45 cores is twice that of the SHA256 benchmark with 28. This indicates that the cores generated for the FFT application are more efficient than those generated for the SHA256 application, probably due to a decreased number of memory accesses.

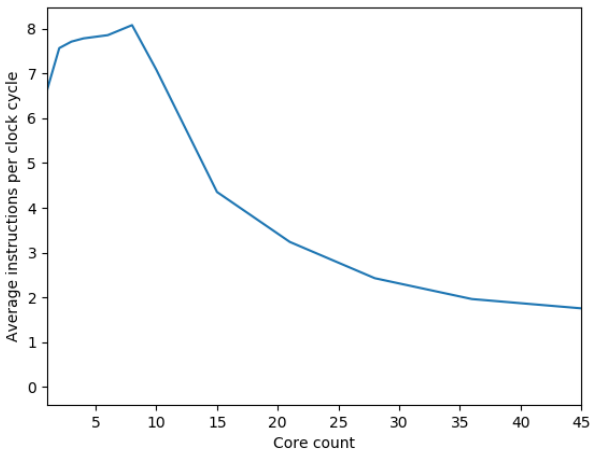

Figure 4.13: Average instruction-per-clock for the FFT benchmark application, with the overhead selection heuristic.

5 Discussion and Further Work

5.1 Discussion of Results

The performance increases observed were not as significant as was initially hoped. As seen in the previous chapter, the best method for selecting accelerator cores for this system is by minimizing I/O overhead. This is because the overheads involved in transferring each input/output parameter over the AXI bus is an expensive operation, costing 4 cycles per parameter.

The power consumption measurements observed are also disappointing, with the measurements making it clear that switching from a single MicroBlaze core to a MicroBlaze core with accelerators only increases power consumption. This is in contrast to Arnone's work in [11] and the GreenDroid project [10], which both show clear improvements in energy efficiency in moving to a coprocessor-dominated architecture.

While inaccuracies in the estimation of power consumption may be to blame for this gap, it could also be due to the difference between FPGA and VLSI technologies. For example, circuits that must be implemented using standard logic components such as lookup-tables in an FPGA may be optimized into more efficient implementations in a VLSI environment. However, the FPGA has the benefit of reprogrammability, so a small amount of inefficiency may be acceptable as a trade-off for being able to reprogram the architecture whenever a change is made to the software, avoiding the need for 'patching' accelerator cores as with [10].

In addition, it is clear from the execution-time breakdowns shown in the previous chapter that, even with high numbers of accelerator cores, a significant amount of execution time is still spent on the MicroBlaze core. In a coprocessor-dominated architecture there is always the risk that the processor will spend as much time managing hardware accelerators as it would executing the application code in a homogenous architecture. For this reason, methods of reducing the reliance of the accelerator cores on a general-purpose processor for management should be explored. Otherwise, a CoDA is unlikely to reduce power consumption.

5.2 Further Work

This section is split into two subsections. The first describes extensions to the architecture that could be implemented to offer better performance or energy efficiency. The second describes further work that could be done in evaluating this system and others like it.

5.2.1 System Architecture Improvements

It is evident from the results observed with testing this system that the input/output overheads are too great to offer significant performance increases. Therefore, it makes sense that ways to decrease these overheads should be investigated.

One such way is to try to avoid the transfer of registers between the MicroBlaze core and the hardware accelerators where possible, by allowing registers to be transferred between cores themselves. For example, consider two cores which execute back-to-back, with the MicroBlaze core only waking up to transfer registers from one to the other. In this scenario, the cores can be treated as a single unit; the MicroBlaze core will wake up to perform control flow (e.g. by executing a branch instruction), but will somehow signal to the cores that they are not to transfer register values to/from the MicroBlaze core, but between themselves. In such a case, the only AXI accesses required are those that acquire register values needed for the control-flow instruction, and those relating to controlling the cores. Such a system could dramatically reduce the number of expensive I/O operations taking place, as well as decrease the time that the MicroBlaze core spends active, improving both performance and energy efficiency.

