The Master Thesis that became so successfull that Torbjørn and Kristian didn't need to express the title in an intelligent way

and still got them supervisor-jobs

Written by:
Torbjørn Langland
Kristian Klomsten Skordal

Supervised by:
Donn Morrison (Main supervisor)
Yaman Umuroğlu (Co-supervisor)

TDT4501 - Computer Science Specialization Project
Norwegian University of Science and Technology
Autumn 2014

Abstract

This is an abstract. It is currently abstract. It is way too abstract for you to understand as it is now.

A s of such, for your enjoyable reading, we will provide with the lyrics from the Lion King song "Be Prepared":

S car: I know that your powers of retention Are as wet as a warthog's backside But thick as you are, pay attention My words are a matter of pride It's clear from your vacant expressions The lights are not all on upstairs But we're talking kings and successions Even you can't be caught unawares So prepare for a chance of a lifetime Be prepared for sensational news A shining new era Is tiptoeing nearer

Shenzi: And where do we feature?

Scar: Just listen to teacher I know it sounds sordid But you'll be rewarded When at last I am given my dues And injustice deliciously squared Be prepared!

(talking) Banzai: Yeah, Be prepared. Yeah-heh... we'll be prepared, heh. ...For what?

Scar: For the death of the king.

Banzai: Why? Is he sick?

Scar: No, fool, we're going to kill him. And Simba too.

Shenzi: Great idea! Who needs a king?

Shenzi (and then Banzai): No king! No king! la–la-la–la-laa-laa!

Scar: Idiots! There will be a king!

Banzai: Hey, but you said, uh...

Scar: I will be king! ...Stick with me, and you'll never go hungry again!

Shenzi and Banzai: Yaay! All right! Long live the king!

All Hyenas: Long live the king! Long live the king!

(Full song again) Hyenas: (In tight, crisp phrasing and diction It's great that we'll soon be connected. With a king who'll be all-time adored.

Scar: Of course, quid pro quo, you're expected To take certain duties on board The future is littered with prizes And though I'm the main addressee The point that I must emphasize is You won't get a sniff without me! So prepare for the coup of the century (Oooh!) Be prepared for the murkiest scam (Oooh... La! La! La!) Meticulous planning (We'll have food!) Tenacity spanning (Lots of food) Decades of denial (We repeat) Is simply why I'll (Endless meat) Be king undisputed (Aaaaaaah...) Respected, saluted (...aaaaaaah...) And seen for the wonder I am (...aaaaaaah!) Yes, my teeth and ambitions are bared (Oo-oo-oo-oo-oo-oo-oo) Be prepared!

All: Yes, our teeth and ambitions are bared Be prepared!

Contents

1 Introduction **3**
- 1.1 Original Assignment Text . . . 3
- 1.2 Suggestion: Dark Silicon and Heterogeneous Systems . . . 3

2 Background **4**
- 2.1 Rise of Dark Silicon, and the four areas of solution . . . 4
  - 2.1.1 The Utilization Wall . . . 5
  - 2.1.2 The four horsemen: Approaches to the dark silicon problem 5
- 2.2 Heterogeneous Architectures . . . 7
  - 2.2.1 The Single-ISA Heterogeneous MAny-core Computer . . . 7
- 2.3 The Bitcoin Currency . . . 8
  - 2.3.1 Mining Bitcoins . . . 8
  - 2.3.2 The SHA-256 Hashing Algorithm . . . 9

3 Related Work **11**
- 3.1 heterogeneous processors . . . 11
  - 3.1.1 Energy Efficiency . . . 11
  - 3.1.2 Performance . . . 11
  - 3.1.3 Optimal architecture . . . 14
  - 3.1.4 Uncore . . . 15
- 3.2 Bitcoin Mining . . . 16

4 Architecture **19**
- 4.1 Hashing tile with DMA module . . . 19
  - 4.1.1 SHA-256 Hashing Module . . . 19
  - 4.1.2 DMA Module . . . 19
  - 4.1.3 Wishbone Arbiter . . . 20

5 Measurements and Evaluation **22**

6 Conclusion **23**

A Detailed DMA Description **25**
- A.1 DMA Architecture . . . 25
  - A.1.1 Overview . . . 25
  - A.1.2 DMA Controller State Machine . . . 26
  - A.1.3 DMA Request FIFO adapter . . . 29
  - A.1.4 Manual subtractor . . . 30
  - A.1.5 Load channel . . . 30

A.1.6 Store channel 32
A.1.7 Arbiter 34
A.1.8 Comparator 35
A.1.9 FIFO Data buffer 35
A.1.10 Interaction in the two-channel setup 35
A.1.11 Tweaks in the system 36
A.2 Verification and testing of DMA Module 36

Chapter 1

Introduction

1.1 Original Assignment Text

Add current assignment text here

1.2 Suggestion: Dark Silicon and Heterogeneous Systems

Chapter 2

Background

Insert smart intro here:
"I'm SMAAAART!!!"

2.1 Rise of Dark Silicon, and the four areas of solution

As Taylor[9] points out, transistor density on a CMOS chip continue to double every two years, according to Moore's Law. Native transistor speed also increases with a factor of 1,4x. Energy efficiency on the other hands improves only with 1,4x, and under a constant power budget, it will cause a 2x shortfall in energy budget to power a chip at its native frequency. The utulization of a chip's potential is thus falling expentionally by 2x per generation. If the power limitation were to be based on the current generation, then designs would be 93,75% dark in eight years. This gives the rise to the term "Dark Silicon", the chip must either be underclocked, or parts of it turned off in order to keep within a set power budget. This is essentially true for chips where traditional cooling no longer can be efficient enough to remove generated heat from a fully powered chip.

Hmmm.... sounds cliche

source needed

According to Dennard scaling, progress were measured by the improvement in transistor speed and count, while according to the new post-Dennard scaling the progress will now be measured by improvement in transistor energy efficiency. While reducing delays have been the previous focus, the focus will now be to utilize as much joule from the transistors as possible.

The transistion from single-core to multicore processors in 2005 was a direct response from the industry to this problem, but adding multiple cores does not circumvent the problem on a longer run. Multicore chips will not scale as transistors shrink, and the fraction of a chip that can be filled with cores running at full frequency is dropping exponentially with each processor generation. Large fractions of the chip will be left dark - either idle for a long time, or significantly underclocked. Hence new designs are required, where new architectural techniques "spend" area to "buy" energy efficiency.

2.1.1 The Utilization Wall

Considered optional for now. If written, should contain the details of why there is the Utulization Wall that leads to dark silicon.

Tror dette blir overflødig, men har lagt av plass sånn i tilfelle

2.1.2 The four horsemen: Approaches to the dark silicon problem

Taylor explains a taxonomy called "The four horsemen" [9]. These are four proposed responses that are emerging as solutions as one transition beyond the transitional multicore stop-gap solution. [8] When looking back, these responses appeared to be unlikely candidates from the beginning, bringing with them unwelcome burdens in design, manufacturing and programming. From an aestethic engineering point of view, none of them would appear ideal. Henceforth the term "Horsemen". But it can be seen from the success of complex multi-regime devices like MOSFETs that engineering as a field has an enormous tolerance for complexity if the end result is better. From this result, Taylor argues that future chips will apply not just one of these alternatives, but all of them.

The four horsemen are called The Shrinking Horseman, The Dim Horseman, The Specialized Horseman and The Dues Ex Machina Horseman.

The Shrinking Horseman

Instead of having dark silicon on the chip, one may simply shrink the chip itself. Taylor[9] views these "shrinking chips" as the most pessimistic outcome. Although all chips may eventually shrink somewhat, the ones that shrink the most will be those where dark silicon cannot be applied fruitfully to improve the product. These chips will rapidly turn into low-margin businesses for which further generations of Moore's law provide small benefit. Futhermore, there are other effects: Exponentially smaller chips are not exponentially cheaper, since mask cost, design cost and I/O pad aread cannot be amortized. Competition will most likely favor chips that utulizes dark silicon to improve overall product, causing chips that are only shrinked to sell at low market price, causing loss to the company. And last, but not least, exponential shrinking leads to exponential rise in power density, and chip temperature will thus follow suit. Meeting the temperature limit will reduce the scaling below the nominal 1,4x expected energy efficienty.

Nært overflødig?

The Dim Horseman

According to Taylor[9], as exponentially larger fractions of a chip's transistors become dark transistors, silicon area becomes an exponentially cheaper resource relative to power and energy consumption. Therefore new architectural techniques that spend area to buy energy efficiency is called for. Instead of shrinking silicon, one may consider populating dark silicon area with logic that is used only part of the time, and interesting new design possibilities occurs. The term "dim silicon" refers to techniques that put large amounts of otherwise-dark silicon area to productive use by employing heavy underclocking or infrequent use to meet the power budget. The architecture has to strategically managing the chip-wide transistor duty cycle to enforce the overall power constraint. Whereas early 90-nm designs such as Cell and Prescott were dimmed because

actual power exceeded design-estimated power, more increasingly more elegant methods are converging, that make better trade-offs. Among the dim silicon techniques are dynamically varying the frequency with the number of cores being used, scaling up the amount of cache logic, employing near threshold voltage (NTV) processor designs, and redesigning the architecture to accommodate bursts that temporarily allow the power budget to be exceeded, such as Turbo Boost and computational sprinting.

The Specialized Horseman

This is the use of dark silicon to implement a host of specialized processors[9]. They can be more energy efficient, or much faster than a general purpose processor. Programs are executed where it is most efficient. Unused cores are power- and clock gated in order to keep them from wasting energy. Specialized logic focuses on reducing the amount of capacitance that needs to be switched to perform a particular operation.

Sort of ambigious on its own. Consider removal of sentence

Specialization is already being realized today in forms of specialized accelerators that span diverse areas such as baseband processing, graphics, computer vision, and media coding. These accelerators enable orders-of-magnitude improvements in energy efficiency and performance, especially for computations that are highly parallel. It is expected to se a rise of systems with more coprocessors than general processors. Tayler refers to them as coprocessor- dominated architectures, or CoDAs.

One of the expected challenges it the so-called "Tower of Babel" crisis, as the notion of general-purpose computation is fragmented, and the traditional clear lines of communication between programmers and software and the underlying hardware is eliminated. For instance, CUDA for NVidia GPUs is not usable for similar architectures, such as AMD. Overspecialization problems between accelerators that cause them to become inapplicable to closely related classes of computation has been observed. In addition, adoption problems are also caused by the excessive costs of programming heterogeneous hardware (such as Sony Playstation 3 vs. Microsoft XBox), and there is always the risk that specialized hardware may become obsolete as standards are revised.

