Master of Science Thesis
The Need For Speed

Magnus Andersson
Christian Melki

Department of Information Technology
Uppsala University
Axis Communications AB

July 31, 2003

Abstract

In future Axis products the need for embedded raw CPU performance will increase due to a combination of increased I/O bandwidth and new kinds of demanding applications. Future System-on-Chip products might serve several GBit channels together with advanced cryptography and multimedia applications, e.g. IPSec, sound and image analysis/processing.
The goal of this thesis has been to give a good overview of current possibilities in the embedded CPU market and to identify good candidates for replacing Axis Communications CRIS architecture. The CRIS is an old, not particularly scalable, architecture.
The present market features a lot of embedded CPUs with different capabilities and prices. Axis Communications main interest has been royalty free cores, that are beginning to appear on the market.
We have looked closer at the SPARC Leon2 and OpenCores OR1200 and found that they are both very viable alternatives to many commercial processors for the embedded market. The Leon2 boasts classical SPARC architecture with industrial quality implementation. OR1200 is backed by a promising community with an interesting future. Both cores have have been used in commercial applications. The Leon2 is used by ESA (European Space Agency) in several applications and the OR1200 has successfully been implemented in FPGA embedded solutions by a few companies.
We have found both cores to be very suitable for embedded solutions and should be taken into consideration.

Preface

This master thesis project was done at Axis Communications AB in Lund, Sweden, for the Department of Information Technology at Uppsala University. The project was carried out over a period of 20 weeks which is equivalent to 20 Swedish university credits.

We would like to take the opportunity to thank the employees at the ASIC-department, especially our supervisor Stefan Sandström for his excellent support throughout the entire time we have spent here at Axis Communications AB. We would also like to thank our examiner at Uppsala University, Prof. Erik Hagersten.

Our best regards to Axis Communications as a whole for letting us spend some excellent months there both learning and working with our thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The purpose of this thesis is to investigate into future embedded CPUs for Axis Communications. Their current architecture, the CRIS[10], has several disadvantages. Scaling of the CRIS becomes very difficult because some of the architectural properties.
The effort of scaling becomes greater than rethinking the entire architecture and choosing something that is easier to work with. But this path is also non-trivial due to the many choices presented, which all need to be carefully evaluated before anyone can make a good decision about the path to take for the next generation.

The future brings many new bandwidth-demanding applications and protocols to the table that need consideration when designing and choosing architecture.
Of special interest are high-bandwidth streaming, file serving and other server like tasks. On the protocol side, names such as USB 2.0, Serial ATA, Firewire, SA-SCSI, Gigabit Ethernet present themselves as high-bandwidth, high-interrupt load protocols. This is when the design choice of caches and MMU's[1] becomes very critical for sensitive applications. Everybody is on the quest for *The Need For Speed* and the effort needed to stay competitive increases with the ever increasing complexity of our hardware systems.

1.2 Problem description

The first part of the thesis will investigate and suggest one of several possibilities to extend the CPU performance in the next generation of ETRAX chip to the range of 400+ MIPS. This shall be done using a freely available architecture.
Examples of this might be:

- Further increase the speed of the existing CRIS CPU including the cache/MMU subsystem. This might also include an investigation of the possibility to re-define the CRIS architecture to match the need for speed.
- Invent a new RISC architecture suitable for high speed embedded implementations.

[1]Memory Management Unit. Handles memory segmentation and memory protection.

- Use a freely available CPU architecture e.g. SPARC[16], DLX, MMIX[12].
- Use parts from a freely available CPU core e.g. LEON SPARC-V8[17] open-source core.

The second part of the thesis will suggest an implementation approach for the CPU, MMU and cache subsystem. Critical parts of the design should be implemented in Verilog[1][2][3] to verify the performance requirements. A detailed simulation model written in Verilog and/or C shall also be developed and used for performance comparisons to the existing CRIS CPU.

- The design shall be described at RTL[2] using Verilog.
- Fastest cache memory/MMU access time is 2 clock cycles @ 400 MHz. The memories available are generally not faster than that.
- Separate instruction and data caches.
- MMU's and protection mechanisms suitable for running Linux.
- GCC[3] support. Existing or "easily" derived from existing GCC port. I.e. the CPU may be a tweaked version of an existing architecture.
- 32 or 64 bit registers and data paths. 64 bits are desirable. Configurable between 32 and 64 is even better.
- 400 MHz, giving a bit more than 2ns per cycle after considering clock skew, clock jitter, FF[4] setup times and FF delays.
- CPI[5] close to one.
- Area less than 200k gates excluding caches and MMU's for 32 bit data paths. One gate is defined as a low fanout NAND gate.
- At least 6 times the performance of CRIS v10 (present in ETRAX100LX[6] running at 100 MHz. This may be measured using a set of benchmarks that is used when tuning GCC for CRIS.
- A clean, simple and regular architecture is desirable.

[2]Register Transfer Level
[3]GNU C Compiler
[4]Flip-Flop. Unit that toggles on clock input.
[5]Cycles Per Instruction.
[6]Axis integrated SoC chip

Chapter 2

Theory

2.1 Basic processor architecture

The definition of a *Central Processing Unit* (CPU) dates back to the *Von Neumann Machine* by John Von Neumann[1]. The Von Neumann machine has five units do do the computing workload. A CPU, a memory, a control unit, input/output units and a bus to tie them all together.
This definition is very general and adapts well to the definition of a modern machine although the definition was invented in 1944. A simple Arithmetic Logical Unit (ALU) fetches instruction in memory, executes them and stores the value back to the memory. This simple idea works well in theory but becomes impractical in reality because the time involved in doing everything in an entire cycle. If we should compare the theory to a car factory floor, then the number of units produced from the factory would be very low. In this imaginary factory there would only work one person. One person to fetch every part needed for one car, assemble the car and then output the result. No one would help this person with fetching the parts needed, assemble the car or output the result from the factory.

The need for optimisations become obvious as the need for more cars increases. Optimising the factory for quicker production is simple at the beginning. Let us start by regarding each step in the manufacturing as a separate part. We would then proceed by hiring one person to do each part. One for fetching the parts. One person for preparing the parts. Another person for assembling the car (albeit, in reality this would probably be the most complex task. We assume here, for the sake of simplicity, that each task is equally time consuming). And the last person to output the cars from the factory. This change will increase the output by quite a few percent. We have now a total of four persons working in our factory instead of one. Everyone has become specialised in their task and becoming really fast at doing it. This assumes that no one in the factory stalls in production because of some missing parts, broken machines or the boss coming down to yell at the poor employee. In the case of a interruption in one of the production units all other units before that one must wait before sending their produced part further into the chain. This is simply the case because no person in the factory can

[1]John Von Neumann (Neumann Janos) (December 28, 1903 - February 8, 1957) was a Hungarian-American mathematician who made important contributions in quantum physics, set theory, computer science, economics and virtually all mathematical fields.

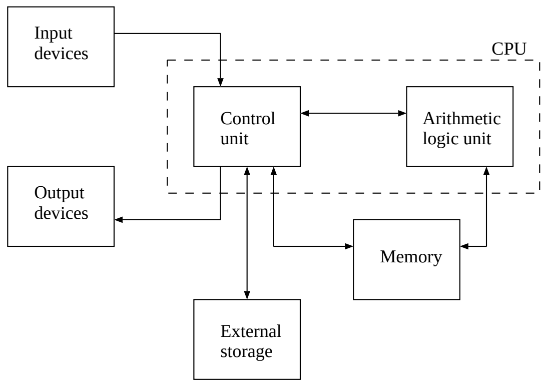

Figure 2.1: Von Neumann Machine

handle more than one production unit at a time.

This type of optimisation will not result in a speedup factor of four however. This is simply so because when the factory gets interrupted by some event it is stalled by up to four people doing nothing, besides twiddling their thumbs, waiting to continue with the car building process. The more people you have working in the factory the longer it takes to resume a correct course of operation after an interrupt. If you have a big factory or very complex production units then a stop in production is going to cost you a lot of wasted time before production can start again. So while building an optimised operation for fast production you become very sensitive to interruptions. This requires you to pay more and more attention to prohibiting unnecessary interruptions. Some are necessary for human beings however. We do not work forever. We need brakes, sleep, and an occasional chat with our colleagues.

This was the first part of optimisation. What's next? We could hire more people to do the same step and install another production line in our factory. That's one possible solution. We could also work on multiple production units in the same factory unit etc. The range of optimisations are wide and the concepts can easily be applicated from the real world to CPU design for our computer. The difference is not that far fetched and so far we have described the basic concepts of caching, pipelining, super-scalar designs, and vector/multiple instructions, multiple data type of machines. However, there are one observation and one law that come in very handy when designing CPUs. Something every designer should be aware of and never forget about. The first one is the observation that Moore[2] stated in 1965 that "*the number of transistors per square inch on an integrated circuit would double approximately every 18 months and that this trend would continue on a foreseeable future.*" The second one is more of a law than an observation. It's called *Amdahl's law of diminishing returns.* Amdahl[3] simply stated that "*the maximum speedup* S *gained from using more processors* P *tends to become* 1/P *as* P *tends to infinity*". Both the observations are from roughly the same time but only Moore's Law has gained public knowledge.

[2]Gordon E. Moore. Co-founder of Intel. (Jan 3, 1929).
[3]Gene M. Amdahl. Computer Scientist. (Nov 16, 1922).

2.1.1 Processor pipelining

The most basic form of processor optimisation is called pipelining. In analogy with the car manufacturing facility we segment the processors job of executing instructions in a similar fashion. The most natural segmentation for a processor has become a 5-stage segmentation, which is taught to the students at almost all undergraduate computer science courses at universities around the world. Many architectures used this segmentation in their infancy and some, less complex processors, still do. The segmentation is as follows.

1. IF - Instruction Fetch. 1:st stage
2. ID - Instruction Decode. 2:nd stage
3. EX - Instruction Execute. 3:rd stage
4. MEM - Memory Access. 4:th stage
5. WB - Write back. 5:th stage

The segmentation is quite natural and does not require much explanation. The first stage, Instruction Fetch (IF) takes care of getting new instructions from the cache to the processor. The IF unit must also keep track of where the processor's current execution in memory is. This is referred as the Program Counter (PC). The PC is normally incremented to the next instruction in memory. But if a jump to a different place in memory should take place, then the PC must be adjusted accordingly. The same applies to the analogy when the factory production gets interrupted. When going back to work, the workers must recall where they where in the current context of things. So when a processor gets interrupted with one thing, before starting another task, actions must be taken to ensure that the current state of context can be recalled when returning to the work set previously left. This is often referred to as context switching (CS).

ID or Instruction decode is then handed the newly fetched instruction and takes care of sorting out just exactly what was in that instruction, what goes where and to what resource it goes. The stage complexity heavily depends on what Instruction Set Architecture (ISA) is used. Many older processors used an ISA which had varying instruction length, many addressing modes and just a lot of complexity. At that time, code size was important and architectures were designed with code size in mind. Many processors had a lot of features and special instructions to do different things, all of which executed in different time. This type of design was natural when processors really did one thing at a time. During those days clock speed was not pushed as hard as it is today. The processors did a few million clock cycles per second and did not have the problem with very slim marginals in time like modern fast paced processors do.

In the past, doing much work per cycle was believed to be the solution to the quest for speed. This proved to be very wrong. Simply put, because most executed instructions proved to be utterly simple. Adding a number to another, subtracting, jumping here, moving this, there. These are very simple instructions compared to multiplications, divisions and square root like functions of arithmetic operations. So the processors became good at defining small code for an infrequent amount of complex instructions as opposed to executing the simple and frequent instructions fast. The problem that occurred now was that the CPUs became so complex that turning up the frequency or doing optimisations for speed was not that simple anymore. So computer scientist around the world figured out a new way to deal with complexity. They completely abandoned the old era of what became known as Complex Instruction Set Computers (CISC) and proclaimed the birth of Reduced Instruction Set Computer (RISC).