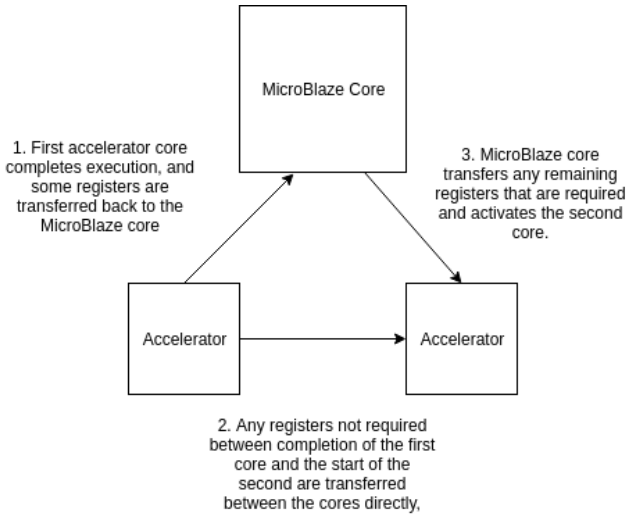

Figure 5.1: The direct transfer mechanism.

There are also improvements that could be made to the cores themselves. One such improvement involves reducing the time that cores spend waiting on memory. It is possible that a significant portion of memory accesses are not prerequisites to immediately following computations, but are instead used several cycles later. This could be used to pipeline memory accesses in accelerator cores, with a memory access being made at the same time as a computation is taking place. Such an improvement would reduce the impact of memory accesses, which are relatively expensive operations, on the performance of accelerator cores. Figure 5.2 shows two possible scenarios in which this technique could be used.

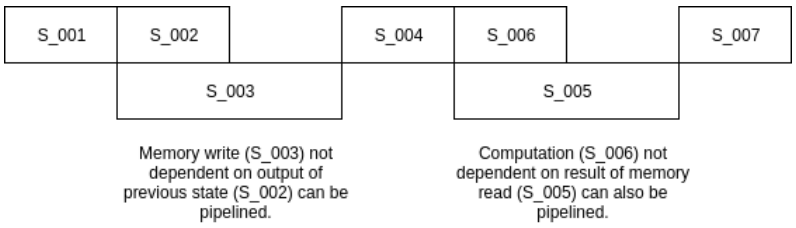

Figure 5.2: Pipelining of independent states.

5.2.2 Further Evaluation Work

It is difficult to make any broad claims about the system designed in this project because of the small pool of test applications used. For this reason, it is necessary to further investigate the performance of this system on a wide range of applications, including those used in the real world. In addition, the limited evaluation possible in a simulation environment resulted in inaccurate estimations of power consumption. A real-world evaluation using a physical FPGA could provide a more accurate estimation of the energy efficiency of this system.

Furthermore, there are improvements that could be made to the way that cores are selected. The heuristics described provide a reasonable (and computationally inexpensive) way to select accelerator cores, but this selection is done statically, and therefore cannot incorporate any information about the runtime behaviour of the system into the selection process. A selection method that involves some form of profiling, similar to that used in the GreenDroid project [10], could ensure that the blocks transformed into accelerator cores are those that are accessed frequently enough for the conversion to provide benefit.

Alternatively, a less conventional approach to core selection could be taken, for example using a genetic algorithm. While this could identify configurations with better performance and power consumption characteristics than could be found with the simple heuristics outlined in this work, it would

require a change in the way that the system is evaluated; FPGA simulation and synthesis is far too time-consuming a process to be amenable to such an approach.

Finally, while the focus in this investigation has been increasing performance by performing more computation in a smaller number of cycles, at the same clock speed, it is possible to instead improve energy efficiency by running the accelerator cores at a slower clock speed. This would maintain the same performance as without the accelerators, but would reduce energy usage by reducing the rate at which transistors in the circuit are switching. It would be worthwhile to measure how much the clock speed of the accelerator cores could be reduced (while maintaining the same level of performance) and determine how much energy efficiency could be gained from this.

5.3 Conclusion

It is apparent from the measurements observed that the performance and energy-efficiency increases offered by the accelerator cores is not as high as was hoped. As outlined in the previous section, it is possible that with some improvements, both to the design of the system and to its evaluation, greater performance improvements could be gained. In addition, a greater pool of test applications could help to identify trends that make some applications well-suited to this approach, and others not.

However, this work does show that the application object code can be analysed, basic blocks for extraction identified, and hardware accelerators generated, all in an automated toolchain. This ability to automate the generation of the architecture, coupled with the re-programmability of the FPGA, means that the design space for a system such as this can be explored with greater ease than if designs had to be made by hand and tested in a VLSI environment.

In addition, a range of methods of selecting accelerator cores have been outlined, and the various traits of systems generated from these different heuristics have been measured. It seems that the desired attributes of accelerator cores can be different depending on whether performance or energy efficiency are the primary concerns, and this tool provides an opportunity to explore this further with real-world evaluation.

Overall, while the observed performance and energy efficiency of this system was not as initially hoped, this work makes a strong case for further exploration of this architectural model in an FPGA environment.