The Dues Ex Machina Horseman

Taylor notes that this one is the most unpredictable[9]. He uses the termology "Deus ex machina" from literature or theater, in which the protagonists seem increasingly doomed until the very last moment, when something completely unexpected comes out of nowhere to save the day. In the case for dark silicon, one deus ex machina would be a breakthrough in semiconductor devices. The required breakthrough would have to be very fundamental, making it possible to build transistors out of devices other than MOSFETs. There are physical limits to what can be done with leakage from MOFSET transistors, and transistors made of other devices can combat this. New transistors must also be able to compete with MOFSETs in performance. Tunnel field-effect transistors (TFET) and nanoelectromechanical system (NEMS) switches are examples on inventions that hint to order-in-magnitude improvements in the leakage problem, but they still fall short in performance.

I chose to not elaborate this one further

2.2 Heterogeneous Architectures

2.2.1 The Single-ISA Heterogeneous MAny-core Computer

The Single-ISA Heterogeneous MAny-core Computer is a proposed infrastructure by NTNU for investigating heterogeneous systems at all abstraction levels, as illustrated in figure [**?**]. The point is to create a flexible framework in which different heterogeneous processors can be created from a collection of processing elements and accelerators. The programming model is kept constant across SHMAC-instances, while underlying implementation changes. This way, software design space exploration is facilitated. It is a tile-based architecture with a mesh interconnect. All processor tiles implement the same ARM ISA and the same memory model, in order to achieve a common programming model.[2]. An illustration of SHMAC can be seen in figure [**?**].

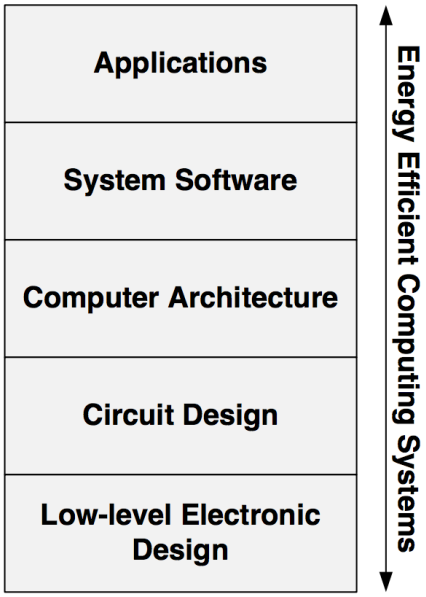

Figure 2.1: All levels of abstraction in computing systems, as seen in [2].

NTNU argues that the SHMAC-approach gives the right tools to reach the research goles outlined in the plan in [2]. For software research, SHMAC, makes it possible to explore substantially more diverse systems than the ones currently provided by the industry. Furthermore, SHMAC-architectures realized in FPGAs will be significatly faster than using simulators. It is expected that co-developing software and hardware will result in substantial cross-fertilization that gives insight into both hardware and software issues. And finally, micro- and macro-architecture components can be combined with novel transistor technologies and ASIC realizations to reach research goals at the lower abstraction levels.

PROBLEM: This paragraph is way too similar to original text. Must fix, one way or another.

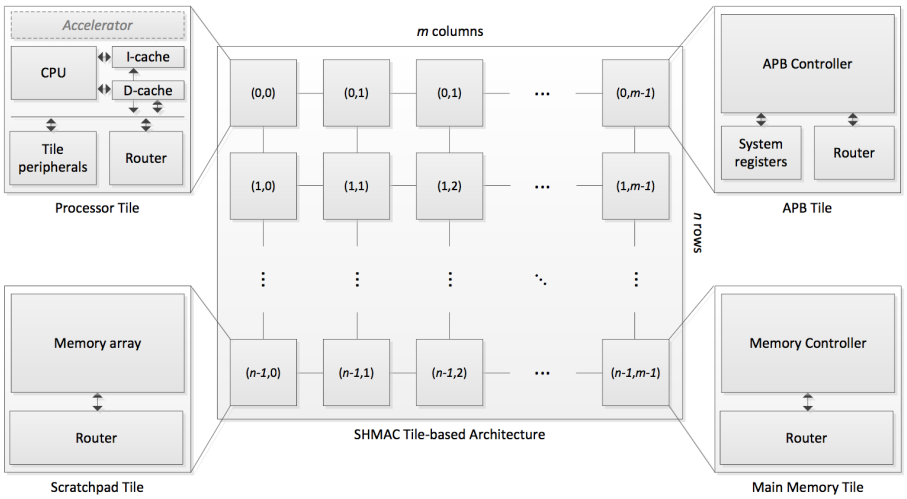

Figure 2.2: High-Level Architecture of ARM-based SHMAC, as seen in [2].

SHMAC Architecture

Work packages

2.3 The Bitcoin Currency

Bitcoin is a new currency, based on cryptographic stuffz. Eloquence

At the core of the bitcoin system is the block chain, a distributed linked-list consisting of blocks which contains the transactions that have been executed since the previous block was generated.

A block is only valid if the arithmetic value of the double SHA-256 hash of its header is below a certain target value. The target value is decided by the network and is set to such a value that on average six new blocks are generated per hour. [3]

2.3.1 Mining Bitcoins

The process of creating a new block for the bitcoin blockchain is often referred to as *mining*.

The process begins with the creation of a transaction that transfers the reward for generating the block into the account of the miner. This transaction is called the coinbase transaction. All transactions transmitted to the bitcoin network since the last block was generated are gathered and a merkle tree is constructed by combining the hashes of these transactions. Explain merkle trees

The root of the merkle tree is inserted into the header for the new block together with the hash of the previous block and various other fields specified by the standard. If the hash is below a target value, determined by the current network difficulty, the block is successfully mined and transmitted to the network.

2.3.2 The SHA-256 Hashing Algorithm

The SHA-256 algorithm is a member of a set of algorithms referred to as the SHA-2 standard. These are described in [**?**] and consists of algorithms for producing hashes with lengths of 224, 256, 384 and 512 bits. The algorithms use simple operations, limited to shifts, rotates, xor, and unsigned additions, common single-cycle operations for general purpose CPUs, in addition to a lookup-table of constants. This allows for high-speed implementations in both software and hardware. The different SHA-2 algorithms differ in how and with what parameters the various operations are invoked.

SHA-256 is the algorithm used in cryptocoin mining. It operates on blocks of 512 bits and keeps a 256 bits long intermediate hash value as state. Bitcoin uses a double pass SHA-256 hash, which first calculates the hash of a block of the data to be hashed and then hashes the hash of the first pass.

Before the first block is processed, the initial hash value is set to a predefined value. The entire message that is to be hashed is then padded by adding a 1 bit to the end of the message and then appending zeroes until the length of the final block is 448-bits. Then the length of the entire message, without padding, is added as a 64-bit big-endian integer to the end of the block.

Then, each input block is split into a 64 times 32-bit long expanded message block, where each 32-bit word W_j is defined according to the formula

$$W_j = \begin{cases} M_j & j \in [0, 15] \\ \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W - j - 15) + W_{j-15} & j \in [16, 63] \end{cases}$$

where M_j is the jth word of the input message block and the functions σ_0 and σ_1 are defined as

$$\sigma_0 = R^7(x) \oplus R^{18}(x) \oplus S^3(x)$$

$$\sigma_1 = R^{17}(x) \oplus R^{19}(x) \oplus S^{10}(x)$$

where the operator R^n means right rotation by n bits and S^n means right shift by n bits [1].

The Compression Function

The compression function is the core of the SHA-256 algorithm. It uses a look-up table of 64 constants, K_j, and the following functions when calculating the new intermediate hash values:

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\Sigma_0(x) = R^2(x) \oplus R^{13}(x) \oplus R^{22}(x)$$

$$\Sigma_1(x) = R^6(x) \oplus R^{11}(x) \oplus R^{25}(x)$$

Before starting the iterations with the compression function, the intermediate hash values from the previous message block are assigned to the variables a–h.

[1] Curiously, [**?**] defines the operator R as shift and S as rotate. We use the more intuitive definitions.

At the beginning of each iteration of the compression function, two temporary values are calculated:

$$T_1 = h + \Sigma_1(e) + Ch(e, f, g) + K_j + W_j$$

$$T_2 = \Sigma_0(a) + Maj(a, b, c)$$

The new hash values are then assigned as follows:

$$\begin{aligned}
h &\leftarrow g \\
g &\leftarrow f \\
f &\leftarrow e \\
e &\leftarrow d + T_1 \\
d &\leftarrow c \\
c &\leftarrow b \\
b &\leftarrow a \\
a &\leftarrow T_1 + T_2
\end{aligned}$$

The compression function is run 64 times, once for each word in the extended message block, W_j. Afterwards, the intermediate hash for the message is updated by adding the variables a–h to the corresponding values of the intermediate hash values from the previous message block.

When the final input block has been processed, the final hash is composed by concatenating the intermediate hash values [**?**].

Chapter 3

Related Work

3.1 heterogeneous processors

Kumar *et al* [6, 7, 5] explores the diverse possibilities offered by heterogeneous architectures. They prediction that core diversity will be of higher value than uniformity, and will offer much greater ability to adapt to the demands of the applications. They argue that the objective function can change over time, such as power conditions, application switches, or changes of demands within the application[6].

Note: Most likely, most of this content is too much, and probably too detailed. But I fill in too much at first, and cut down afterwards (don't have to reread the papers that way)

3.1.1 Energy Efficiency

In one experiment, Kumar *et al*[6] ran a simulation where they combined four generations from the Alpha family: EV4 (Alpha 21064), EV5 (Alpha 21164), EV6 (Alpha 21264) and a single-threaded version of EV8 (Alpha 21464). Only one application would run at the time, on one core, while the others were powered down. The architecture, with the said cores and their relative sizes to one another can be seen in [**?**].

14 benchmarks from SPEC2000 were used in this experiment, simulated using SMTSIM, and both energy and energy-delay were measured.

For this experiment, it was assumed that the system knew the needs of every application, and selected core statically. Results showed average energy savings of 32%, and average performance loss at only 2.6%, relative to the EV8 core. Kumar [**?**] also used this experiment to show that dynamic core switching outperforms the best static core selection. Simple heuristics would sample the cores at regular intervals, and execute core switch based on the results. The heuristics achieved up to 93% of the energy-delay gains compared to the best static selection.

Figure needs resizing

3.1.2 Performance

In another experiment, Kumar *et al*[7] showed that heterogeneity can be exploited to gain increased performance, for multithreaded workload. They point out that heterogeneous architectures are advantagous for two reasons.