RISC was to make all the complexity clutter go away. All Instructions should have equal length, minimise all addressing modes inside the CPU and try to keep execution times as even over the instructions as possible. All the CISC's were supposed to go away with the coming of this revolutionary thinking. Intel x86, Motorola m68k was predominant in the CISC market at the time and they had gathered such a large user base and code base that moving all code from one architecture to a new one was something the customer did not want to do. At the time there where no programs for the RISC machines so CISC continued to live and are still used today. The most notable example is the Intel x86 which almost all personal computers use today. It was simply cheaper to put more brain power behind the x86 to speed it up than trying to force users to rewrite their programs for the new architecture. Most modern architectures today belong to something called the Post-RISC era. Processors today are not CISC nor RISC. They may have an external shell of either definition but on the inside, the CPU is a completely different beast. The ID-stage was greatly simplified by the RISC revolution and became manageable once again in the quest for speed.

	Clock number								
Instruction	1	2	3	4	5	6	7	8	9
Instruction *i*	**IF**	**ID**	**EX**	**MEM**	**WB**				
Instruction *i* + 1		**IF**	**ID**	**EX**	**MEM**	**WB**			
Instruction *i* + 2			**IF**	**ID**	**EX**	**MEM**	**WB**		
Instruction *i* + 3				**IF**	**ID**	**EX**	**MEM**	**WB**	
Instruction *i* + 4					**IF**	**ID**	**EX**	**MEM**	**WB**

Figure 2.2: 5-stage pipeline example[4]

The EX stage does exactly what it states. It executes the instructions and gives a result to the next stage. An interesting note is that the EX stage is usually a very small pipeline part of a modern processor. For example, out of the Intel Pentium 4's 20 pipeline stages, only one is the EX stage. The rest of the stages deal with the complexity of such a long pipeline and all the logistics involved around it.

The MEM stage deals with memory operations. If the instruction was a load then this stage loads the address computed in the previous stage. If the instruction was a store then the MEM stage stores the value from a register to the address computed in the previous stage.

The WB stage deals with a load instruction and the result of an ALU operation. It stores the value from either an ALU operation or a load instruction back to the register file.

2.1.2 New architectural solutions

After the RISC era there the world has seen a few more impacts on architectural design of processors. The two most prominent technologies are SIMD and VLIW (EPIC[4]). Both were invented in the quest for exploiting potential parallelism in code and data.
SIMD was introduced and categorised by Flynn[5] in *Flynn's Taxonomy* where he segmented the fundamentals of parallel processing into 4 sub categories.

- SISD - Single Instruction, Single Data
- SIMD - Single Instruction, Multiple Data
- MISD - Multiple Instruction, Single Data
- MIMD - Multiple Instruction, Multiple Data

SIMD is a type of vectorised processing used by many CPU's today. The most notable examples are the integer and floating point vector units provided within these machines. Familiar names like SSE, SSE2, MMX, MMX2, 3dNow!, Altivec, VMX etc might ring a bell. These instructions normally segment a processors data work units into smaller subsets for operation by a single instruction.
SIMD still work within the complexities of dynamic scheduling per instruction issued to the processor. This means that the processor itself have to track completion and stalls on any given SIMD flow.

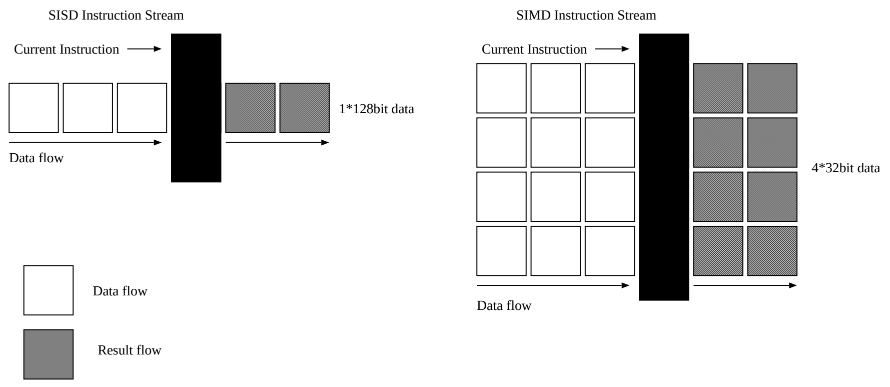

Figure 2.3: SIMD Instruction Flow

This is why VLIW was invented. VLIW is a type of MIMD machine. Within a Very Large Instruction Word you can fit multiple of smaller instructions with their accompanying data. This way you get to have static scheduling inside VLIW instructions and dynamically scheduled in between VLIW instructions. Of course you can take the VLIW concept all the way to make the machine entirely statically scheduled, thus relying on the compiled code to be very aware of the machine it's executing on and on the instruction flow of the code.

[4]Explicitly Parallel Instruction Computing
[5]Flynn, M.J invented the taxonomy in 1966.

The benefit of an entirely statically scheduled processor would be that much complexity is removed from hardware and resources can be freed to concentrate on executing instructions rather than handle the work of scheduling them. This would mean that correctly compiled code on a statically scheduled machine could potentially be very fast. Because the compiler has access to the entire source when generating code, it could create some great optimisations that a dynamically scheduled processor can't, simply because it doesn't know what to expect in the next flow of instructions.

The disadvantages of a statically scheduled processor are several. In fact, you can't just take compiled code for a statically scheduled machine and run it on the same architecture but with wider instruction path for example. All code must be recompiled to exactly match the processor. Dynamically scheduled processors are much easier to deal with in that respect. A modern P4 by Intel can still execute code from it's ancient sibling, the i386, and still do it very fast at native speed whereas a statically scheduled processor might be forced to emulate it's sibling for the code to work. Other disadvantages could be that if there is no parallelism to exploit in the code, you probably have a good deal of hardware just sitting idle and loosing out to a dynamically scheduled machine.

Both SIMD and VLIW have their applications, still more SIMD than VLIW as of today however. Most of the problems with VLIW arises from the software complexity in the form of a very advanced compiler that needs to schedule all instructions in advance to gain speed. The technology is extremely compiler dependent.

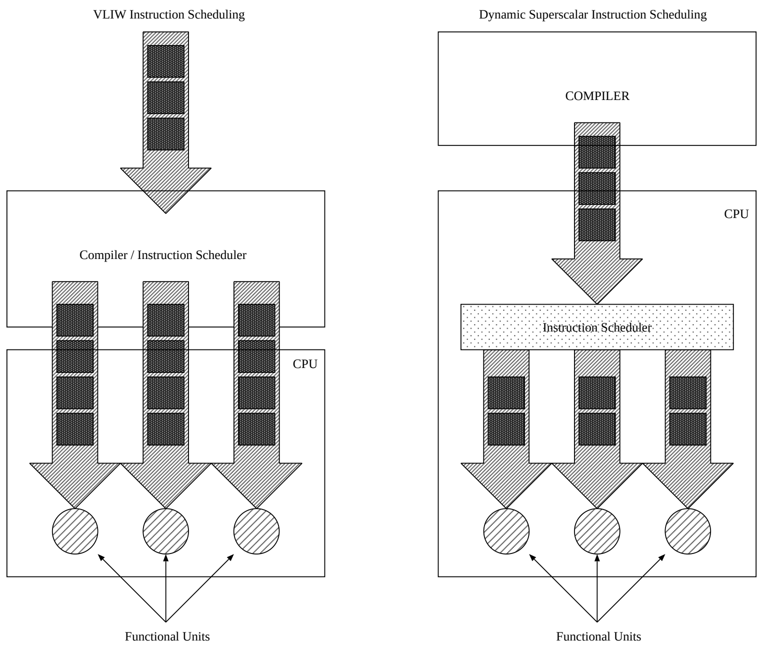
VLIW Instruction Scheduling
Compiler / Instruction Scheduler
CPU
Functional Units
Dynamic Superscalar Instruction Scheduling
COMPILER
CPU
Instruction Scheduler
Functional Units

Figure 2.4: VLIW Instruction Flow

In the context of embedded applications VLIW has not yet reached into the embedded processors market. SIMD instructions exist and should probably be used, depending on your application. But still, most of the embedded processors are either very old constructions or simple RISC processor of some kind.

2.1.3 Caches

Caches are probably the first thing that is needed to reach execution throughput. Without caches the processor would have to access a more distant cache (the main memory or perhaps an even slower media) to fetch data. The analogy is that for every part needed to build the car the poor employee would have to run to the warehouse in a different city each time he needed something instead of just getting it from the local warehouse just next to his workplace at the production line. Without fast data access we cannot feed our CPU with data. So we need something that can provide data in the same pace that the CPU wants it. But why don't we design all memory as fast as the processor? If it was possible to provide tremendous amount of data as fast as the CPU demands them, there would be no need for caches. The problem is that nobody has solved it yet. When terms like manufacturing costs, physical sizes and energy consumption comes in to consideration this seems like an impossible problem. So all known techniques gain speed through a well thought out cache-hierarchy.

Level	**1**	**2**	**3**	**4**
Name	registers	cache	main memory	disk storage
Typical Size	< 1 KB	< 16 MB	< 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	5,000,000
Bandwidth (MB/sec	20,000-100,000	5000-10,000	1000-5000	20-150
Managed by	compiler	hardware	operating system	operating system/operator
Backed by	cache	main memory	disk	CD or tape

Figure 2.5: A modern cache hierarchy[5]

A well designed hierarchy is often divided in several levels with specific characteristics at each level. The level closest to the CPU is actually the register file which is often forgotten and left out from the hierarchy discussion. The register file holds all registers for the processor that is used in calculations. However, all registers are seldom created equal. One of them might be used for a constant zero. A few others might be used for jumps and other context switching tasks. But the majority are often general purpose registers. The register file is a low latency, high bandwidth cache putting out data every cycle at the same pace as the core clock. Next level is the Level 1 (L1) cache. It is the next closest cache and is also a fast paced cache that is located inside inside the core capsule and runs at the same clock speed as the core itself. It is a low latency, high-bandwidth memory that has somewhat longer access time than the register file but access to it can still be pipelined. It is a general cache that can have many properties. It can be unified or splitted in the instruction/data sense. Since there is a task of finding your data in the cache there is also a tag memory that you compare with the address to know if you have the right data there or not.

When designing an L1 cache there are several big questions that comes to mind. First of all is the cache replacement policy. What do we evict from our cache to fit new data? A cache can not hold an infinite amount of data. Least recently used? Random replacement? Last recently used? Also of big importance is the cache associativity. Where do we place data in a cache? All the choices affect both complexity on chip and speed of different programs that uses the cache differently. The goal is of course to provide the best performance with minimal complexity which results in a fine balance between the two.

Level 2 (L2) is a bigger and slower cache just above the L1. In modern CPU's the L2 is located on-chip which usually enables the L2 to run at CPU-frequency. Most of the rules for the L1 cache applies in exactly the same way for the L2. They are siblings in functionality but located at different levels with different latency and size capabilities.

And the chain of cache levels goes on. Many CPU's include a Level 3 (L3) cache. They are often located off chip and does not run at CPU-frequencies. They can be several megabytes large and are still quite fast. These type of caches are usually found in large server environments where heavy context switching tasks are common. From the L2 and upwards you usually find the main memory and memory controller. Modern processors have a very complex memory controller to optimise the memory access. This is done by prefetching data before the processor requires it and having store buffers to generate a direct return on write to the previous level. Although many architectures support prefetch instructions the processor and code seldom generates those type of requests. So the memory controller is often doing a non clue-based form of prefetching. Much like an intelligent form of mini-cache. The use of such a memory controller is to hide the latencies involved in accessing main memory, instead of a local cache, as good as possible. There are a lot of other techniques to hide memory latency and increase bandwidth, for example Bank Interleaving or just wider memory buses. As usual there is a tradeoff between complexity and speed. The sweet spot can be quite hard to find and varies in time with technology and economical resources.