A Hardware Accelerator Interface

Signal	Direction	Description
CLK	I	Clock signal.
RST	I	Reset signal. Active HIGH.
SEL	I	Core-select signal. Core is active when this is HIGH, and inactive when this is LOW.
DONE	O	Done signal. This goes HIGH once the core has finished its computation, and goes LOW when the core is reset.
M_RDY	I	Memory-ready signal. Signals to the core that a memory transaction has completed.
M_RD	O	Memory read strobe. Indicates that the core is reading a value from memory.
M_WR	O	Memory write strobe. Indicates that the core is writing a value to memory.
M_ADDR	O	Memory address. Address of word being accessed in memory.
M_DATA_OUT	O	Data out. Data written to memory is placed on this line.
M_DATA_IN	I	Data in. Data read from memory is placed on this line.
IN_Rxx	I	One or more input register lines. Inputs to the accelerator core are written to these lines from the MicroBlaze core.
OUT_Rxx	O	One or more output register lines. Outputs from the core are written to these lines once it has finished computation.
CARRY_IN	I	Carry signal input. Indicates the state of the carry flag at the start of the core's execution.
CARRY_OUT	O	Carry signal output. Indicates the state of the carry flag at the end of the core's execution.

Table A.1: Hardware accelerator input/output signals.

B Instruction Translations

Described here are the translations of MicroBlaze instructions to VHDL statements, used to automatically generate computational circuits from sequences of computation instructions. Due to time constraints, not all instruction translations were implemented; only those found in the two benchmark applications are present.

B.1 ADD - Addition

B.1.1 ADD

Computes the unsigned addition of rA and rB and stores the result in rD. The carry flag is set to 1 if the addition overflows, and 0 otherwise.

```
temp := ('0' & rA) + rB;
carry := temp(32);
rD := temp(31 downto 0);
```

B.1.2 ADDI

Computes the unsigned addition of rA and an immediate value and stores the result in rD. The carry flag is set to 1 if the addition overflows, and 0 otherwise.

```
temp := ('0' & rA) + unsigned(to_signed(imm, 32));
carry := temp(32);
rD := temp(31 downto 0);
```

B.1.3 ADDK

Computes the unsigned addition of rA and rB and stores the result in rD. The carry flag is unaffected and retains its previous value.

```
rD := rA + rB;
```

B.1.4 ADDIK

Computes the unsigned addition of rA and an immediate value and stores the result in rD. The carry flag is unaffected and retains its previous value.

```
rD := rA + unsigned(to_signed(imm, 32));
```

B.2 RSUB - Reverse Subtraction

B.2.1 RSUBK

Computes the unsigned subtraction of rA from rB and stores the result in rD. The carry flag is unaffected and retains its previous value.

```
rD := rB - rA;
```

B.3 CMP - Compare

B.3.1 CMP

Computes the unsigned subtraction of rA from rB and stores the result in rD. The MSB of rD is updated to show the actual relation of rA to rB as signed values.

```
rD := rB - rA;

if signed(rA) > signed(rB) then
    rD(31) := '1';
else
    rD(31) := '0';
end if;
```

B.3.2 CMPU

Computes the unsigned subtraction of rA from rB and stores the result in rD. The MSB of rD is updated to show the actual relation of rA to rB as unsigned values.

```
rD := rB - rA;
```

```
if unsigned(rA) > unsigned(rB) then
    rD(31) := '1';
else
    rD(31) := '0';
end if;
```

B.4 SEXT - Sign Extend

B.4.1 SEXT8

Sign extends the 8-bit value in rA and stores it in rD, by copying bit 7 of rA into bits 31-7 of rD.

```
rD(31 downto 7) := (others => rA(7));
rD(6 downto 0) := rA(6 downto 0);
```

B.4.2 SEXT16

Sign extends the 16-bit value in rA and stores it in rD, by copying bit 15 of rA into bits 31-15 of rD.

```
rD(31 downto 15) := (others => rA(15));
rD(14 downto 0) := rA(14 downto 0);
```

B.5 AND - Logical AND

B.5.1 AND

Performs the logical AND of the values in rA and rB and stores the result in rD.

```
rD := rA AND rB;
```

B.5.2 ANDI

Performs the logical AND of the value in rA and an immediate value and stores the result in rD.

```
rD := rA AND unsigned(to_signed(imm, 32));
```

B.6 OR - Logical OR

B.6.1 OR

Performs the logical OR of the values in rA and rB and stores the result in rD.

```
rD := rA OR rB;
```

B.6.2 ORI

Performs the logical OR of the value in rA and an immediate value and stores the result in rD.

```
rD := rA OR unsigned(to_signed(imm, 32));
```