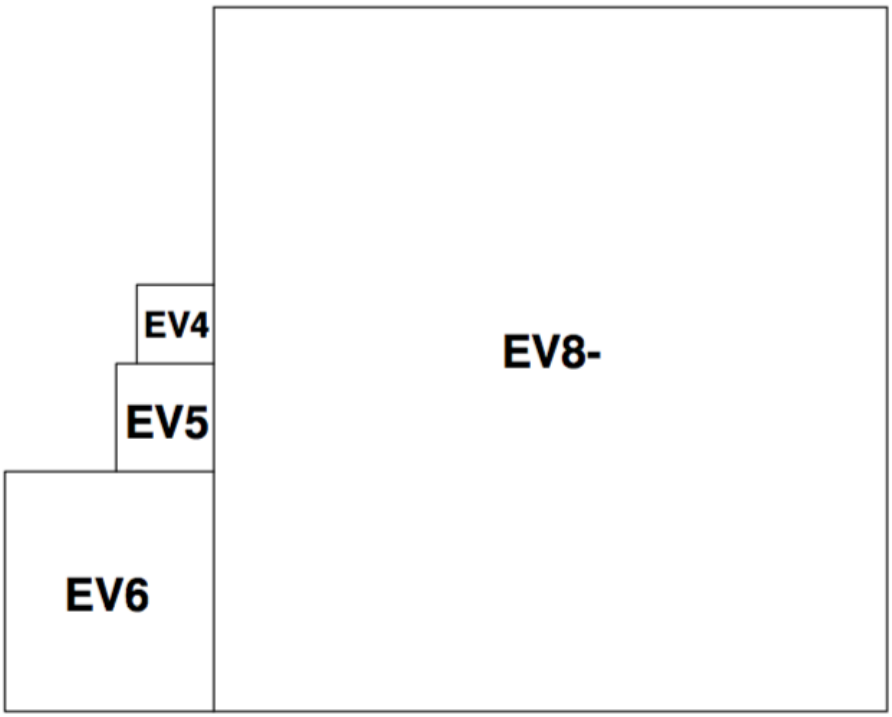

Figure 3.1: Cores used in [6], and their relative sizes.

Firstly, they offer efficient adaptation to application diversity, as applications differs in their resource needs. Some are compute intensitive and make good use of an out-of-order pipeline with high issue-width, while others may be memory-bound, and will underutulize advanced cores. They may perform almost as well on an in-order core with low issue-width[7].

Secondly, heterogeneous architectures offers more efficiently use of the die area for a given thread-level parallelism. A complex core can be replace by multiple smaller simpler cores. Since the process or thread level paralellism varies within most systems, a mix of cores that offers some large cores for hish single-thread performce, and some small cores with high throughput per die area, is a potentially attractive area. [7]

In contrast to the previous paper by Kumar *et al*[6], this experiment assigned multiple threads for multiple cores, and best global assignment was considered above best assignment for a selected application. EV5 and EV6 were chosen for this project, with one EV6 nearly equal to five EV5 in size. With the total area anvailable for the architecture assumed to be 100 mm$\hat{2}$, the space could have maximum 4 EV6 cores, or 20 EV5 cores Three EV6 cores and five EV5 were chosen for this experiment, with expectation that it would perform well over a wide range of available thread-level parallelism. A small number from SPEC2000 benchmakrs were chosen, with the focus on evaluating varying number of threads. SMTSIM and Simpoint were used for the simulation. Evaluation metric was weighted speedup, which in this context is the arithmetic sum of the

fix exponent

Mind-blowing, consider removal

individual IPCs of the threads constituting a workload divided by their IPC on a baseline configuration when running alone. In addition to performance gain, application response time within varying queue lengths are tested as well, to give insight on how well heterogeneous processors handles large queues, compared to best homogeneous systems.

Best static scheduling ensures that threads that are least affected by the difference between the EV5 and the EV6 are assigned to the EV5 processors, when all EV6 processors are busy. When the number of threads passes 4, the weighted speedup increases on the heterogeneous system, compared to a homogeneous CMP with 4 EV6. When the number of threads passes 13, a CMP with 20 EV5 performs better, though this can be changed with a different combination of EV5 and EV6 cores. This can be seen in figure [**?**]. Compared to 4 EV6 cores, the heterogeneous processor performed up to 37% better with an average 26% improvement over the configuration considering 1-20 threads. Compared to 20 EV5 cores, the performance was up to 2.3 times better, and averaged 23% better over that same range. [7] Using dynamic heuristics for core assignment further increased the performance.

Didn't show result figure from previous paper, consider if this leads to inconsistency.

Following 2 sentences are nearly ripoff from [7, **?**]

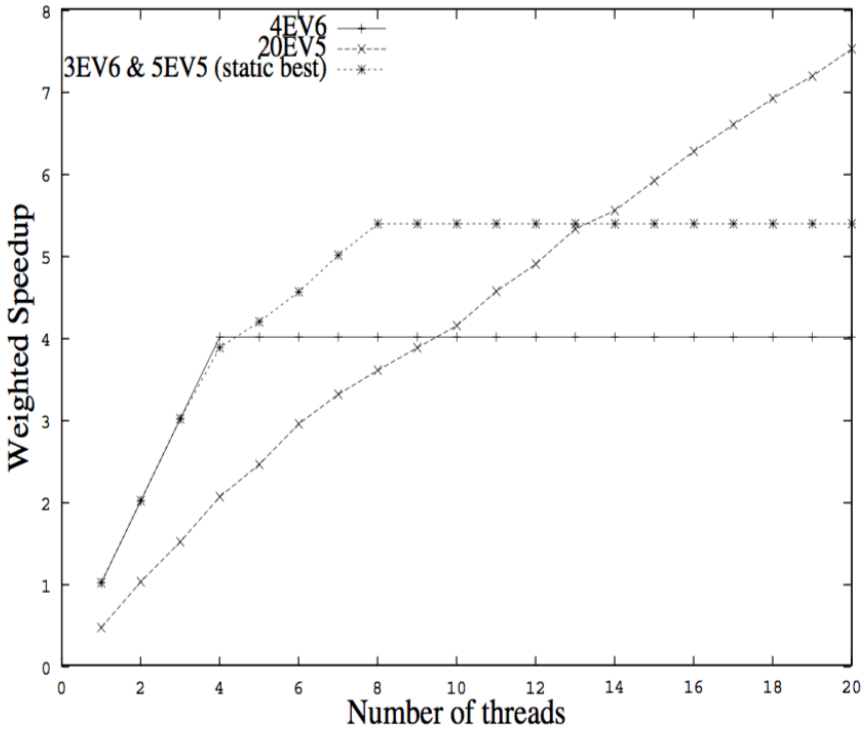

Figure 3.2: Benefits from heterogeneity - static scheduling for inter-thread diversity, as seen in [7].

For testing response time in an open system, and how various que length affects it, jobs with an average distrubition of 200 million cycles were generated were created randomly and executed. Then different mean job arrival rates

with exponential distribution were simulated. Testing revealed a great difference between saturation for a homogenous system with 4 EV6, and the heterogeneous processor. For the former, the unbounded response time is seen as the arrival rate approaches its maximum throughput around 2 jobs per 100 million cycles. From there, the run queue became infinate. The heterogeneous system remained stable well beyond that point. The degredation was also more graceful under heavier loads than for homogeneous processors. This can be seen in figure [**?**].

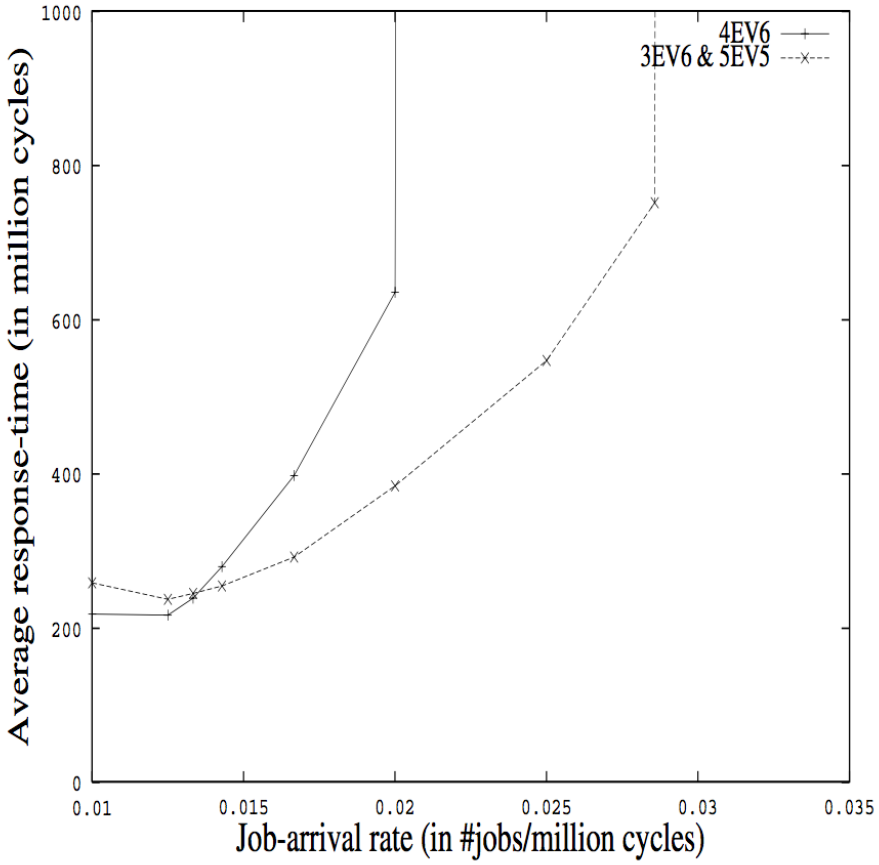

Figure 3.3: Limiting response-time for various loads, as seen in [7].

3.1.3 Optimal architecture

In a third experiment, Kumar *et al*[5] has taken a closer look at what is good heterogeneous design. They argue that while the previous experiments gave increased energy efficiency and performace, a heterogeneous system should be designed from scratch, instead of using pre-existing core designs. While they surpassed homogeneous designs, pre-existing cores failed to reach the full potential of heterogeneousy for three reasons: They present low flexibility in choices, the core choices remain monotonic, and third, best heterogeneous design are

composed of specialized core architectures[5]. But also, if pre-existing cores are not used, additional costs in design, verification and testing must be evaluated, to see if the benefits is worth the cost.

Kumar *et al* made three significant contributions in this experiment. First, benefits of heterogeneity in power and area efficient architectures is re-evaluated, with new benefits and higher gains shown. Secondly, methodologies for for arriving at good heterogeneous designs are demonstrated. Thirdly, a number of key principles critical to effective design of future chip multiprocessors are identified.

Several conclusions were derived: The most efficient heterogeneous multiprocessors were not constructed from cores that make good general-purpose uniprocessor cores, or cores that would appear in a good homogeneous multicure architecture. Each core was individually tuned for a class of applications with common characteristics. The results are usually non-monotonic processors. And performance advantages of heterogeneous multiprocessors, including non-monotonic also holds for completely homogeneous workloads. In those cases, the diversity across different workloads are exploited.