At the end of the memory hierarchy there is a non-volatile storage medium to be found. In desktop PC's it is the hard drive with a part of the media allocated for virtual memory operations. Most of the code is also loaded from the hard drive at boot-up. But there are other non-volatile storage mediums around. Many servers have started to use a solid state memory located in a network attached storage (NAS) environment (or locally) to burst data from. These solid state drives are much faster than normal magnetic media. Especially in the area of reducing latency.

2.1.4 Memory management units

Somewhere along the path of computer evolution it became infeasible to dedicate the entire processor address space to all processes without any restrictions over what they could do with that space. At any time, in any computer, where there are multiple processes running and trying to do their task, it would be to expensive to dedicate a process the entire physical memory range. Especially since most processes only use a very small part of the address space to do their calculations in. So it became natural to segment the physical memory range into smaller blocks and allocate them to different processes. Equally natural became a protection scheme for all the segmented blocks to protect them from being claimed by unauthorised programs in the same system. This proved to be a very effective way for an Operating System (OS) to maintain control over hardware and processes in the system. This is an abstraction that leads to the wakeup of the OS each time a faulting access to a memory segment has been made. This type of Virtual Memory (VM) and interrupt based process rescheduling are the bases of most modern OS.

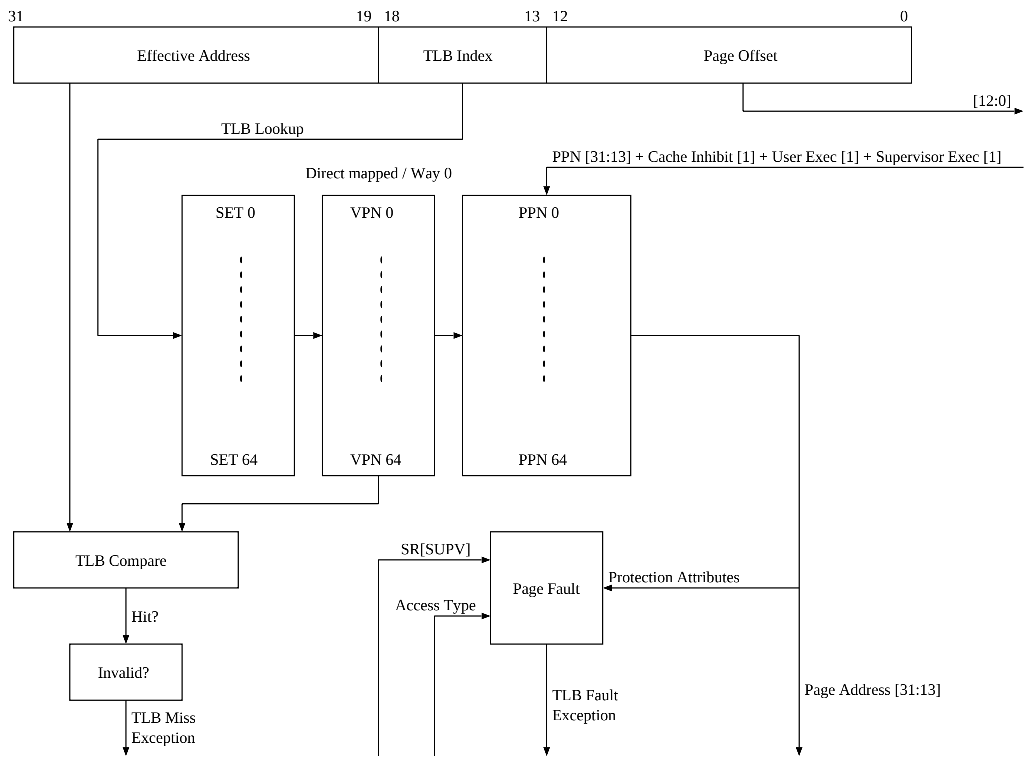

Figure 2.6: OpenRISC1200 Direct Mapped, Software Tablewalk IMMU

But since the MMU's keep track of the segmented physical memory it also became useful to allocate segments on a storage medium and swap in those segments when needed by the processor instead of flushing memory of important contents and reloading raw unprepared data from the storage medium once again. So instead of doing this time consuming task the OS swaps the already prepared segments to disk just waiting for them to be reused in their already prepared state. It saves a lot of processing time for the system and the feature was actually the most common used one for some time before the entire memory space got protection rights. This was necessary to alleviate the programmers of the burden of having to check if the main memory was exhausted. A not to seldom error message, *Out of memory* appeared on several OS before they got this feature. When the feature became a standard it released programmer productivity that had been hampered by the incapabilities of the OS.

2.2 Studied processor architectures

In this section we look at what processor architectures are viable and technically interesting. The sheer amount of architectures available does present a screening dilemma. It is quite hard to get information about all the available architectures so that a valid performance and technical evaluation can take place. And for practical reasons it is very hard to do tests on a large amount of architectures for comparison. So we have to resort to white-paper information.
Embeddable properties, technically or historically interesting solutions have been the criterias for choosing the well known architectures below.

- **Alpha:[6]** Nonexistent in embedded applications. For technical reference only
  How technologically superior the Alpha might have been, it was never intended for the embedded market. Clean design and advanced technology targets this CPU against blazing fast performance only. The only thing that stands in its way for scaling in performance is that the legendary Alpha design team have been bought up by different companies and are now spread over the entire world doing work in other teams. Resources have been allocated away from this architecture and it will most probably fade away soon in favour of newer architectures.

- **ARC:[8]** ARCTangent-A5
  The ARCTangent-A5 is a direct competitor with Tensilicas XTENSA V. ARC is one of the new players on the market with softcores and very flexible configuration. Both speed and energy efficiency are in the same department as the XTENSA but the ARC resorts to a more common 32-bit or 16-bit instruction set instead of the odd 24-bit instruction that XTENSA V uses. Both ARC and Tensilica sell these blocks as softcores with special instructions if there is a wish for such from the customer.

- **ARM:[9]** Intel StrongARM, Intel PXA XScale, Motorola Dragonball MX, ARM 10, ARM 11
  ARM is a very strong contender in minimalistic embedded devices running on battery power. Typical for the ARM architecture is that it also has very high computational power per energy unit, something that is most preferable in battery powered units. The ARM architecture is pretty clean and is available in 32 and 16-bit versions (the THUMB-ARM) which can reach pretty high frequencies. Up to 1.2GHz ARM-10 Full Custom has been noted by Samsung. ARM Ltd. itself have noted 400MHz+ from their new ARM-11 softcore.

- **CRIS:[10]** CRIS v10
  The CRIS is a 32-bit architecture with 16-bit instructions defined and built by Axis Communications. It suffers from bad scalability due to architectural scalability issues. It has mediocre computational power but is quite energy efficient and produces comparably small code. The CRIS itself is often included in very competent and good priced packages with lots of I/O possibilities.

- **DLX/MMIX:[12]** Computer Science Undergrad Classic Study Architectures
  MMIX and DLX are two fictional architectures. They are fictional in the sense that they have never been implemented as physical devices and therefore have not been validated in physical operation in any way or form. They are both used

to teach undergrad CS students Computer Architecture. DLX is a classical 5-stage pipeline with a clean design and a well defined ISA. The MMIX is a 64-bit clean device with abundant resources for high powered computation with the classical RISC in mind. The architecture has a well defined ISA but the actual implementation is left to the reader as an exercise. Both are free for any type of use and carries no royalties or patents covering them. We have included them here for reference since DLX is really interesting for study purpose only but MMIX however, still proves to be an interesting 64-bit machine.

- **FCPU:** FreeCPU
  FCPU is a very small project that defines an open architecture and invites everybody to join in for development. The project has not come far and is considered an non viable alternative today.

- **m68k:** Motorola V4E, Motorola 68060
  A classic architecture is the Motorola m68k. Still widely used in tiny embedded applications such as microcontrollers it has however reached it's end of life. The architecture is not developed for speed reasons any more. The fastest CPU in the m68k series is the 68060 which reached around 60 MHz. All of the Motorola CPUs above 68030 do contain a MMU and can be used in an OS for full scale tasks. They are not particularly fast or energy efficient and belong to the relics of a forgotten CISC era.

- **MCore:** Motorola MCore
  Motorola Mcore should perhaps be regarded as a development from the Motorola ColdFire which was developed from the m68k. They are not the same architecture yet they have many similarities. The MCore architecture was supposed to be targeted against the missing segment in Motorola's mobile phone division and applications like such. Powerful, small and very energy efficient the MCore competes with the 16-bit THUMB instruction set from ARM in both code size and efficiency. But with time the MCore was forgotten and is not used much today.

- **MIPS64/MIPS32:[11]** NEC VR7700, NEC VR5500, Sandcraft SR71010A, PMC Sierra RM7065C, BMC 112X, MIPS 20Kc / MIPS M4K, AMD AU1000, Lexra
  The MIPS family is known for their clean design and high computational power. It's not particularly energy efficient but carries high computational power per cycle. It's available in both 32 and 64-bit versions and has been widely adapted by many companies much like the ARM. The MIPS is popular among very high speed network products and a few other embedded areas. A quick note goes to Lexra, a company which tried to build their own MIPS32 device and got sued by MIPS Intl. for patent infringement and later settled with MIPS by becoming a licensee of MIPS32.

- **OpenRISC 1000:[14]** OpenRISC 1200
  The OpenRISC 1000 architecture is a new fresh and open initiative by a free organisation named OpenCores. It aims to provide a flexible and clean ISA with all the basic functionality in place. It is a well defined architecture with a working implementation available. The most prominent benefit of the OpenRISC architecture is the openness towards new developers. Everybody is free to add new components on a semi-restricted basis. All in all, it's a reasonable choice for a free architecture.

- **PPC:[20]** IBM 750CXE, IBM 750FX, IBM 750GX
  The PPC is a prominent and powerful architecture. Interesting is that the PPC has always provided high computational power per energy unit consumed. For example. IBM's latest 750GX is a 1GHz, 1M L2 device which typically consumes under 8watts of power. The PPC has always been a strong contender in the high-performance embedded segment. The PPC is available in both 32-bit and 64-bit versions of the ISA.

- **SPARC:[16]** ESA SPARC Leon, Fujitsu SPARClite
  SPARC is owned by SPARC Intl. and is a patented architecture but open for use. SPARC in general should not be confused with Ultra-SPARC which is owned and patented by Sun Microsystems. Sun Microsystems are often associated with the SPARC brand name but are themselves a licensee of the SPARC v8 and v9 standards from SPARC Intl. The SPARC v8 is a fast and clean architecture that is rather old but proven. SPARC v9[18] is the 64-bit extension of v8. It's not particularly energy efficient, much like the MIPS. The most usual implementation of the SPARC v8 architecture comes in the classical 5-stage form and hits the limits in frequencies rather quickly. The architecture itself could go a quite bit higher but no such projects exist today to our knowledge. The ESA SPARC Leon was implemented by Jiri Gaisler on behalf of the European Space Agency. It's main purpose was to provide a clean and widely used architecture at ESA with built in fault tolerant logic. Instead of producing a processor in an extremely expensive radiation tolerant process technology the choice fell on providing fault tolerant logic in a standard process technology. The ESA SPARC Leon is a very configurable processor that has an architecture which has been proven over and over again.