B.7 XOR - Logical XOR

B.7.1 XOR

Performs the logical XOR of the values in rA and rB and stores the result in rD.

```
rD := rA XOR rB;
```

B.7.2 XORI

Performs the logical XOR of the value in rA and an immediate value and stores the result in rD.

```
rD := rA XOR unsigned(to_signed(imm, 32));
```

B.8 SR - Shift Right

B.8.1 SRL

Shifts the value in rA right by one bit, with a 0 being shifted into the MSB. The result is stored in rD. The bit shifted out of rA is placed in the carry flag.

```
temp := '0' & rA(31 downto 1);
carry := rA(0);
rD := temp;
```

B.8.2 SRA

Shifts the value in rA right by one bit, with the MSB being duplicated (keeping the sign bit). The result is stored in rD. The bit shifted out of rA is placed in the carry flag.

```
temp := rA(31) & rA(31 downto 1);
carry := rA(0);
rD := temp;
```

B.8.3 SRC

Shifts the value in rA right by one bit, with the carry flag being shifted into the MSB. The result is stored in rD. The bit shifted out of rA is placed in the carry flag.

```
temp := carry & rA(31 downto 1);
carry := rA(0);
rD := temp;
```

C MicroBlaze ABI Register Usage Convention

Register	Type	Description
R0	Dedicated	Hardcoded 0.
R1	Dedicated	Stack pointer.
R2	Dedicated	Read-only small-data-area pointer.
R3-R4	Volatile	Return values/temporaries.
R5-R10	Volatile	Passing parameters/temporaries.
R11-R12	Volatile	Temporaries.
R13	Dedicated	Read-write small-data-area pointer.
R14	Dedicated	Return address for interrupts.
R15	Dedicated	Return address for subroutines.
R16	Dedicated	Return address for trap (debugger).
R17	Dedicated	Return address for exceptions.
R18	Dedicated	Reserved for assembler.
R19-R31	Non-volatile	Must be saved across function calls (callee-save).

Table C.1: MicroBlaze ABI register descriptions [16].

Bibliography

[1] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, L. Rideout, E. Bassous and A. R. Leblanc, 'Design of ion-implanted mosfets with very small physical dimensions,' *IEEE Journal of Solid State Circuits*, vol. SC-9, pp. 256–268, Oct. 1974.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam and D. Burger, 'Dark silicon and the end of multicore scaling,' *IEEE Micro*, vol. 32, pp. 122–134, Jun. 2012.

[3] Xilinx Inc., *The Xilinx MicroBlaze soft processor core*, https://www.xilinx.com/products/design-tools/microblaze.html, Accessed: 2019-02-02.

[4] G. E. Moore, 'Cramming more components onto integrated circuits,' *Electronics*, pp. 114–117, Apr. 1965.

[5] K. Rupp, *42 years of microprocessor trend data*, https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/, Accessed: 2019-02-01.

[6] M. B. Taylor, 'Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse,' in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 1131–1136.

[7] R. Razdan and M. D. Smith, 'A high-performance microarchitecture with hardware-programmable functional units,' in *27th Annual International Symposium on Microarchitecture*, Nov. 1994, pp. 172–180.

[8] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson and M. B. Taylor, 'Conservation cores: Reducing the energy of mature computations,' *SIGARCH Computer Architecture News*, vol. 38, pp. 205–218, Mar. 2010.

[9] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson and M. B. Taylor, 'Efficient complex operators for irregular codes,' in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb. 2011, pp. 491–502.

[10] M. B. Taylor and S. Swanson, 'GreenDroid: Exploring the next evolution in smartphone application processors,' *IEEE Communications Magazine*, vol. 49, pp. 112–119, Apr. 2011.

[11] A. Arnone, 'Feasibility of accelerator generation to alleviate dark silicon in a novel architecture,' PhD thesis, University of York, Jun. 2017.

[12] Alain Mosnier, *SHA-256 Algorithm Implementation*, https://github.com/amosnier/sha-2, Accessed: 2019-03-02.

[13] Author unknown, *Fixed-Point Fast Fourier Transform Algorithm Implementation*, https://gist.github.com/Tomwi/3842231, Accessed: 2019-03-10.

[14] *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, Inc., 1996.

[15] EE Herald, *Low Power VLSI Chip Design: Circuit Design Techniques*, http://www.eeherald.com/section/design-guide/Low-Power-VLSI-Design.html, Accessed: 2019-04-28.

[16] Xilinx Inc., *Microblaze processor reference guide*, https://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf, Accessed: 2019-02-16, 2008.