Itemize or cut down at this point

Simplification assumptions dropped

A fixed number of four cores was used for this experiment. In addition to testing the various core builds, the combinations were tested against varying area and power constraints, with their performance compared to the best homogeneous processors. Static mapping were mainly used for this experiment, but few tests with dynamic mapping showed further increase in performance, since a thread could be moved around to most suitable core at any phase of its execution.

Just a head-note. Remove when agreed or fixed

It turned out that the more constricted the anvailable power or area, the more gain it was with custom heterogeniety, as homogeneous multiprocessors could only perform any task at maximum if the power budget was generous. But as designs become more aggressive, one will want to place more cores on the die, and power budgets per core will likely tighten more severly. Heterogeneous designs dampen the effects of constrained power budgets [5].

Any best heterogeneous design for the different tested power and area constraint were very different from best homogeneous design, underlining the need to design heterogeneous processors from a clean slate, rather than modifying an existing core design. Monotonous designs (for instance, different generations of cores from same family) does not exploit heterogeneity equally well. For instance, the benchmark *mcf* from [6] were mapped on EV6 or single-threaded EV8-, in spite of having low ILP. The reason was the larger caches of these cores, causing the advances capabilities of these larger cores to be underutulized. Summed up: Heterogeneous multicore processors designed from clean slate gives better performance. Blegh!...

In lack of better word to finish off this mind-boggling work...

3.1.4 Uncore

Gupta *et al*[10] ran experiments with a heterogeneous multiprocessor, with the goal of measuring the impact of "uncore" power on the energy efficiency of heterogeneous multicore processors. In this experiment, they focused on the use of client devices, where energy is a premium resource and the workload profiles are diverse. The uncore is a collection of components of a processor, not in the core, but essential for core performance. Some of these components are the last

level cache (LLC), integrated memory controllers (IMC), on-chip interconnect (OCI), power control logic (PWR), etc. With growing cache size and integration of various SoC components on the CPU die, the uncore is becoming an increasingly imporant contributor [10].

If a core has varying levels of sleep states, where the "deeper the sleep", the less power used, then uncore will be as active as the most active core. If three cores are idle (or on low activity, and one core is on highest activity, then the uncore will be on the highest activity. If for a task, the choice is between a big fast core, or a more power efficient small core, and the small core is chosen, the ucore will stay active for longer time, impacting the possible energy saving. This can be seen illustrated in figure [**?**].

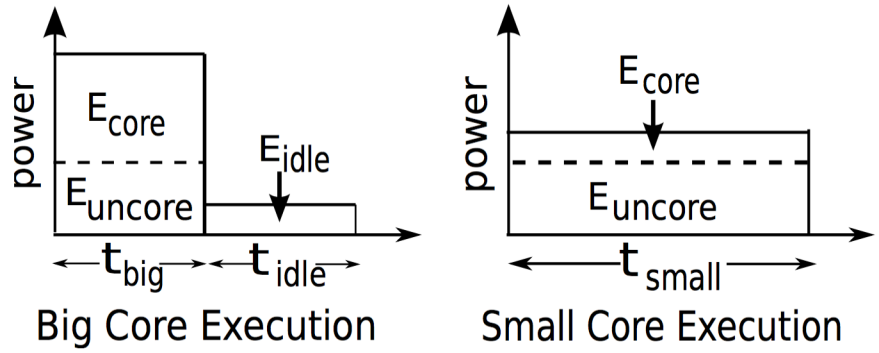

Figure 3.4: Effect of uncore power on the energy efficiency of heterogeneous cores, as seen in [10].

Skipped client workload description WARNING

The analysis considered two uncore conficurations: fixed and scalable. The first one used the same uncore subsystem for both big and small cores. The second modelend an uncore where certain components were turned off or powered down when moving to small cores. Examples are fewer memory channels, controllers, or smaller caches used.

Figure [**?**] shows the energy saved when going from big to small core, with both fixed and scaled uncores. It is clear that without scaling of the uncores, energy saving is reduced, even lost in some cases. Figure [**?**] shows the relative contrivution of core and uncore energy consumption for all the applications during big core execution, on a fixed uncore configuration.

It is clear that it is important to take uncore power into account for scheduling operations, and design of scalable uncore design is motivated, to obtain large gains from heterogeneous multicores.

3.2 Bitcoin Mining

Suggestion: Age of bespoken silicon-paper-thingy, + those FPGA things?

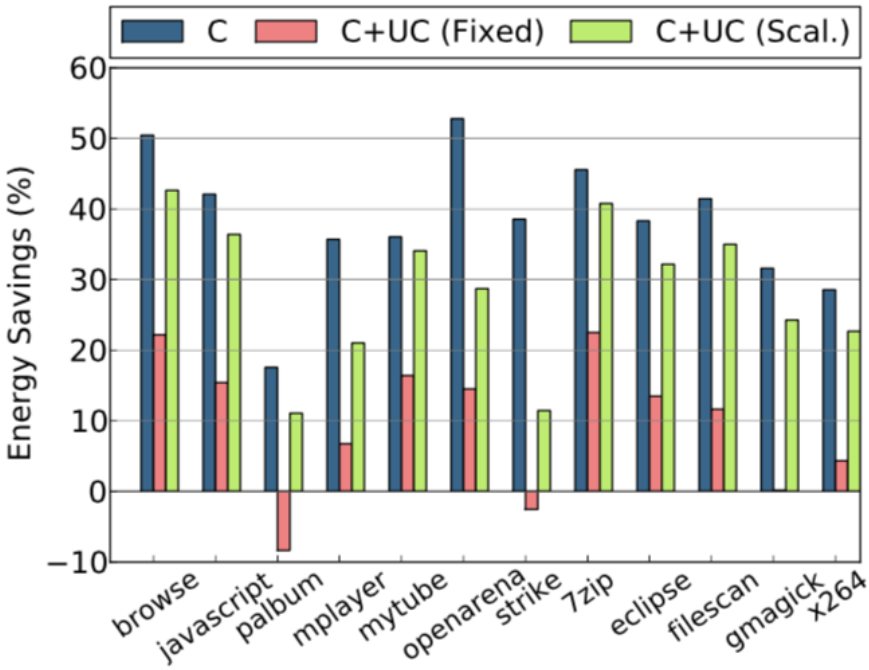

Figure 3.5: Energy savings of using small cores, with core-only savings (C), and with SoC-wide savings (C+UC), with a fixed uncore, and with a scalable uncore, as seen in [10].

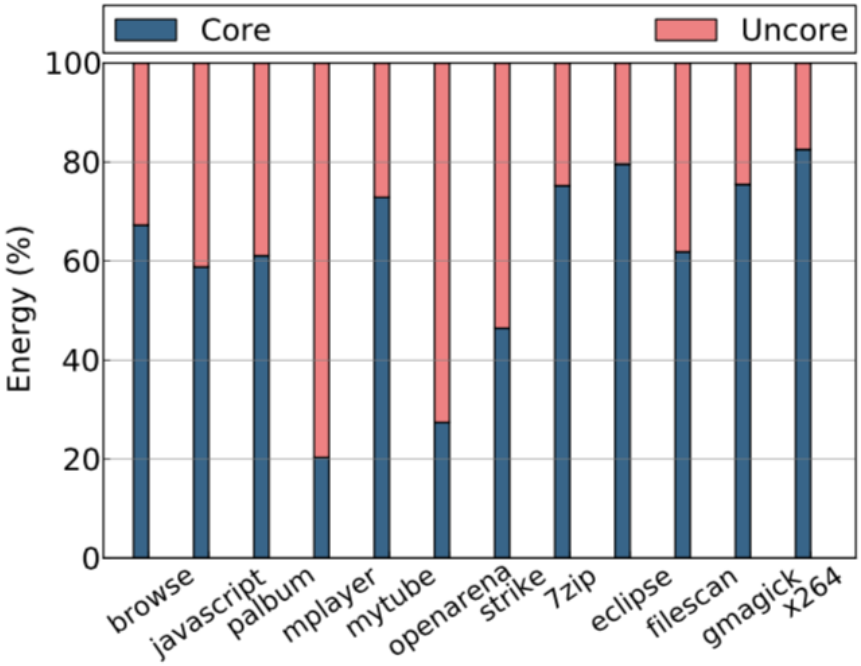

Figure 3.6: Core and uncore energy contricution for big cores and fixed uncores, as seen in [10].

Chapter 4

Architecture

This chapter introduces the architecture of the new tile that has been developed. Blegh!...

4.1 Hashing tile with DMA module

For this project, a new tile has been made for SHMAC, which can be seen in figure 4.1.

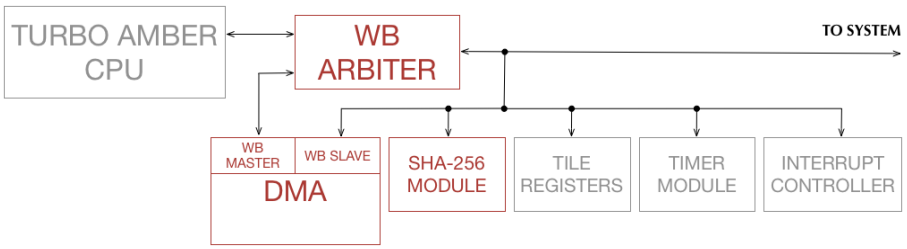

Figure 4.1: New SHMAC-tile, consisting of a unique SHA-256 hashing tile, a DMA Module and a Wishbone Arbiter.

The tile is based on the Turbo Amber tile, which consists of a Turbo Amber core, tile registers, timer module and interrupt controller. The new components for this tile are the SHA-256 hashing module, the DMA Module, and the Wishbone Bus Arbiter, all marked in red.