- **SuperH:[19]** Hitachi SH4, Hitachi SH5
  Hitachi's SuperH architecture can be found in a couple of places. Most notably in the Sega Dreamcast which has now reached it's end of life. The SuperH architecture was primarily designed as a media processor for embedded application and does serve it's purpose well. It is not very scalable due to it's specialised architecture. It produces quite small code sizes for a 32-bit machine with 16-bit instructions and is pretty energy efficient for the number of calculations performed.

- **x86:[21]** VIA Nehemiah, Transmeta Crusoe, Intel Pentium III Mobile ULV
  Intel x86 can be considered the worlds most popular ISA. With bases in the CISC era this ISA is very cluttered and ugly. It was never designed for speed but has survived for practical reasons and a huge code-base. It produces relatively small code but it's not particularly energy efficient when scaling in frequency. The architecture is kept alive due to the enormous amounts of brain power put down in it's survival. There are however a few very interesting implementations that reign in the x86 embedded segment. The Via Nehemiah is a very small core but yet energy efficient and quite fast. Intel have their own offerings in form of the old Pentium III Mobile Ultra Low Voltage CPU. The Intel CPU is quite energy efficient as well, but not as small as the Via CPU. Transmeta produces a very odd beast for the x86 market. The actual core is a VLIW instruction machine which translates to x86 instructions in software on the fly. The Crusoe is very energy efficient and a very interesting CPU.

- **XTENSA V:[22]** Tensilica XTENSA V
  Tensilica is a new player in the game with the XTENSA. It features an odd 24-bit instruction size that is clean, very flexible and powerful. It is available as a softcore with respectable speeds. Features are as all softcores come, very flexible, very configurable and generally nice properties. The architecture is quite energy efficient and can easily be made more efficient for different purposes by including special instructions and special hardware to deal with certain problems.

2.2.1 Industrial trends

When designing a next generation processor, there is a need to be very aware of the tangent of current developments within the industry and where it is taking technology into the future. There are of course many topics to be touched when trying to look into the future. It's quite easy to say that everything will become more complex, bigger, faster with more features. This is a correct observation when looking at our history so far. But the difficult part is not the assessment of bigger and faster but rather how things will become bigger and faster. What technologies will be used, why and how? Are they a natural step in the development? If not, what is?

First, a quick look into our history of I/O and RAM protocols. In the beginning there were a lot of serial protocols which co-existed with many parallel protocols. Then came a time when speed could not be reached with serial protocols. Serial was left behind for the simple and slow protocols, PS/2, RS-232 etc. Parallel was the way to go i.e IDE/ATA, SCSI, PCI, SDRAM, etc. But parallel technology is inherently hard to push at high frequency because of electrical interference and skew problems with the other parallel signals. During the last 5 years a revolution has begun in the segment of I/O-protocols. What we have now is USB, USB 2.0, Firewire A and Firewire B and we are looking at the coming of protocols such as Serial-ATA, SA-SCSI, Hypertransport and RAMBUS. RAMBUS was considerd for the PC-market, but patent and royalty disputes between RAMBUS Inc. and other DRAM technology manufacturers drove them off the market. RAMBUS technology however, is very potent, powerful and nowhere near dead.

So with that in mind we are seeing a revolution for serial protocols. From serial to parallel and back to serial protocols again. Of course the most extreme form of complex protocols utilise both technologies. But constructing hardware for the future should perhaps be tuned to deal with a more high frequency and narrow data-path rather than a broad and slow data-path.
When looking at the CPU itself we also see a few emerging trends. In the CISC era and the infancy of the modern CPU the processor itself was a very slow unit. Yet machines where built that did amazing things for their time. A notable example is the Commodore C64 and the Commodore Amiga series for example. Not exactly blazingly fast processors even for their time. The trick to their success was the many co-processors. Dedicated graphical units, sound units, I/O units, DMA units etc. The CPU itself did not need to do that much work because it was offloaded.
However, CPU's became faster and began to do work that the co-processors used to do. This method worked for a short while before the needed computational power by the co-processors outgrew the main processors capabilities. Today we are back at the co-processor state. The load from specific tasks has since long outgrown the processor especially in the embedded segment where the processor usually is not that powerful.

Today we are looking at 3d-processors with texture units, vertex units and pixel units, network chips that can offload hard work from the TCP-stack itself as well as common cryptography used on the Internet, audio positioning processors and protocols etc. History repeats itself once again.

While the number of co-processors have grown, the CPU itself has more or less stayed the same. The same on the outside at least. CPU's have moved from a RISC era to a Post-RISC era. The external ISA remains the same throughout several generations as the code base is far to big to revamp over night. The most stunning example is the old x86 ISA which is kept alive by Intel and AMD. The exterior of the modern x86 CPU's like the Intel Pentium 4 remains x86 but the interior is a completely different beast. The P4 translates the x86 instructions to it's internal format of micro-ops and executes them before writing back the results. More or less every modern CPU is a Post-RISC machine with an internal format specific to the major revision of the core.

A quick summary on the status of the microprocessors today:

- Modern processors are fast. It's becoming harder and harder to push in the same pace that Moore predicted.
- Current leakage, heat production and energy consumption are making themselves more and more pronounced.
- Frequency scaling, voltage regulation and aggressive throttling as an direct result from the already mentioned problems.
- Frequency is not the same dominant factor for success as it has been before. Frequency costs energy.
- All-round efficiency is becoming more and more important. Technologies like SIMD and VLIW are becoming standard in both vector/DSP units as well as general processors. Give the transistors a run for their money.
- Multi-threaded demands on execution are beginning to appear from the software side.

2.2.2 Free or open architectures and performance claims

There are basically 3 free and open architectures on the market today. The ESA SPARC Leon with the SPARC architecture. The OpenRISC 1200 with the OpenRISC 1000 architecture and the MMIX.
The last architecture, the CRIS, is free for use by Axis only since it is not released under any free license.
OpenRISC 1200:

+ Verilog, lots of possibilities to configure.
+ Open architecture, already implemented.
+ Existing GNU tool-chain. Binutils, GCC, GDB, uClinux, uClibc, eCos. Linux is under development.
+ Existing simulator.

+ LGPL license (Lesser General Public License).
- Weak MMU, TLB, Cache.
- Somewhat spacious ISA.

CRIS (free for use by Axis):

+ In-house technology.
+ Existing and working GNU tool-chain. Binutils, GCC, GDB, glibc, Linux.
+ Already adapted to Axis applications.
- Not particularly scalable.
- Not particularly extendible.

MMIX:

+ Existing GNU tool-chain.
+ Open architecture.
+ Existing simulator.
+ Well defined architecture.
- Many instructions.
- Large by definition. 256 64-bit registers makes a 16 times bigger register file than a classical RISC and 64 times bigger than the CRIS.
- Never implemented.

ESA SPARC Leon:

+ VHDL, lots of possibilities to configure.
+ Already implemented.
+ Existing and working GNU tool-chain. Binutils, GCC, GDB, rtems, eCos, uClinux, uClibc, glibc, Linux.
+ LGPL license.
+ Implemented in other commercial purpose.
- Unknown future for the next generation from the developers
- No MMU as standard. Patch does exist.

Only two of the three free architectures have been put into physical devices. Namely the OpenRISC 1200 and ESA SPARC Leon. The MMIX exists only on paper and it is hard to estimate how an implementation would turn out. MMIX however, was meant to scale. But there are a few oddities in the architecture that could slow it down a little. The problem with the MMIX is that it was designed for high end computing only and never was intended for embedded use. Of course there is a possibility to down-scale

the MMIX but that would hardly make the MMIX better than the other choices.
Speed-wise, the Leon claims 165MHz in a 0.18 micron cell technology using worst case scenarios. The OpenRISC team have claimed 150MHz (250-300MHz in flyers and implementation manual, this number was later corrected to 150MHz in a discussion with the OpenRISC team) in a 0.18 micron cell technology using worst case scenarios. Both CPU's reach approximately 0.8 to 0.9 dhrystone 2.1 MIPS per MHz.

2.2.3 Free or open busses and performance versus complexity

The industry has a plethora of buses to connect cores and peripherals together. But as usual, all of the buses are based on IP-blocks that are owned by some company. There are however a few exceptions to this. There are 3 available buses that are free for use without any costs involved.

ARM AMBA:[23][24]

- AHB - Advanced High-Performance Bus
  - 32-128+ bit bus width with up to 16-beat bursts.
  - 32-bit address space. Access protection mechanism.
  - Multi master, split transfers.
  - Single cycle bus master hand-over.
  - Arbitration support (REQ, GNT, LOCK).
  - Data throttling.
- APB - Advanced Peripheral Bus
  - Low performance, low power.
  - Single master.
  - Very simple, 4 control signals plus clock and reset.
  - 32-bit address space.
  - Up to 32-bit data bus.
  - Separate read and write data bus.

IBM CoreConnect:[25]

- PLB - Processor Local Bus
  - Overlapped read/write (2 per cycle).
  - Split transfer. Address pipelining.
  - Separate read/write data. 32-64+ bit data bus width.
  - 32 bit address space. 16-64 byte bursts.
  - Supports unaligned and 3 byte transfers.
  - Arbitration (REQ, GNT, LOCK).
  - Late and hidden arbitration with 4 levels of priority.
  - Special DMA-modes. Fly-by and mem-to-mem.

  - Address and data phase throttling.
  - Latency timer.

- OPB - On-Chip Peripheral Bus
  - Multiple masters, 32-bit address space.
  - Separate read/write data bus. 8-32 bit data bus.
  - Dynamic bus sizing, retry support, burst support.
  - Devices may be memory mapped.
  - Bus timeout function.
  - Arbitration support (REQ, GNT, LOCK).
  - Bus parking support.

Silicore Wishbone:[26]

- Wishbone B3
  - One bus for all applications.
  - Simple, compact architecture.
  - Simple timing.
  - Single clock transfers.
  - Handshaking between cores allow throttling.
  - Multi-master support.
  - Arbitration defined by the end user.
  - 64-bit address space
  - 8-64+ bit data bus.
  - Single read and write cycles with burst support.
  - Read-Modify-Write cycles, Event cycles.
  - Supports retry.
  - Supports memory mapped, FIFO and crossbar interface.
  - User Tag's for identification of data transfer types.

By inspection one can see that IBM's CoreConnect is the most advanced of the three buses. It can fill every need of a modern computer and is truly a powerful bus. ARM AMBA is also a very complete bus that covers every need within a SoC and it's used by the SPARC Leon. Both of them require a bit more than most embedded devices need. The Silicore Wishbone is a simple but yet quite a powerful bus. It is restricted to single cycle transfers and cannot handle more advanced split transfers. But for most applications this is perfectly fine. Wishbone is used by the OpenCores OpenRISC 1200.

2.2.4 Historical performance of different architectures

The feasibility of the the frequency goal must of course also be assessed. Therefore it can be quite interesting to review the historical performances of different 5-stage pipeline architectures to see where the architecture sets limit for scaling. Of course this type of historical evaluation is highly technology dependent. But you can still get some clues from watching historical speedups in frequency.

Processor	*Frequency*	*Stages*	*Technology*	*Availability*
Intel Pentium	200 MHz	5-stages	0.35	1996
Intel Pentium MMX	266 MHz	5+-stages	0.25	1997
AMD K5	120 MHz	5-stages	0.35	1997
IDT Winchip C6	240 MHz	5-stages	0.35	1997
Motorola PPC 7400 (G4)	500 MHz	4-stage w-pipe	0.22	1999
SPARC 64	141 MHz	4-stage	0.40	1996
MIPS R10000	275 MHz	5-stage	0.35	1998
ARM 920T	375 MHz	5-stage	0.13	Now
ARM 940T (920T embed.)	180 MHz	5-stage	0.18	Now
HITACHI SH7750	200 MHz	5-stage	0.25	Now
HITACHI SH7750R	240 MHz	5-stage	0.18	Now
IBM PPC 405LP	266 MHz	5-stage	0.18	Now
Fujitsu FR500	266 MHz	5-stage	0.18	Now

Table 2.1: Historical and Current Custom Core Processors

All these custom cores show that it is very difficult to hit 400MHz with a 5-stage machine using a standard cell library. The processor closest to that goal is the ARM 920T with 375MHz at a 0.13 micron process technology. It is necessary to remember that all these cores have specially designed caches. They have reached their target with custom parts.