4.1.1 SHA-256 Hashing Module

HÆSJÆSJÆSJÆSJ!!! KRISTIAAAAAAAAAN

4.1.2 DMA Module

Originally made for the project during the Autumn 2014 at NTNU, the DMA Module has been modified further for use in this project. It can be seen in figure 4.2. would need a reference

The DMA module was made as part of the project to test if separate data transfers with a DMA module could improve performance and energy efficiency

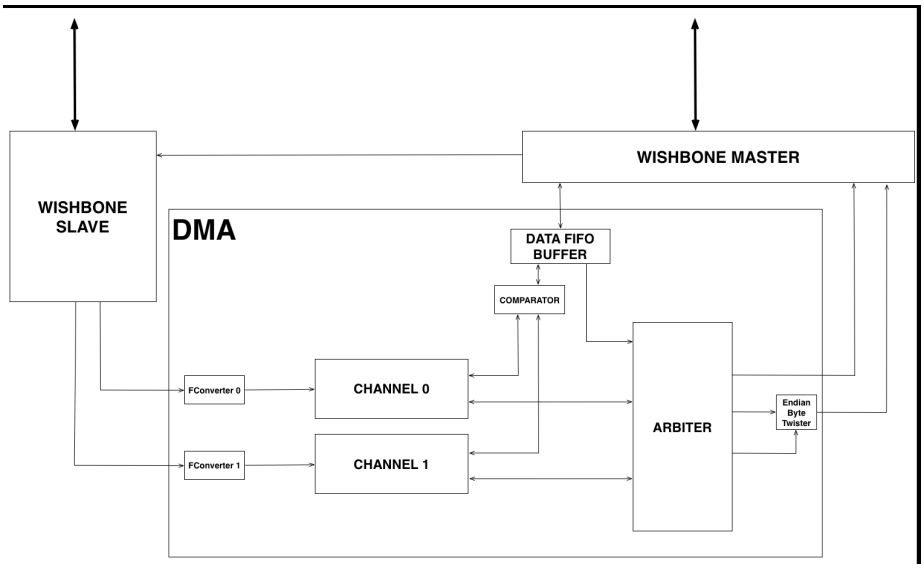

Figure 4.2: DMA Topview, including Wishbone interface.

in the Hashing process, leveraging the tile CPU for other work and specializing this module more efficiently for data transfer.

The DMA consists of three submodules: Wishbone Slave for receiving transfer requests, Wishbone Master for interfacing data transfer with the Wishbone bus system, and the DMA module itself, which generates the individual transfer commands.

The Wishbone Slave consists of three registers for each DMA Channel, making it six in total. The registers are used for base source, base destination and detailing the transfer request. When request is activated, the selected channel receives data from the slave, and executes the transfer. An arbiter selects commands from the channels if both are active. Every single command from the channels are passed on to the Wishbone master, which executes the command. The command is either a load or a store command. Any received data from a load command is passed on to the correct channel, and when a channel if finished, the imformation is passed from the Wishbone Master to the Wishbone Slave, so it can modify its registers and interrupt the CPU.

In addition for this project, the DMA Module also has submodule for endian byte switching, since the results from the SHA-256 hashing process is stored with lowest endian byte first. Switching the bytes instantly as they pass through the submodule should be more efficient than in software, since the software solution would require to load the data and shift the bytes for each existing byte.

The 14th commandment: Thou shall satisfy Kristian. The 13th commandment: Thou shall satisfy ME!!!

Denne SKA med

Either put in example, or set in reference to software solution that does not use DMA, when it is added to the report.

4.1.3 Wishbone Arbiter

With the presence of a Wishbone Master in the DMA Module, the tile has now two masters that will demand access to the shared Wishbone bus network. Arbritation is therefore required.

For this project, the ARB0001a: 4 Level, Round-robin Arbiter from WISHBONE Public Domain Library for VHDL has been taken in use. Figure [**?**]

shows how the arbitration works, with Round-robin. The arbiter will in turn check each input master for bus requests, and grants access thereby. If a master has no request, the arbiter will continue to the next. For this project, only two masters are present. Arbitration normally take a full clock cycle. See [4] for detailed implementation.

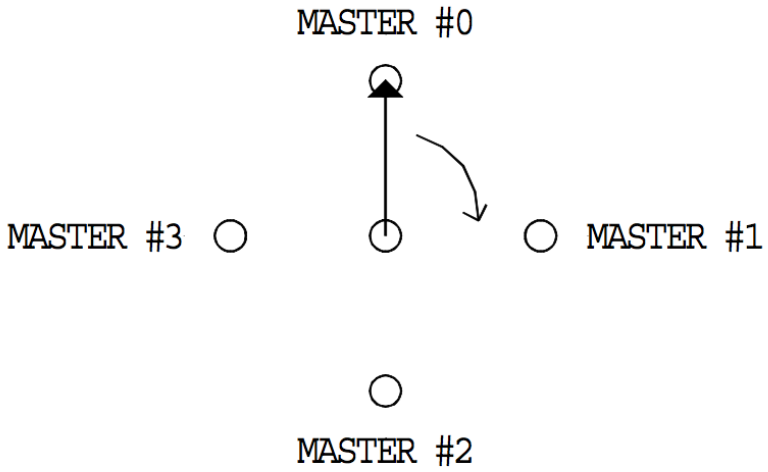

Figure 4.3: Wishbone Round-robin Arbiter, as seen in [4].

Round-robin arbiters work well in data acquisition systems where data is collected and placed into shared memory, since peripherals must often off-load data to memory on an equal basis. The choice of this arbiter is due to using an already established Wishbone arbiter cuts away time from designing a new one, which may even end up less efficient in worse case. An alternative to round-robin arbiters is priority arbiters. They are usually disadvantagous in said systems, but does not have the arbitration overhead [4].

In the case of arbitration between a CPU and a DMA Module, a priority arbiter may be used to achieve either burst mode, where DMA gets the full right on the bus until the transfer is finished, or transparent mode, where the CPU always has first right. The latter mode gives the slowest DMA transfer, but has shown to give best overall system performance. [1].

Discussion of alternatives. The following may be moved to future work.

Hmm.... Sounds uncalled for

Find those book sources again, if time. Current source does not validate claim of overall performance

Chapter 5

Measurements and Evaluation

Chapter 6

Conclusion

Bibliography

[1] Dr. Feza Buzluca. Direct memory access - Ninova. http://ninova.itu.edu.tr/tr/dersler/elektrik-elektronik-fakultesi/22/blg-322/ekkaynaklar?g358570, 2013.

[2] EECS. Single-isa heterogeneous many-core computer (shmac) project plan, March 2014.

[3] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, October 2008. Accessed December 7th, 2014.

[4] Wade D. Peterson. Technical manual: Wishbone public domain library for vhdl, October 2001.

[5] Dean M. Tullsen Rakesh Kumar, Norman P. Jouppi. Core architecture optimization for heterogenous chip multiprocessors, September 2006.

[6] Norman P. Jouppi Parthasarathy Ranganathan Dean M. Tullsen Rakesh Kumar, Keith I. Farkas. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction, December 2003.

[7] Norman P. Jouppi Parthasarathy Ranganathan Dean M. Tullsen Rakesh Kumar, Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance, 2004.

[8] Michael B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse, June 2012.

[9] Michael B. Taylor. A landscape of the new dark silicon design regime. *IEEE MICRO*, September 2013.

[10] David Koufaty Dheeraj Reddy Scott Hahn Karsten Schqan Ganapati Srinivasa Vishal Gupta, Paul Brett. The forgotten 'uncore': On the energy-efficiency of heterogenous cores.

Appendix A

Detailed DMA Description

This appendix contains additional details about the DMA implementation, and serves to give a deeper understanding of how each component in the DMA works in details. Section 1 describes every component in detail, while chapter 2 describes tests used to verify the DMA module.

A.1 DMA Architecture

This section aims to give the reader an insight into every detail of the DMA Module. Every component is detailed with all inputs, outputs and their functionality, as well as how they interact with each other.

A.1.1 Overview

Figure A.1 shows the entire DMA module, its components, and its inputs and outputs.

Components

The DMA module consists of the following components:

DMA Controller - The controlling unit responsible for handling transfer requests, monitor channel activity and sending out interrupts.

Channels - A channel is capable of performing a requested transfer, by issuing loads and stores. The system currently implements two channels. Each channel is split into two sub-modules, called the "load channel" and "store channel".

Arbiter - Used to select the next output: an interrupt, a store or a load command.

FIFO Buffers - Two FIFO buffers are used to store transfer requests and data.

Comparator - Used to identify which channel has requested the next data.

Adapter - Used between the request FIFO buffer and the DMA controller for proper synchronization between their signals.

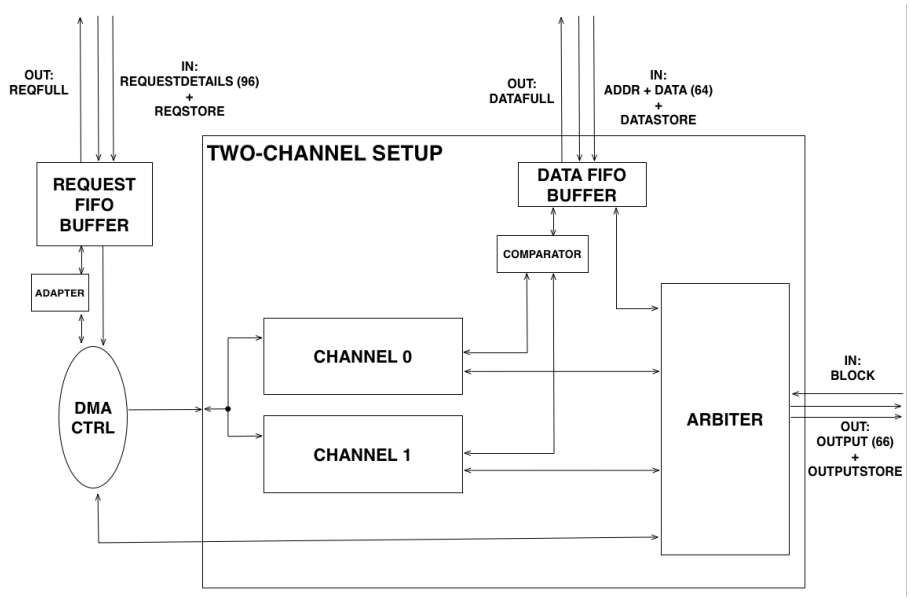

Figure A.1: Overview of the DMA module, with its inputs and outputs

Inputs and outputs

Inputs and outputs for the DMA toplevel are mainly composed of the the input and outputs of the FIFO buffers and the arbiter.

Request FIFO buffer - The inputs are control signals for storing data in the FIFO buffer, and details on 96 bits: 32 for the source address, 32 for the destination address, and 32 for transfer details, including transfer ID from the requesting peripheral and the amount of data to transfer. The output signal is the FIFO full signal, signaling that the buffer is full.

Data FIFO buffer - The inputs are a control signal for storing, and 64 bits of details about the data: 32 bits for source address, and 32 bits for the data itself. The output signal is a FIFO full signal, signaling that the buffer is full.

Arbiter - The outputs that passes through the arbiter are a control signal to be used externally, and details on 66 bits: the 2 first bits are used to identify the type of command, the rest are the details (interrupt ID, or load/store address and data). The input block signal may be used by the external system to stop the DMA module from sending out any more data if the external system is saturated and must process the current output before receiving any more. The impact of this depends on the external system.

A.1.2 DMA Controller State Machine

The DMA controller is implemented as a state machine. It is responsible for handling requests to do data transfers, monitoring active channels, and sending

out an interrupt signal when a channel has completed its data transfer. The state machine can be seen in figure A.2.

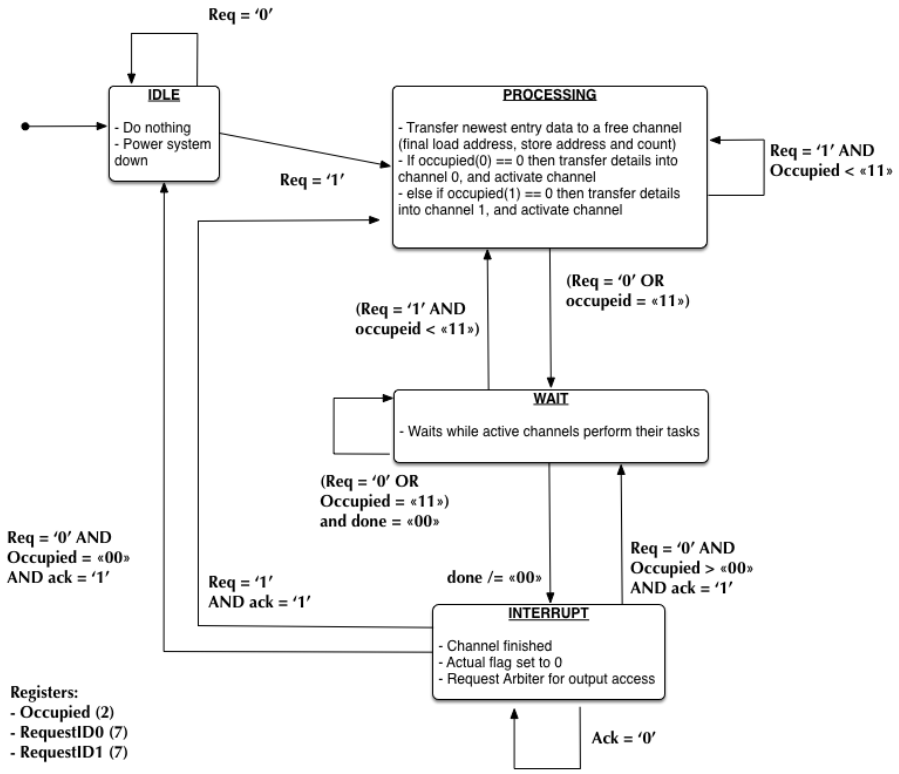

Figure A.2: Detailed DMA Controller State Machine

States

The states used in the state machine are IDLE, PROCESSING, WAIT AND INTERRUPT.

IDLE - Entered when there are no requests for data transfer. The DMA controller waits until a request signal is received.

PROCESSING - Entered when the controller has received a request and there is a free channel to execute the request. Outputs from the DMA controller to a free channel is transferred, activating the channel.

WAIT - Entered when both channels are active and there are no more room for new requests to be processed or when there are no more requests to process while there are still channels active.

INTERRUPT - Entered when a channel is finished with execution and no longer active; causes the DMA controller to request the arbiter to let it send out an interrupt signal. Until the arbiter acknowledges the request, the controller will remain in this state, and no new transfer requests will

be processed until done. When acknowledged, the controller will send out an interrupt signal with necessary details, including the interrupt ID.

Inputs

The DMA controller have the following inputs:

clk - Input clock signal.

reset - Reset signal that reverts state machine back to idle.

req - Input request signal for transferring data.

loadDetails - The initial 32-bit source address

storeDetails - The initial 32-bit destination address

reqDetails - 32-bit details relevant for the transfer. Contains the count (bits 32-21), which is used for both offset and counting down the execution to finish, and ID of the requestor (bits 20-13), which is used in the interrupt signal. Bits 12–0 are reserved for future use.

activeCh0 - Monitoring signal from channel 0, signaling that it is currently active.

activeCh1 - Same as activeCh0, but for channel 1.

interruptAck - Signal from arbiter that the DMA controller has access to send out interrupt signal from the system.

Outputs

These are the following output signals from the DMA controller:

reqUpdate - Signal to acknowledge the received data transfer request (req) and its details, so that next request may be sent. Used in facilitating pop signals at the request FIFO buffer.

FLAOut - 32-bit FLA (Final Load Address) output sent to channels. Calculated by adding loadDetails with count.

FSAOut - 32-bit FSA (Final Store Address) output sent to channels. Calculated by adding storeDetails with count.

counterOut - 32-bit counter output sent to channels.

set0 - Signal used to activate channel 0 and store FLA, FSA and counter data to it.

set1 - Same as set0, but for channel 1.

interruptReq - Request signal to the arbiter for sending out interrupt

interruptDetails - 34-bit interrupt details, sent out of the DMA top module when the controller requests the arbiter for access, and is acknowledged. Contains 2 bits to recognize interrupt with value “11”, and contains ID (7 bits) of the transfer requestor. Bits 24–0 are currently not in use.

Internal registers and signals

The DMA controller also has some internal registers and signals:

occupied - A 2-bit register, with each bit representing a channel. When the DMA controller activates a channel, it also sets the corresponding bit in the occupied-register to 1. The occupied register is compared with the incoming active signals from the channels. Whenever a channel finishes and deactivates, there is a difference between the active signals and the occupied-register, and the controller compares the values and the active signals to determine which channel has finished.

requestID0 - A 7-bit register, containing the ID of the requestor of the transfer currently being executed in channel 0. The ID is included in the interrupt signal the DMA controller sends out when the transfer is done executing in channel 0.

requestID1 - Same as RequestID0, but for channel 1.

totalActive - A 2-bit signal, which is the concatenation of both the active signals from the channels. Tp function properly, the DMA controller needs the active signals immediately when set signals are sent out, yet there is a cycle delay between sending the setX-signal until the channel is active. Therefore the setX-signals are also checked when totalActive is set. This is done by using setX OR activeX.

workDone - This is the signal used to determine whether a channel is done executing or not. workDone is calculated by subtracting the values of the occupied-register with the values of the totalActive signal. If the value is not "00", then a channel has finished, and the controller enters an interrupt state. The occupied-register is updated as well.

A.1.3 DMA Request FIFO adapter

The DMA controller is designed to process input data that arrives together with the request signal, but the FIFO design that is used to queue requests is not fully compatible with this setting. The suitable signal from the FIFO buffer to the DMA controller is the empty-signal, as NOT-empty can be used as req-signal. The reqUpdate-signal from the DMA controller can be used to pop new data. However, the data must be popped before it can be read, causing a one cycle delay between the received req-signal and the associated data. In addition, the first arrived data has to be popped without relying on the reqUpdate-signal, since the first NOT-empty must arrive in the controller together with the data. And finally, when the last data is read and the buffer is empty, the controller must not pop any more content, or it will cause problems in the FIFO. The reqUpdate signal must be ignored somehow.

The challenge is to make sure that req-signals to the DMA controller arrive at the same time with the corresponding data, and to make sure that incorrect pops to not occur. This problem is solved by adding an adapter between the request FIFO buffer and the DMA controller. It delays the request signals from the FIFO buffer by one clock cycle, and sends out pop-signals at correct time.

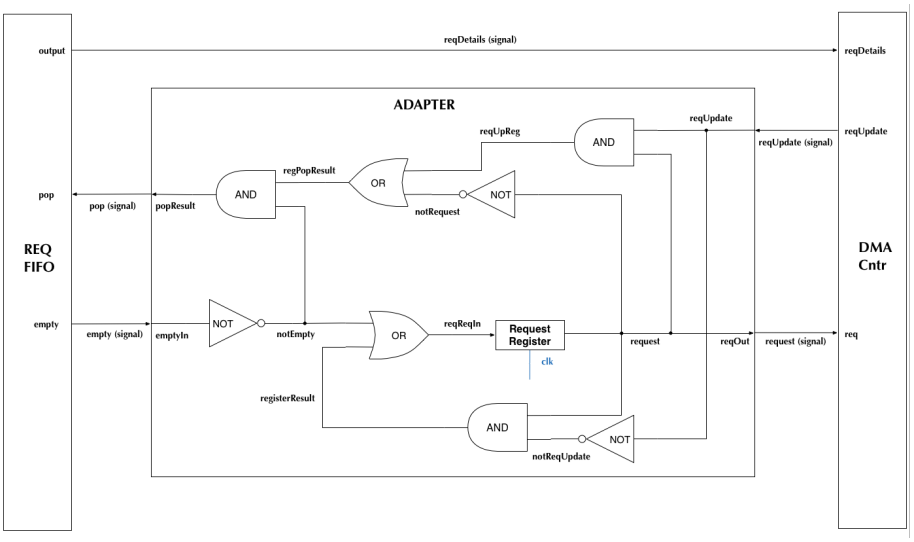

Figure A.3: Adapter between the FIFO buffer and the DMA Controller

Figure A.3 shows how the adapter is built, and also how the adapter, the FIFO buffer and the DMA controller are all connected to each other.

The adapter has one register for storing request signals, but consists otherwise of combinatorics. The NOT-empty signals are always delayed by a cycle, making sure they arrive together with the next corresponding data to the controller. If the register's value is 0, the first NOT empty-signal will cause the adapter itself to pop the buffer, and register changes value to 1. As long as NOT-empty is active, the adapter pops whenever reqUpdate is active. When the buffer finally is empty, the next reqUpdate signal will simply set the register's value to 0.

A.1.4 Manual subtractor

One of the main components used inside both load channels and store channel is the manual subtractor. It works by taking three inputs, two for subtraction and one as control signal. If the control signal is active, the first input will have its value subtracted by the second input, and the output is the result. If the control signal is not active, there will be no subtraction, and the first input value is the output value.

A.1.5 Load channel

DMA channels are split into two parts, a part responsible for issuing load commands, and a part responsible for issuing store commands with appropriate data. These are called the "load channel" and "store channel", respectively, and are not to be confused with channels 0 and 1 in figure A.1, as they both contain these two sub-modules.

The load channel is responsible for generating load requests that are sent out from the DMA. It can be seen in figure A.4.