Processor	*Frequency*	*Stages*	*Technology*	*Availability*
MIPS32 4KP	210-255 MHz	5-stage	0.13	Now
MIPS32 4KP	160-190 MHz	5-stage	0.18	Now
ARC tangent-A5	170MHz	4-stage	0.18	Now
ARM1026EJ-S	266-325 MHz	6-stage	0.13	Now
ARM1136-JF-S	400MHz	8-stage	0.13	Now
Tensilica Xtensa V	260-360 MHz	x-stage	0.13	Now
MIPS64 5Kc	270-326 MHz	6-stage	0.13	Now
MIPS64 20Kc	533MHz	7-stage	0.13	Now

Table 2.2: Current Softcore Processors

The current state of softcores shows that there are difficulties reaching above even 200MHz using a 5-stage design. Most of the cores doesn't even define the caches used for reaching specific speeds above 250MHz. The manufacturers often state a speed that has been reached with specially selected caches for the purpose of reaching their speed target only. The cores themselves are capable of speeds above the current soft-core limit. For example, the MIPS 20Kc have reached speeds in excess of 700MHz using carefully selected parts and manufacturing methods. So the memory banks set the current limit. And the price of specially manufactured cache memory blocks is high.

Chapter 3

The choice

3.1 The cores

Only two of the free cores are actually feasible to do something useful with. This conclusion can seem to have come a little late and should have been quite obvious from the start of the thesis. But since this is a thesis and we are doing this for educational purposes we decided to learn a bit about other architectures as well.
The two competitors are the ESA SPARC Leon and the OpenCores OpenRISC 1200. Both of them are simple RISC CPUs with classical attributes to their ISA. The SPARC v8 intellectual property is owned by SPARC Intl. and the implementation is owned by Gaisler Research under a LGPL license. The OpenRISC IP is owned by by the authors Damjan Lampret & assoc. under a LGPL licence. A quick rundown of the implementation and architectural features of the cores.
OpenRISC 1200:

- Architectural:
  - Load-Store architecture.
  - Harvard architecture.
  - 32-bit, 32-regs.
  - Big Endian architecture.
  - Memory address mode: Register indirect + 16-bit sign immediate.
  - Memory address mode: PC-relative.
  - Branch delay slot.
  - Jump mode: Register indirect.
  - Jump mode: PC-relative.
  - 8k page-size.
  - 8k/16M/32G 3-level MMU. Software table-walk on miss.
  - 32/64-bit virtual address, 35-bit/64-bit physical.
  - MMU support for multiple contexts and full page protection.
- Implementational:
  - Written in Verilog.

- 5-stage pipeline.
- OpenCores Wishbone B3 Bus.
- Cache: 1-way, 8kB Data + 8kB Inst, 16b CL-size, physical tag, LRU, Write-through.
- No cache coherency.
- No cache-line prefetching.
- Hardware multiplier and MAC. Hardware single-step divider.
- Power management unit.
- No clock gating.
- Advanced SDRAM and Flash Memory Controller available.
- PCI-bridge available.
- Ethernet available.
- AC97 available.
- USB 1.1 PHY and controller available.
- UARTs available.
- Tick-timer unit: Max-timer 32-bit cycles, maskable 28-bit between interrupts.
- Simple JTAG Debug unit with more complex developer interface.
- GNU-toolchain.

Sparc LEON:

- Architectural:
  - Load-Store architecture.
  - Harvard architecture.
  - 32-bit, 24-reg per window + 8 globals.
  - Address mode: Register indirect + immediate.
  - Address mode: Register + Register.
  - Big Endian architecture
  - Branch delay slot.
  - 4k page-size
  - 4k/256k/16M 3-level MMU. Hardware table-walk on miss
  - 32-bit virtual address, 36-bit physical.
  - MMU support for multiple contexts and full page protection.
- Implementational:
  - Written in VHDL.
  - 5-stage pipeline.
  - ARM AMBA-2.0 Bus.
  - Cache: 1, 2 or 4-way split Data and Inst, 1-64KB set, 16-32 CL-size, Write-through with lock-bits. LRU, LRR or Random.

  - Cache coherency.
  - No cache-line prefetching
  - Hardware multiplier and MAC. Full hardware divider.
  - Simple power-down mode.
  - Memory controller slave on AHB-bus.
  - 2 Interrupt-controllers 15 + 32 interrupts.
  - PCI-bridge available.
  - Ethernet available.
  - Two 8-bit UARTs.
  - 32-bit parallel I/O-controller.
  - FP and CP interfaces.
  - Tick-timer unit: 2 Tick-timers 24-bit + WDOG 24-bit.
  - Full RS232C Debug unit.
  - GNU-toolchain.

Both cores have more or less the same features. The OR1200 shortcomings lies in its weak implementation of caches and MMU's. The OpenRISC architecture specifies more complex MMU's and caches but they are not implemented. Features such as lock-bits, cache-line prefetching, invalidation, coherency, and context id's are specified. The OR1200 strength comes from the backing of a community with many functional units that are ready to be attached to the processor core.

SPARCs strength comes from a very complete and good implementation. Both caches and MMU's are very solid since they have their base in the SPARC v8 standard. It has most features that could be wanted from an embedded processor.

3.2 Reasoning behind the choice

From all potential CPUs, excluding IP-block that are for sale from commercial companies we have narrowed the choice down to two CPUs. Both CPUs are excellent choices for future considerations of embedded applications. SPARC has a excellent track record and everything is available and ready today for embedded applications. GNU toolchain, Linux, all tools that compile for the standard SPARC architecture. The code-base is huge and verified. So it's an excellent potential that a SPARC processor has to offer Axis Communications. The SPARC history, going back all the way to 1984 for the SPARC v7 standard and 1990 for the v8 standard, has set the pace of an entire computing industry at times. The SPARC Intl. group has however control over the SPARC trademark and memberships can be bought by companies with interest in the architecture. For 100,000 U.S dollars a year for a full membership you can have seats in the Board Of Directors and Architecture Committees. Yes, the architecture is free for download but it is in the hands of a commercial governing body.

The OpenCores choice is on the other hand completely free. There is no absolute governing body that controls it as a trademark nor it's future or who gets to decide what goes in and what goes out. There is a community that places trust in certain developers who have proved themselves with both technical excellence and moral excellence to govern the OpenRISC future. Both the architecture and all the implementations carry a open LGPL license or a license of equal power to protect the code from commercial ownership. OpenCores is a free community which ideology is that as much of the hardware code that can be free should also be free. And the community thrives and grows. With more and more developers and projects opening each day it is only a question of time and maturity before they get enough momentum to launch a full scale revolution against many IP-block holders and manufacturers.

There is also a minor technical issue with the SPARC. It's standard register window size is not suitable for embedded purposes. Register windows are useful for function ingress and egress behaviour but tend to cost a bit more during context switches since the used part of the register window needs to be pushed to memory. The register window is considered hairy by many and sometimes much more of a problem-maker than a problem-solver. You have the possibility to reduce the window-size to one window only resulting in 8 global, 8 local, 8 in, 8 out registers. 32 in total. This method however has a tendency to cause breakage and general havoc in operating systems as well as applications like compilers etc. This breakage is because all applications that confirm to SPARC v8 expects a register window of a certain size. The issue should however be considered minor since embedded processors often run the same code and therefore you have a opportunity to test all code that will run on the CPU thoroughly.

In the end, we choose the OpenCores OpenRISC 1200. Why this sudden and seemingly rushed decision? First off, it was not rushed. We took quite a long time analysing the two cores in many areas. In the end, we felt that OR1200 had a lot to offer for the future. Not because it is more revolutionary than the SPARC or remarkably better at executing faster, cleaner or more efficient but rather for it's community and what a totally free community can offer you in the long run. This however does not make the SPARC Leon a bad choice for future applications. On the contrary, it is an excellent CPU with exceptional quality. But we have to consider one of the two CPUs given our short timeframe.

3.3 The OpenCores community

www.opencores.org is the home of one of the worlds biggest communities of free IP. It has many members and the list of members keep growing for each day that passes. Many registered FPGA or ASIC projects with different purposes have their home there. So why do we need something like OpenCores and their designs?

Because through open designs, people start cooperating to produce extremely good designs rather than competing. People can develop and customise new designs, devices and tools upon their own need. Open designs can be used for educational purposes that will lead to improvements of the design. Students can show and prove their ideas by designing at OpenCores. Open designs will also help people to debug their systems and to fix them easily. The development time of new systems can be reduced by using reusable verified open design cores. Retired, work-less, enthusiastic people, and anybody who has some free time can spend it in making things good for himself and everybody, yet protected by a Free License.

When commercial companies make their designs open, anyone can study it, test it and conclude few things that can improve the design by supplying their comments to these companies. Also by making the design of their appliances open, anyone can repair it. Unfortunately, this is not common practice. There are a lot commercial management ideas that have proved their failure over time. Such ideas like the paranoia which makes companies divide designs between teams so that no single team could recreate an entire design for espionage reasons. This is a bad idea and may produce bugs in the design because each team does not know enough about the work of the other teams. By also making small closed groups working together without sharing the ideas with the public may lead to even buggier designs. As a result the OpenCores methodology can reduce these kinds of bugs.
There are also other advantages for open hardware design in business. For example, design cost is reduced when open hardware design concept is adopted. Because the design, verification and debug is shared between many designers and manufacturers which reduces the over all time needed to reach a final product state.

The negative side of open development is of course that there is no driving commercial interest in the product. Commercial products are often very viable and rewarding for the company that manufactures and sells them. If there is a commercial drive behind a project you also often have guarantees that the product comes with. You can also buy support deals with the first party vendor if you really need it. This type of arrangements are very often needed in development of new commercial products so you can be assured that each party holds their part of the deal that has been made.
This type of arrangements can't be had from open source development. At least not from the first party. You could always build deals with third party vendors (like Redhat Linux) and work from there. But the first party is generally not available for such deals.
In the last few years the hardware resources has increased and become available for hardware designers at a low cost, which have made the designers slack about many issues they used to reflect about many years ago, like hardware resources and timing optimisation. The same problem is now facing software programmers when they do not care about memory usage nor processor power.
As a contrast to this, open hardware design methodology will lead to make designers consider these facts again because of many improvements made by many designers and the limitation of the hardware resources available for open designs.
With a quick summary, this type of design methodology will lead to improve the overall technology by allowing rapid prototyping, testing and lot of feedbacks from the designers which in turn is generally a positive behaviour for everybody.
The defined goals of the OpenCores community:

- To define OpenIP and OpenHW.
- To define a license for cores and designs.
- To define a methodology of design and way of standardisation the design interface.
- To encourage designers to use Free tools and to put their designs in the form of free tools format. Also to convert already made designs to Free tools format.
- To group lot of designs under previously mentioned license and available free tools, to make it easier for the designers to choose between them.

- To search and define what circuits, cores and software tools are needed and start new sub-projects for them.
- To build tools, supporting designs and documentation to already existing designs, such as compilers, drivers and external circuits.
- To give support for anyone who needs some designs, cores and tools or even documentation.

3.3.1 Suggested proposals between the developers and Axis

During our thesis we have been in contact with the core OpenRISC developers. Without any doubt they have been extremely helpful, polite and generally nice people to deal with. Despite the sometimes odd questions that we have asked they have kept their calm and answered professionally. If they where a support department for a commercial product they would have received the highest ratings for technical knowledge, availability and helpfulness. We are pretty sure that any group outside of OpenCores will find the OpenRISC developers a very good group to work with.
Since the OpenRISC team is looking for partnership with foundry's and design-teams outside OpenCores it was natural that Axis Communications would be asked if they where interested in teaming up to promote the usage of the processor with their well known industry name.
We believe that this could be a very viable relationship that Axis could decide to take on in the future.

3.4 Architectural quality

There is a lot to be said about the OpenRISC 1000 architecture. The OR1000 architecture supports several extensions of the ISA.

1. ORBIS32 - OpenRISC Basic Instruction Set. 32-bit.
2. ORBIS64 - OpenRISC Basic Instruction Set. 64-bit.
3. ORFPX32 - OpenRISC Floating Point eXtension. 32bit.
4. ORFPX64 - OpenRISC Floating Point eXtension. 64-bit.
5. ORVDX64 - OpenRISC Vector/DSP eXtension. 64-bit.

The OR1200 is made with the ORBIS32 extension in mind. It does not support any other extension and probably never will. Since the OR1000 architecture is a 32-bit architecture the ISA extensions always operate on 32-bit instructions and handling 32 or 64-bit worth of data. So the ISA extensions does not imply 64-bit instructions, just data.
The register file is thus always maximum 32-entries and 32 or 64-bit large. Smaller register files such as a 16-entry register file can be used if it is desirable. There is also a possibility to implement a shadowed register file if needed for fast context switches. The flexible architecture does not define hardware solutions for everything on purpose.

For example, the tablewalk of the MMU's on a MMU miss is left as a software tablewalk. Of course there is the possibility to define a hardware tablewalk if necessary.
OpenRISC 1000 is by default a Big endian architecture but the manual states that both big and little endian are supported if byte reordering hardware is implemented. The byte ordering of the architecture is controlled by a supervisor register flag.
The entire architecture is very clean and flexible. All 5 extensions to the architecture are closely explained in the architecture manual, both implementation, structure and exceptions. Both caches and memory handling in general are very well defined.

3.4.1 Architecture road-map

The road-map describes what the OpenRISC team have in mind for the architecture in the future. There has been quite a few discussions on where to take the architecture and what people want to use in their projects. The result has become a road-map which shows where to go in the future.
We had quick discussions with both the OpenRISC team and Jiri Gaisler. The OpenRISC team where very quick to point out their current road-map. Jiri however, has not provided any road-maps of his future work with the SPARC architecture. This is quite a shame since it probably makes other teams look with uncertainty at future possible implementations.

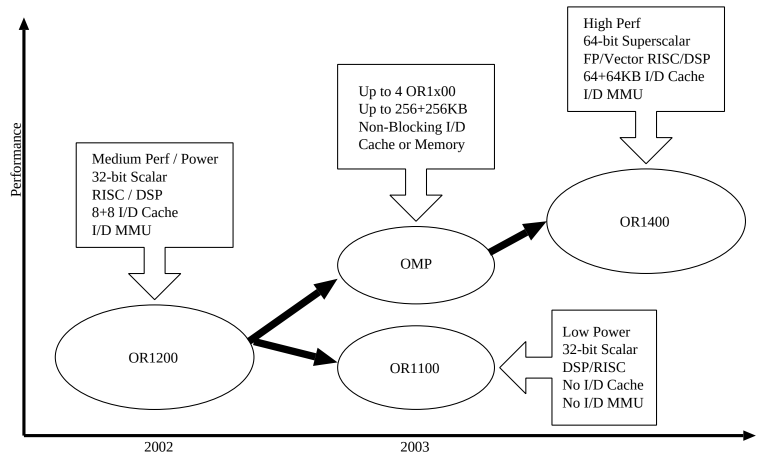

Figure 3.1: OR1000 road-map

For a company like Axis then the OR1400 and further is probably the most interesting CPU.

3.5 Tool-chain quality

There are other important aspects to a computer architecture besides the actual hardware description and the hardware implementation. You probably also need something to run on your new hardware. You need at least three components. A compiler for a specific language, an Operating System that has been ported and understands the layout

of your hardware and a userland library that can interface your programs as well as the OS and understands both of them. There isn't always a need for the userland library. But having such a library makes life easier on the programmer from the userlevel, kernel functions can often be complicated and hard to deal with. And living without a uniform behaviour from the userland applications in form of an intermediate library, could potentially wreck havoc in the OS.
So far, all of the free architectures have supported the GNU toolchain for building and debugging application. This is a very good thing since the GNU toolchain is a very feature rich and stable toolchain for these purposes. The tools included are GNU CC a.k.a. GCC, GNU Binutils which is a collection of useful tools for binary manipulation and the GNU Debugger a.k.a. GDB.
The MMIX and CRIS and the x86 tools will be left out from the quality discussion since none of them are considered a choice for a future target.

Inarguably, the SPARC toolchain is by far the most complete and stable of the two

Processor	*Target and version*
Axis CRIS	elf32-CRIS / r53
Knuth MMIX	elf64-MMIX / 20020718(cvs)
OpenRISC 1200	elf32-or32 / 20030602(cvs)
SPARC Leon	elf32-SPARC/ 1.1.5.2
x86	elf32-i386 / x
x86	elf32-i386 / x

Table 3.1: Current Tool-chain state

Processor	*Binutils version*	*GCC version*	*Libraries*	*Operating System*
Axis CRIS	2.12.1	3.2.1	glibc	Linux
Knuth MMIX	2.12.90	3.2	newlib	x
OpenRISC 1200	2.11.93	3.1	uClibc, newlib	uClinux, eCos, rtems.
SPARC Leon	2.13	3.2.2	glibc	Linux, eCos, rtems, others.
x86	2.12.90.0.1	2.95.4	glibc	Linux, others.
x86	2.12.90.0.1	3.0.4	glibc	Linux, others.

Table 3.2: Current Tool-chain state cont.

toolchains. The SPARC has seen development for more than a decade on compilers, operating systems and various tools. The OpenRISC toolchain exists in two versions. One for the 3.x series of GCC by the OpenRISC team themselves and one external for the 2.95.x series by the University of Cantabria in Spain. The two compilers are now in a merging state towards one stable and complete compiler. But they do still have many issues to sort out, being a new architecture and all. The quality is slowly getting better. The toolchains that needed compilations compiled quite cleanly. A few warning at most but nothing that caused any major breakage. The SPARC Leon tool-chain came in a binary distribution so it was not possible to test a complete compilation of the various tools. The OpenRISC toolchain was built from source with the help of a simple guide on the OpenRISC site.

3.5.1 Codesize comparison

Codesizes have always been important for embedded devices. They usually have a relative small amount of memory to play with and an even smaller non volatile flash memory to store data in. Since flash memory is more expensive than general memory it is also interesting to see how compressed code behaves. For the comparison we have used the CRIS, which is a 16-bit instruction machine, the x86 which is a CISC variable instruction length machine. We have also used our primary and secondary target, the OpenRISC and the SPARC which both are 32-bit instruction machines. For comparison, we also included the MMIX which also is a 32-bit instruction machine.
All the machines are 32-bit wide for data except for the MMIX which is a 64-bit wide data machine.
For the purpose and validity of the test we tried to even out as many oddities between the architectures and compilers.
We chose to measure the programs in the following way:

1. Choose minimal common base to stand on. Eg *arch*-elf and no OS dependencies.
2. Choose compiler flags. -O2 should be considered GCC standard.
3. Choose compiler flags that makes the toolchains behave correctly for each architecture.
4. Choose not to link binaries. It is omitted since we want minimal library dependence.
5. Choose to strip binaries of unnecessary symbol data.
6. Choose to concatenate all ELF objects into one file and measure it.
7. Choose to use industry standard Lempel-Ziv compression (gzip).

Finding programs that would compile with minimal library dependence, minimal OS dependence and still being valid in the scope of the thesis proved quite difficult. In the end we settled for 3 programs and a simple testcase. The testcase is a hello world program. It is not included and was used for reference only.
Boa, the first of the three programs is a minimalistic webserver often used in embedded applications, for example Axis products. Hypercube is another minimalistic webserver that is about half the size of boa. Secure is a crypto program that is very OS independent and was chosen for that reason.

CPU/GCC Version	*Hypercube v.0.4*	*Boa v.0.94.13*	*Secure 0.34beta*
Axis CRIS/3.2.1	63328 bytes 0%	135076 bytes 0%	35504 bytes 0%
Knuth MMIX/3.2	(no system)	(no system)	65344 bytes +85%
OpenRISC 1200/3.1	77796 bytes +23%	166408 bytes +23%	42052 bytes +18%
SPARC Leon/3.2.2	83160 bytes +31%	174864 bytes +29%	43140 bytes +22%
x86/2.95.4	61724 bytes -3%	129860 bytes -4%	40360 bytes +14%
x86/3.0.4	59004 bytes -7%	(not tested)	(not tested)

Table 3.3: Code-size compared with CRIS

CPU/GCC Version	*Hypercube v.0.4*	*Boa v.0.94.13*	*Secure 0.34beta*
Axis CRIS/3.2.1	21717 bytes 34%	44009 bytes 33%	11396 bytes 32%
Knuth MMIX/3.2	(no system)	(no system)	13132 bytes 20%
OpenRISC 1200/3.1	25019 bytes 32%	52522 bytes 32%	14793 bytes 35%
SPARC Leon/3.2.2	23816 bytes 29%	49468 bytes 28%	12658 bytes 29%
x86/2.95.4	20940 bytes 34%	42973 bytes 33%	11427 bytes 28%
x86/3.0.4	20414 bytes 35%	(not tested)	(not tested)

Table 3.4: Percentual size of compressed code compared to the original

After a couple of sessions with the compilers the end results where a bit mixed. Uncompressed code sizes where approximately as expected. CRIS sharing the lead for smallest binaries with x86. The CRIS should be smaller by default since it is a 16-bit instruction architecture. But the small x86 binaries probably depends on that the GCC port for x86 is very good at generating code. OpenRISC and SPARC generates approximately the same code sizes and MMIX takes up the rear with it's large 64-bit data sizes.
Compressed codesizes tell a different story. Given that the MMIX had the largest code sizes uncompressed it now closes in on its competitors. The strangest part about this is how the OpenRISC takes the last place with the biggest sizes. This is remarkably strange since it should be quite equal to the SPARC in compressed size.
Even when comparing text segments inside the ELF objects the OpenRISC code demonstrate the same peculiar behaviour, ending up biggest and most compression unfriendly in general. The CRIS also shows a rather low compression score. This is probably due to the densely packed ISA, while MMIX shows enormous improvements in both

CPU/GCC Version	*Hypercube v.0.4*	*Boa v.0.94.13*	*Secure 0.34beta*
Axis CRIS/3.2.1	35736 bytes 0%	69552 bytes 0%	15724 bytes 0%
Knuth MMIX/3.2	(no system)	(no system)	31224 bytes +99%
OpenRISC 1200/3.1	50139 bytes +40%	97866 bytes +41%	20923 bytes +33%
SPARC Leon/3.2.2	47360 bytes +33%	89990 bytes +28%	17552 bytes +12%
x86/2.95.4	37248 bytes 4%	66654 bytes -4%	18496 bytes +18%
x86/3.0.4	35444 bytes -1%	(not tested)	(not tested)

Table 3.5: Text segment compared with CRIS

CPU/GCC Version	*Hypercube v.0.4*	*Boa v.0.94.13*	*Secure 0.34beta*
Axis CRIS/3.2.1	11111 bytes 31%	44009 bytes 63%	4616 bytes 29%
Knuth MMIX/3.2	(no system)	(no system)	4869 bytes 16%
OpenRISC 1200/3.1	13386 bytes 27%	52522 bytes 54%	7185 bytes 34%
SPARC Leon/3.2.2	10589 bytes 22%	49468 bytes 55%	4290 bytes 24%
x86/2.95.4	10888 bytes 29%	42973 bytes 65%	4262 bytes 23%
x86/3.0.4	10590 bytes 30%	(not tested)	(not tested)

Table 3.6: Percentual size of compressed text compared to the original

text only and full ELF-objects. On the whole, SPARC and OpenRISC generate approximately equal sized binaries ranging between 15-30 percent bigger uncompressed and approximately 5-30 percent bigger compressed binaries compared to the CRIS. The rather large range is created from the OpenRISC because the large compressed binaries it creates.