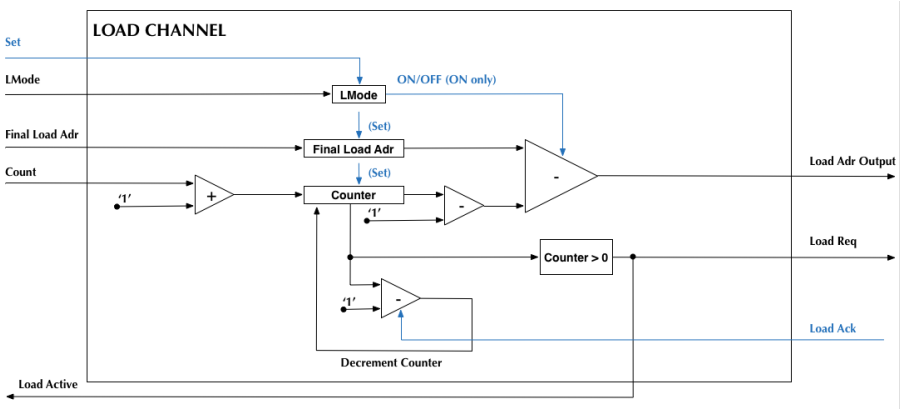

Figure A.4: Load channel

Inputs

The load channel have the following inputs:

clk - Input clock signal

reset - Reset signal

set - Signal from DMA controller used to set all registers to the input values

LMode - Value for LMode register

FLA - 32-bit value for the FLA register

count - 32-bit value for the counter register

loadAck - Signal received from the arbiter, specifying that the current load request is acknowledged.

Registers

The load channel have the following registers:

LMode - Register for loading mode, connected to the subtractor.

FLA - Contains Final Load address

counter - Used to count down the number of load requests to be generated. Also used to calculate offset from FLA, if LMode is 1.

Outputs

The load channel have the following outputs:

loadAdrOutput - 34 bit signal with the address for the current load command. Is concatenated with two bits with value “00”, used as MSB.

loadReq - Request signal from the load channel to the arbiter, for requesting output access to issue load command to the system.

loadActive - Signal sent out, used to indicate that the load channel is active.

With the exception of loadAck from the arbiter, all input signals are received from the DMA controller. When set is active, all registers are overwritten by the matching input signals. LMode determines if the load requests are for a fixed address, or a block of addresses, by activating or deactivating a manual subtractor that the FLA and the counter are connected to. If LMode is active, the current load address is calculated by subtracting the FLA with the current counter value (minus 1). The counter is used for both offset calculation, and counting down the number of remaining requests. A unique number is needed to identify whether the current job is done or not, and the value 0 has been chosen for that. As long as the counter has a value above 0, the job is not done, and both the load request and the load active signals are active. The load channel does not need any additional data or input signals besides the registers, therefore the channel is designed to issue loads as long as the counter is not 0. At the final count, the final load address should be issued, but if the counter value is 0, there will be no requests. This is avoided by incrementing the Count input value by one when issuing a new job (set = 1), and decrementing the counter's current value by one before subtracting with FLA. Thus, when issuing the final load, the counter's value is "...001" while the counter's output value to the manual subtractor is 0. The load channel can issue the final load, and then the counter reaches 0 when done. The counter is also connected to another manual subtractor, which decrements the value with 1 if activated. Except from when the set-signal is active, the counter is always overwritten by the result from this subtractor. This manual subtractor is activated by the loadAck signal from the arbiter. Whenever the loadAck signal is active, the current load address output passes through the arbiter, and the counter is decremented by one.

Why final loading address, and counting down the offset?

When designing the load channel, it was needed to have at least a static address for loading, an offset from that address, and a value for counting down the remaining loads to be issued. The needs are the same for the store channel as well. The current design incorporates both offset and counting into one single register. The current address can be calculated by having a starting address that is added together with the current offset. Alternatively, one may use a final address and subtract it with the offset. If counting down, the zero-value is distinguishable from non-zero values, and can be used to know whether the countdown is finished or not. If a counter has an incrementing value, there must be an extra register that contains the final count value so that the load channel can know when the job is finished. In order to utilize as few registers as possible for the task, the counter and offset value has been joined into one register, and the system counts down towards zero. This eliminates the need for any extra final count registers, since the counter can be checked for zero or non-zero values. This means also using the final load address and subtracting it with the offset from the counter, when calculating current address.

A.1.6 Store channel

The store channel is responsible for generating store requests, together with the correct data that has been loaded from a distinct memory address into the DMA

module. It can be seen in figure A.5.

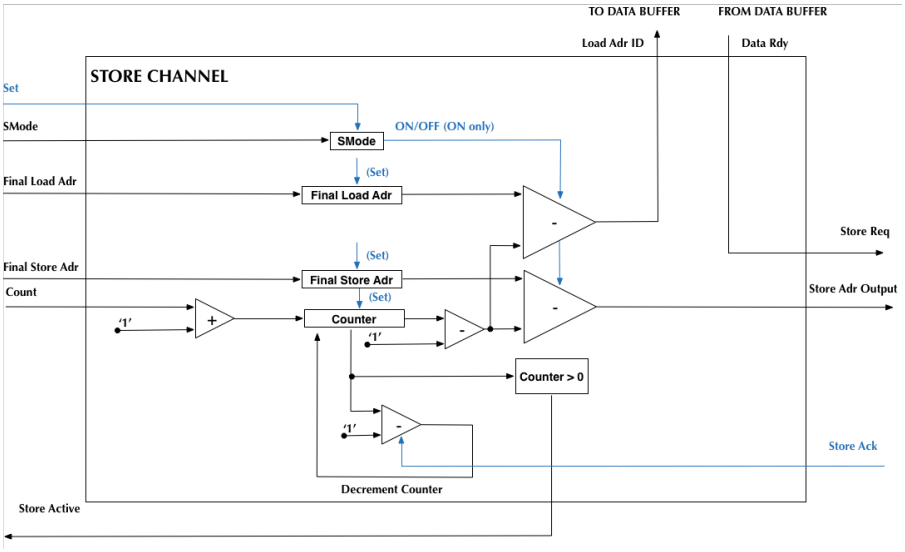

Figure A.5: Store Channel

The design of the store channel have similar inputs, registers and outputs to the store channel, which can be seen in figure A.5. There are some differences of note:

FSA and FLA - Final store address is unique to the store channel. The store channel still consists of the FLA, needed to compare loaded data with the origin address.

Unique inputs and outputs - Not found in the load channel. These are the input dataRdy, and the output loadAdrID.

storeReq and dataRdy - Request signal storeReq from the store channel to the arbiter is not based on the counter, but on the input signal dataRdy.

Any loads issued by the associated load channel is expected to reach a data buffer shared by all channels. All received data has an ID, informing which address in the memory the data was loaded from. Load address ID, based on the FLA decremented with the counter, is compared to the ID of the next data in the buffer. If there is a match, the next data belongs to this channel, and a dataRdy-signal is sent in. In this design of the store channel, the dataRdy is used as a request signal. The reason behind this design is to make it possible to treat the store channel as a black box from outside, so that the if the store channel is replaced or changed in the future, the outside world does not need to know anything of it. Since there is a corresponding store address for every load address, the counter is used to calculate offsets from both the FLA and the FSA. Notice that there is no data sent into the store channel. The data is provided by a shared data buffer, and sent straight through to the arbiter. The store channel sends out the store address, together with two bits for store commands. These two bits are concatenated with the address, and are the two MSBs, with value “01”.

A.1.7 Arbiter

The arbiter is used to arbitrate between all channels and the DMA controller, when any of them is ready to send out their output to the system. For each clock cycle, it selects between a number of requests, and allows selected command and assosiated data to pass through. It can be seen in figure A.6

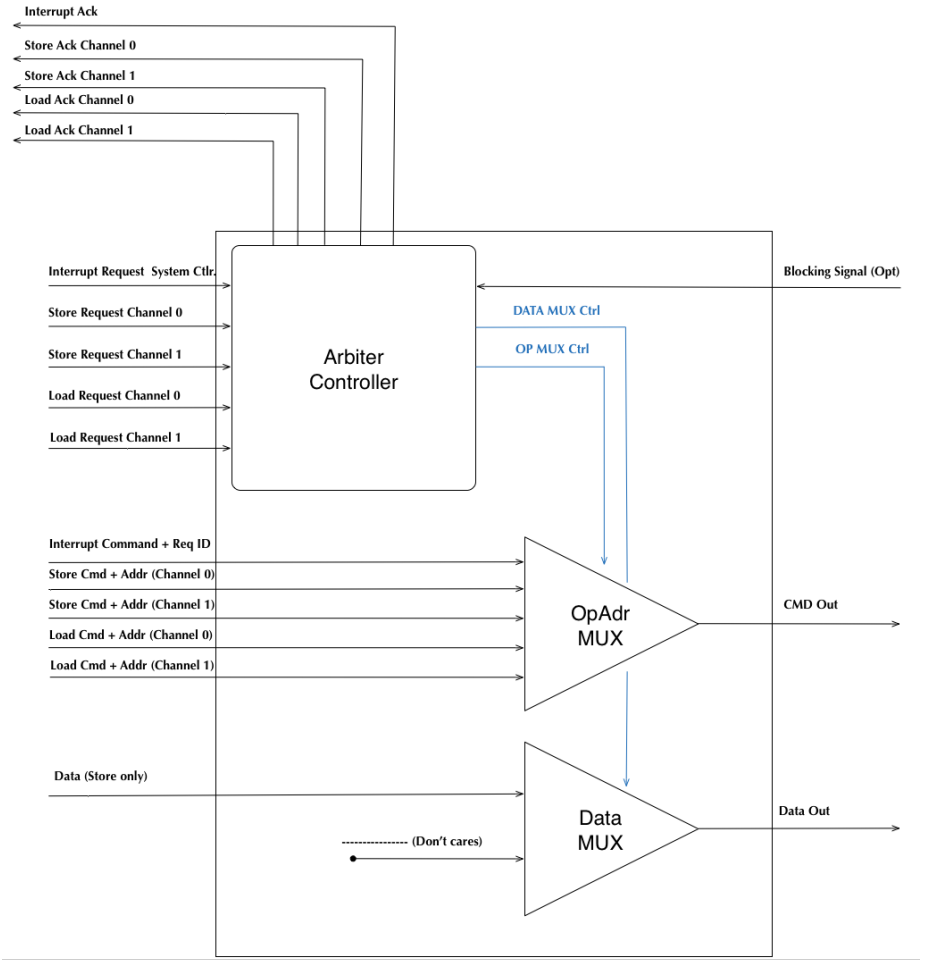

Figure A.6: Store channel

The arbiter receives store requests and load requests from each channel, as well as interrupt requests from the DMA controller. Each channel receives one store acknowledge signal and one load acknowledge signal from the arbiter. The DMA controller also receives interrupt acknowledge signals from the arbiter. For each channel, there are two 34-bit command inputs, one for storing requests and one for loading requests. There is also a 34-bit interrupt input from the DMA controller. All these command requests go through a 34-bit multiplexer, set by the arbiter controller, based on which request is acknowledged. For instance, if a store request from channel 1 is selected, the store details from channel 1 is the one that passes through the multiplexer and reaches CMD out. The data input

comes from the shared data buffer, and is sent through the data multiplexer whenever there is a store request is acknowledged. There is also a blocking signal connected to the arbiter. If the blocking signal is active, then no requests will be acknowledged, and any work inside the DMA module, whether it is in a channel or in the DMA controller, will be halted. The system bus is expected to be the bottleneck in this design, and any outputs from the DMA module must not be sent out faster than the system can handle. Should there be any need, the system needs to be able to halt the DMA module from issuing any more commands until the previous commands have been handled.