3.6 Available and free features to the core

Since the OpenRISC 1200 has a Wishbone interface, which is the standard interconnect interface supported by the OpenCores community, there is a plethora of devices that can be connected in a simple fashion to the OpenRISC core. On the OpenCores site there are over 30 projects that are wishbone compliant and useful in a SoC project.

- Ethernet MAC 10/100 controller (also used by the Leon for the PCI devboard).

- CAN protocol controller.
- SPI core.
- EPP v1.9 controller.
- I2C controller.
- IrDA protocol stack.
- USB 2.0 function core.
- USB 1.1 function core.
- USB 1.1 PHY core.
- UART 16550 core.
- PWM core.
- JTAG Test access port (TAP) core for OR1200.
- Bus switch fabric for SoC cores.
- PCI bridge core (also used by the Leon for the PCI devboard).
- OCIDEC IDE controller.
- Advanced memory controller for SDRAM and Flash.
- AC97 protocol controller.
- Simple VGA/LCD controller.

With this extensive list of available code to a SoC project, one could easily build quite a competent SoC indeed. For example Voxi AB, a voice recognition company based in Sweden, built their Voxic speech recognition device with the help of OpenCore modules and a couple of developed cores. The implementation was based on a FPGA development board but could easily be fitted in an ASIC product if full commercialisation should become a reality.

A California based company called ROSUM has created a GPS like tracking and positioning device that relies on RF signals from broadcast television systems. The device is based on the OpenRISC processor, a few more OpenRISC modules as well as a couple of their own modules much like the Voxi AB setup.

Since these small companies have not relied on any bigger CPU manufacturer for the components of the advanced devices they have, as Voxi AB put it, "*no last time buy*" on the processor or on it's components. They have no need for a processor manufacturer for their little embedded devices. All their needs are covered by the OpenCores community and the devices they develop there.

3.7 Licensing implications

Both the SPARC Leon and the OpenRISC 1200 are licensed with the GNU LGPL. The LGPL differs from the original GPL in many ways. First of all, the Lesser GPL is called lesser just because it is lesser in its protective specifications of the code compared to the GPL. Both the GPL and the LGPL are cunningly written. They are very specific but still diffuse enough to be hard to find loopholes through. As a lawyer it is probably a nightmare to have to fight a GPL license issue as well as it probably is a blessing to have it on your side.
The main difference between LGPL and GPL is that the former will tolerate that you link your own private code with it as long as you keep your modifications to the LGPL part of the package free and open. This would not be the case with GPL however. GPL requires that all code be compatible with the GPL if the package should be closed as an entity, as well as all the source available to all third partys.
So for a company like Axis the LGPL is a welcome licence for the cores. It should present no problems at all as long as all code that once was LGPL remains LGPL. This would mean that any modifications made to the code would be redistributed back to the copyright holder and that the company is very clear about it's copyright policy when dealing with the code.
Furthermore, all the code that is developed for own use will be protected as the company's own property as long as it doesn't interfere with the LGPL code.

3.8 The future

The OpenRISC architecture has a lot of potential and should be considered a very viable option for future designs. Perhaps it's not as much the architecture as the OpenCores community itself. Actually, the needs of such powerful communities as a complement to commercial alternatives are great. Much like Linux have contributed to the world with a powerful OS with great flexibility and openness, OpenCores could do the same when it gains enough momentum. Then the world would see a revolution in how bigger companies deal with their intellectual properties and how they develop new IP. IP is not a bad thing, on the contrary, it pushes development and rewards the inventors. But when an unhealthy monopoly or a monopoly consortium takes over the entire marketplace, then the customer will suffer and, thus paying for overpriced products. This happens from time to time and after a while new competitors pop up to create a equilibrium in the market, to the profit of the customer, development and everyone in general.
For example, Intel had their share of monopoly for a long long time. Then AMD entered the PC processor market and started to take piece by piece from Intel. On the other hand, Intel still dominates the PC processor market, but the healthier equilibrium between two competitors have developed.
Microsoft are in a very similar situation today. They have dominated the market for desktop OS and are slowly chewing themselves into the server market. With no natural competitors, Microsoft was bound to grow like they did, becoming one of the wealthiest companies in the world. Every competitor in their way was either bought up, slowly phased out or sued. Then came a competitor that Microsoft could not buy, phase out or sue. GNU/Linux creates a healthy marketplace situation in a monopoly much like AMD is doing for the PC processor market. Likewise in this situation, Microsoft still dominates the market, but not without worries or competitors.

Perhaps it's time to see this revolution come to the EDA and IC design industry. A overpriced market with many competitors that seems to live in a mutual understanding of almost extortion like pricing on the products. There is a need for a free community that can develop hardware for free.
So no matter what the OpenCores community is up to, be it processors or other equipment, it is only a good thing for the market in general.

Chapter 4

Implementation

4.1 Original implementation

In the first, and as of today, the only implementation of the OpenRISC 1000 architecture, achieving clock speed wasn't the primary goal. Instead, the main focus was on stability and getting a working core up and running. To be able to analyse the implementation, a schematic layout[1] of the entire CPU was drawn with the Verilog code as reference. The source code is well structured and segmented depending on which task the module has. I.e arithmetic logical unit is a module named or1200_alu.v, instruction fetch unit is named or1200_if.v etc. This structure seems nice and every module has it's specific task. However, many signals in the CPU has a long combinatorial behaviour and the impact of this will be shown later.

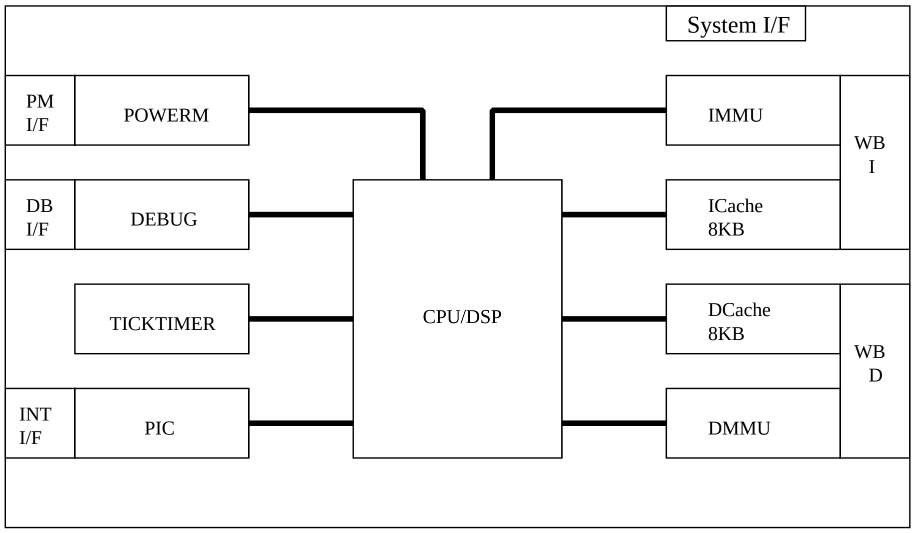

Figure 4.1: Overview of the OR1200

The design is relatively complete and it have been implemented on FPGA's running

[1]The schematics only exist in digital form due to the sheer physical size.

uClinux. An ASIC design of the CPU has not been done yet, but it's just a question of time. The only things that are missing in the package are of course the memories. Memories could be synthesised as flip-flop types but that would result in a large and slow design. You are much better of synthesising with memory implementations from your ASIC or FPGA library vendor. This is left to each user of the OR1200 to find and implement their own memories of choice.
The OR1200 implementation is very configurable. Some of the options you can tweak are:

- Choose ASIC/FPGA target.
- Change cache sizes.
- Change number of TLB-entries.
- Synthesise with/without caches, synthesise with/without MMU's.
- Choose ASIC or generic multiplier.
- Turn on/off debug unit.
- Turn on/off tick timer unit.
- Turn on/off power management unit.
- Define special instructions.

The core deals with the configurations in a series of definitions located in the main configuration file, the or1200_defines.v. This file has everything needed to configure the core. If anything else should be needed, such as special execution units, they must be added in the pipeline path as usual.

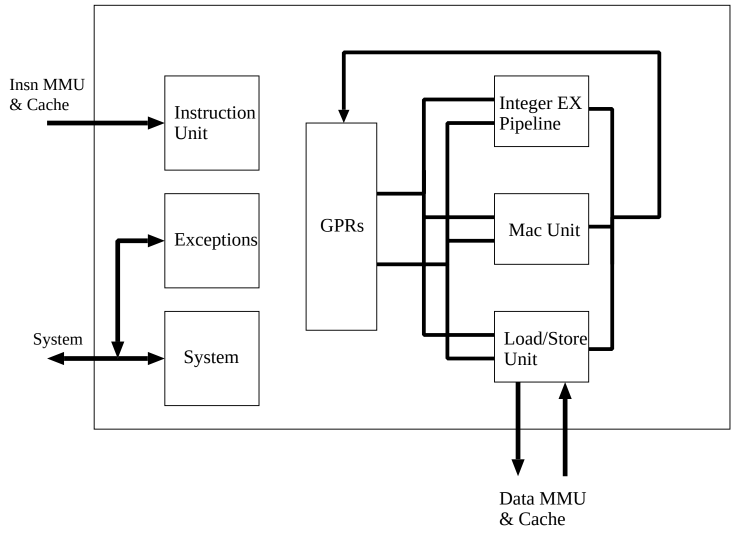

Figure 4.2: CPU/DSP Block Diagram

4.1.1 Features of OR1200

On OpenRISC home-page there is a flyer named ’OpenRISC 1200 IP Core Overview’ which describes the functionality of OR1200. The flyer describes the core as complete and fully functional with good performance.

Features of OR1200 according to the flyer:
High Performance 32-bit CPU/DSP

- 32-bit architecture implementing the ORBIS32 instruction set.
- Scalar, single-issue 5-stage pipeline delivering sustained throughput.
- Single cycle instruction execute on most instructions.
- 250 MIPS performance @ 250MHz worst case conditions.
- Predictable execution rate for hard real-time applications.
- Fast deterministic internal interrupt response.
- Thirty-two, 32-bit general purpose registers.
- DSP MAC 32x32.
- Custom user instructions.

L1 Caches

- Harvard model with split data and instruction cache.
- Instruction/data cache size scalable from 1KB to 64 KB.
- Physically tagged and addressed.
- Cache management special purpose registers.

Memory Management Unit

- Harvard model with split data and instruction MMU.
- Instruction/data TLB size scalable from 16 to 256 entries.
- Direct mapped hash-based TLB.
- Linear address space with 32-bit virtual address and physical address from 24 to 32 bits.
- Page size 8KB with per-page attributes.

Sophisticated Power Management Unit

- Power reduction from 2x to 100x.
- Software controlled clock frequency in slow and idle modes.
- Interrupted wake-up in doze and sleep modes.
- Dynamic clock gating for individual units.