The priorities for the arbiter controller are:

- Block, then interrupt, then store, then load
- If there are two stores or two loads competing, then the least recently used channel gets priority. This way the commands sent out from the DMA module will alternate between the two active channels. This is separate for load requests and store requests.

A.1.8 Comparator

The comparator is used to compare the current output data's loading ID with the loading ID the requesting store channels are waiting for. The channel that has a match receieves a dataRdy-signal from the comparator.

A.1.9 FIFO Data buffer

The implemented data buffer is a FIFO buffer. It is 64 bits wide, 32 bits containing the address from where the data is loaded, known as loadID, and 32 bits containing the data itself.

A.1.10 Interaction in the two-channel setup

The channels are set up in what is called the TwoChannelSetUpBuffered module. This setup can be seen in figure A.7. The figure shows how two channels 0 and 1 are combined in this submodule together with the FIFO data buffer, a comparator, and an arbiter.

The reason behind this choice was to have a working environment where testing the interaction between the channels, arbiter and input buffer, were possible before having finished the DMA controller.

All inputs to the channels from the DMA controller go through synchronous buffers. Originally, the setup was designed without buffers, but behavioural simulation showed that whenever a set-signal was received exactly at a clock edge together with altered input values, the previous counter value would be written into the counter registers in the channels. The rest of the inputs to the channels were correctly written. The inputs for the counters passes through some additional combinatorics before reaching the registers, so apparently these values did not reach the registers in time with the set-signal. Adding buffers for all input signals solved this problem, by giving all inputs to the channels an entire cycle to synchronize. The cost is an additional cycle when activating the channels.

The DMA controller, which sends in these inputs, is also connected directly to the arbiter. Whenever the DMA controller needs to send interrupt signals, this goes through the arbiter.

Notice that each setX_buf registers are also connected to the active-signal of their corresponding channel that are sent back to the DMA controller. Even if buffering the inputs delays the channel activation by one cycle, the DMA controller should still receive the active-signal as early as possible.

Popping data from the data buffer at the right time is also a challenge. Whenever a channel stores the latest data from the FIFO buffer, new data should be popped, but only if there is more. If the buffer is empty, no more pops should be issued. Whenever new data arrives to an empty buffer, it must be popped first before any of the channels can read the ID. This is solved by implementing some combinatorics, and a register called popFirst. When new data arrives, there must be a cycle between the push and the pop. popFirst is set by combining the empty-signal with the push-signal, which are both active when the first data arrives, and popFirst will be used in popping the first data during the next cycle. Notice that either a store-ack signal must be active, or that there must be no data-rdy signal active. This is to avoid a potential deadlock due to data being overwritten too early. Simulation showed that if the last data was popped, but not sent due to block or interrupt, it would be overwritten if new data arrived in the FIFO buffer before the interrupt or block was over. Otherwise, if there is content in the buffer that is to be popped, it will be done so as long as the empty signal is low, and as long as any of the store ack-signals from the arbiter is active (meaning that data with store address is sent through and out of the DMA module).

Whenever something is sent through the system, either a store request, load request, or interrupt, an additional signal, storeOutput, is set. This is done by combining all ack-signals with an OR gate into storeOutput. This is to be used as a separate signal to the system that handles the output from the DMA module, to signal that a new command is available.

A.1.11 Tweaks in the system

The DMA module was designed for word addressing, while the system used to test the hashing module and the DMA module in this project uses byte-addressing. The current design does not support both, and in order to both save time and to adapt the DMA module to the test system, the following changes were made:

- The DMA controller multiplies the counter value with 4 before calculating FLA and FSA
- The multiplied countervalue is stored in the channels
- The channels decrement with 4 instead of 1

A.2 Verification and testing of DMA Module

The DMA module is tested by first testing the individual components, then their interaction with each other.

In the individual testing, the component receives different combinations of its inputs over time. The test shows whether the component behaves as expected, based on the input. This is done by reading the outputs based on inputs and/or values of internal registers. Components that are tested individually are:

- **Adapter**
- **Arbiter Controller**
- **DMA Controller**
- **Load Channel**
- **Store Channel**

In the interaction testing the components are tested for their interaction with each other. Since the components have been tested individually, they are assumed at this point to be functional. Internal values are thus ignored, also since the focus of the tests are to ensure that the interaction between the components are working as expected. The interaction tests are:

Arbiter - Tests that internal multiplexers are set correctly by the controller, based on the selected requests.

Two Channel setup, single request - First of two tests that checks if the two-channel setup works as expected, by activating a single channel. Data with ID is fed manually to the data buffer. The tests ensures that a load channel sends out number of requests according to count value, and that whenever data arrives, stores are issued and get priority above loads. Also tests if interrupts and blocks get highest priority, and that active signals are activated and deactivated correctly. Correctness is ensured by reading the output values (mainly detailsOutput and dataOutput) from the two-channel setup.

Two Channel setup, two requests - Second of two tests, this one activates both channels. In addition to the first test, this one also ensures that loads from both channels alternates when both are active, and that correct channel identifies and stores the correct data that has been loaded. When one channel is done, the other channel will have full access to the arbiter, and loads will no longer alternate. Correctness is ensured by reading the output values.

Toplevel - This is where the entire DMA module is tested, as requests are sent in. Each cycle with load, store, interrupt and blocking are counted (either identified by the command value, or by the blocking signal when it is high, in the test). For each load issued, a register receives the load address, and sends it back to the DMA module with a push signal and data (since the data value is not important for this test, each value is incremented by 8). Thus there is a cycle delay from a load is issued to the data arrives to the DMA module. In the real world, there will be far more cycles from a load is issued, until its data arrives in the DMA module. Whether issuing multiple loads or not is possible also depends on the system it is integrated with. There will be many differences between this project, where the DMA

module is connected to a single shared bus network, and when the module is integrated into SHMAC, where the main network is a packet-switched network. In this test, however, all data arrives quickly, as if using a bus and memory that is extremely fast. One reason is to simplify the test by reducing the necessary waiting time in the test itself, and another is that the module is tested for near-maximum capacity. There are three tests in total: One with single request (only one channel active), one with two requests (both channels active) and one with four requests (test for waiting requests). Correctness is ensured by comparing the test counter values with expected number of loads, stores, interrupts issued, and number of blocks.

It turns out that in the tests for each round of blocks, the block counter counts one block less than the number of cycles a block is active. It also turns out, when analysing the signals, that the module is inactive in the correct number of cycles as long as the block signal is active; the block counter merely does not count during the first cycle during a block. Since this is considered a flaw with the test itself and not with the DMA module, the error is considered negligable.

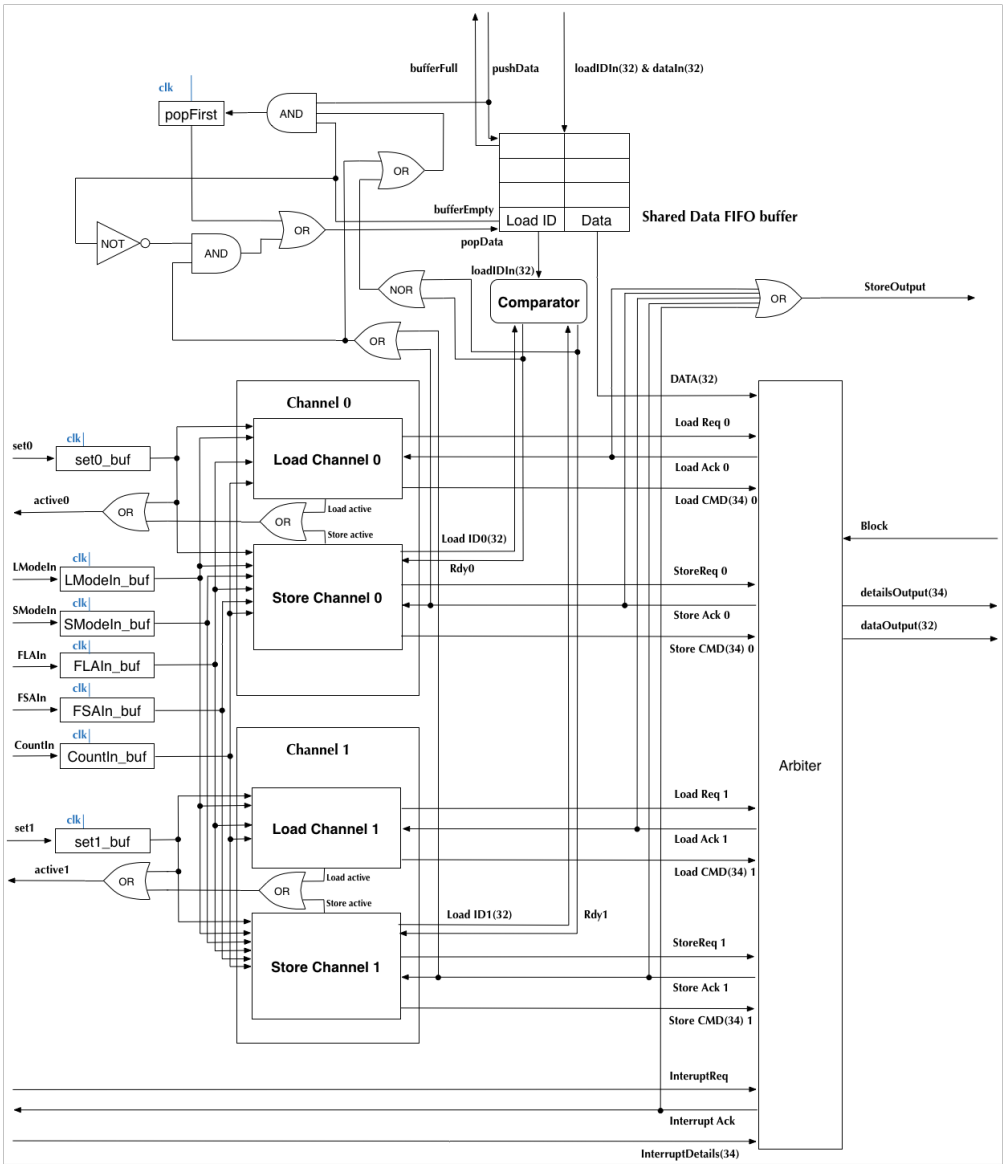

Figure A.7: The two-channel setup, buffered version