Advanced Debug Unit

- Conventional target-debug agent with a debug exception handler.
- Non-intrusive debug/trace for both RISC and system.
- Real time trace of RISC and system.
- Access and control of debug unit from RISC or via development interface.
- Complex chained watchpoint and breakpoint conditions.

Integrated Tick Timer

- Task scheduling and precise time measuring.
- Maximum timer range of 2^{32} clock cycles.
- Maskable tick-timer interrupt.
- Single-run, restartable or continuous mode.

Programmable Interrupt Controller

- 2 non-maskable interrupt sources.
- 30 maskable interrupt sources.
- Two interrupt priorities.

Custom and Optional Units

- Additionally units such as a floating-point unit can be added as standard units.
- 8 custom units can be added and controlled through special-purpose registers or custom instructions.

Development Tools Support

- GNU ANSI C, C++, Java and Fortran compilers.
- GNU debugger, linker, assembler and utilities.
- Architectural simulator.

Operating System Support

- Linux.
- uClinux.
- OAR RTEMS real-time OS.
- Leading 3rd party products such as Windows CE and VxWorks are planned to be available.

System Interface

- System interface optimised for SoC applications.
- Low-latency, open-standard dual WISHBONE interfaces.
- Dual interface — simultaneous flow of instruction and data.
- Variety of peripheral cores optimised for transparent interconnection with the OpenRISC 1200.

4.2 Timing analysis

When synthesised with a 0.18μ library the reached clock speed was 150MHz. This synthesis was done with most parts in the processor enabled except memories, which were synthesised as black boxes. The synthesise showed a big difference with regards to the OpenRISC team claims. One explanation for some of the difference in performance could depend on the choice of manufacture technology. When synthesised with different libraries the difference between the fastest and the slowest library were as much as 50%. But the difference was to big to be explained by libraries alone. Probably the performance claimed by the team was taken out of the blue, which has also been confirmed by the OpenRISC team over an email conversation.

Synthesising the memories as black boxes shouldn't be a problem at these low clock frequencies. Standard cell memories of today have equal or better performance. However, the demands on the memories get higher with raised frequency. With clock speeds over 300MHz it's hard to find memories that matches with that type of performance.

To be able to reach 400 MHz a 0.13μ library was used. With the same configuration as with 0.18μ library the maximum clock speed were 238MHz. In further testing power management, PIC unit, tick timer and debug unit were disabled. The interest is if the CPU with MMU's could do 400 MHz with the current implementation. At this early time the debug feature and the power management doesn't play a essential role and so we decided to work on the CPU-core alone by disabling the external features.

When doing timing based analysis you have to consider the output of the synthesis tool that you are using. Most modern EDA[2] tools use timing constraints in the inner optimisation loop to try to meet demands placed on the synthesis by the user. Simply put, if you place high frequency demands on your core, the tool will do it's best to match the performance, given certain criteria. By constantly checking endpoint to endpoint timing on all paths in the core the tool can come up with a list of the longest timing paths and start to optimise from these and downwards. Since the logic that you are realizing can not be faster than the slowest path, this type of analysis makes perfect sense.
Depending on the effort level and a few other parameters, the tool will either meet your demands or give up sooner or later. When you can't meet demands by tool usage alone it is up to the user to tweak the paths logically and functionally to receive a better synthesis result.
The problem of doing timing based analysis is that the result from each run can vary quite a lot and it is not always meaningful to ask the program of all critical paths at the same time. This way you would surely receive a very long list of almost equal paths that can be quite difficult to sort through. We chose to deal with one path at a time during the entire project.

[2]Electronic Design Automation tools. Such as Cadence BuildGates, Synopsys DesignCompiler etc.

4.3 Critical paths

4.3.1 Path of spr_pc_we

In the first synthesis run, a critical path starting in the debug unit and ending in ctrl alias the instruction decode, was found. After a bit of research this path was taken care of by disabling the signal *spr_pc_we* in module *genpc*. By removing the signal in *genpc* this will result in a malfunction in the debug unit which will not be able to load up the program counter with new values. This doesn't affect us for the time being since we have disabled all debugging anyway. But for future units this could be a major problem.
The performance increased from 238MHz to 242MHz. This is barely a noticeable difference, but a new long path was revealed.

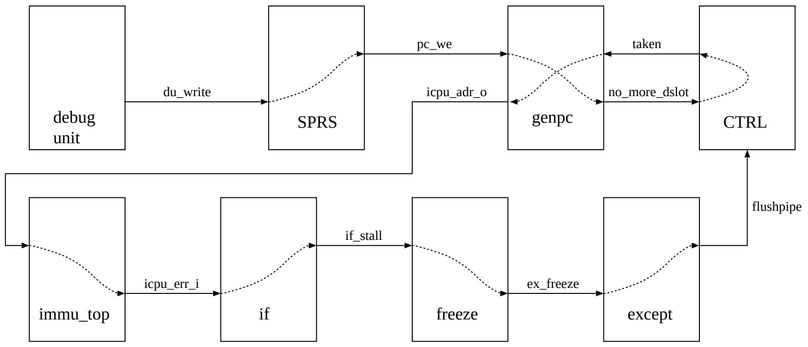

Figure 4.3: The first long path, which was disabled.

4.3.2 Path of binsn_addr

After disabling the signal *spr_pc_we* the next critical path that occurred was through *binsn_addr* in module *genpc*. This signal begins in module *except* and is used when calculating the program counter value.

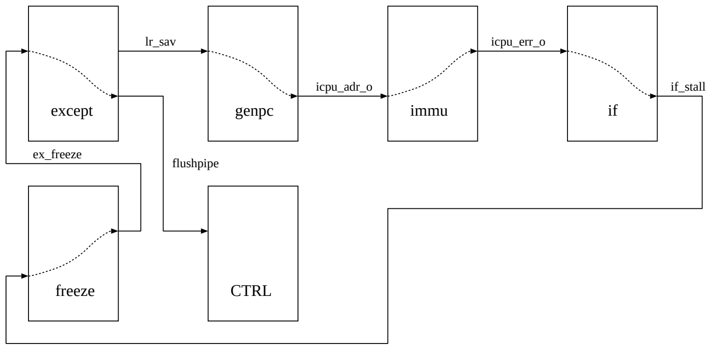

Figure 4.4: Path through *binsn_addr*.

The biggest timing problem in this path is a 30–bits adder that costs about 0.78ns.

To solve this timing problem we decided that we should move the 30-bit adder to the previous cycle where timing wasn't as critical. The adder is used to add the two signals *binsn_addr* and *branch_addrofs* which is the branch offset to give the new program counter value.

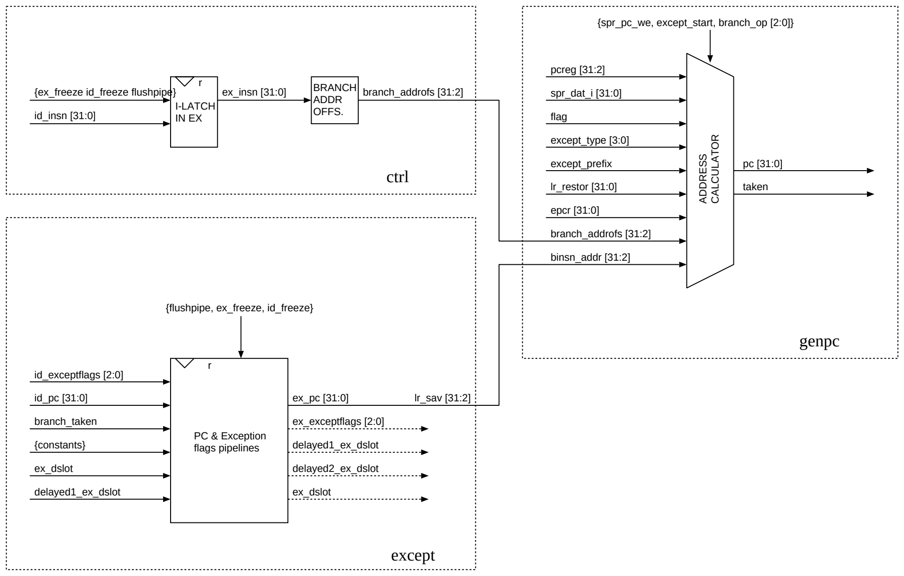

Figure 4.5: Original implementation of address calculation.

The adder is located inside the address calculation mux in module *genpc*, see figure 4.3.2. To be able to move the addition and still have correct functionality the signal *branch_addrofs* also had to be moved to the previous clock cycle.

4.3.3 Disabled MMU's path

With this change the functionality of the CPU remains the same but the performance has been improved and now it became possible to run the CPU with a clock frequency of 276 MHz. Unfortunately the next critical path, see figure 4.3.3 that we discovered was not very easy to shorten or split. This path goes from module *genpc* through Instruction Cache, MMU and on to instruction fetch module. The bottleneck seemed to be in module *genpc* and in module *immu_top*. When MMU's were disabled the clock speed improved to 340MHz.

4.3.4 Broken design path

After this we tested to move the latch for next address calculation in module *immu_top*, see figure 4.3.4. By moving this latch the clock speed improved to 367 MHz. But doing so the CPU stopped working. The test however, showed what performance the CPU with a different design on genpc and MMU's could do. With this configuration the longest path went through the data MMU.

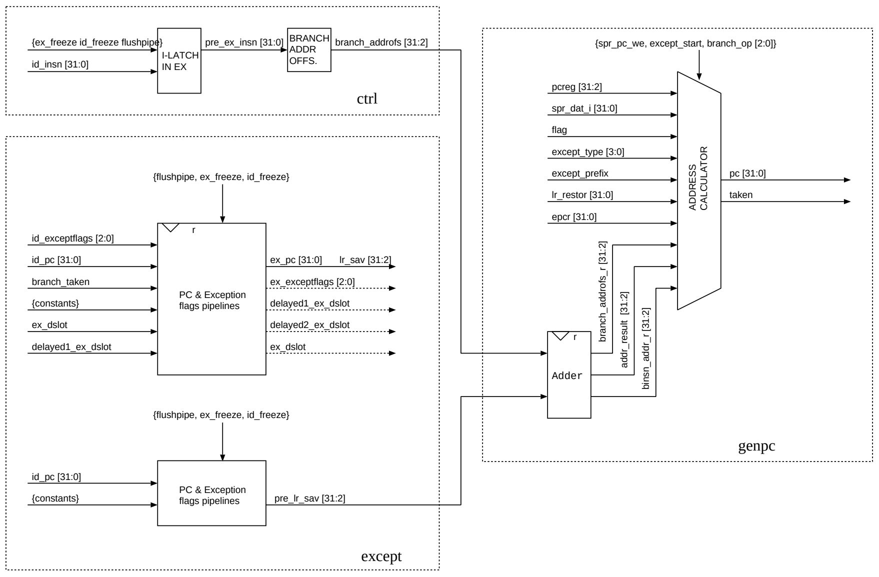

Figure 4.6: Modified implementation of address calculation.

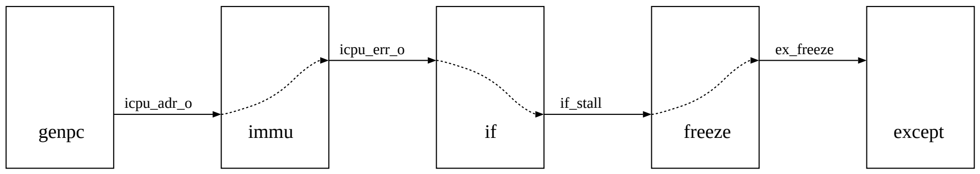

Figure 4.7: Path through *genpc*

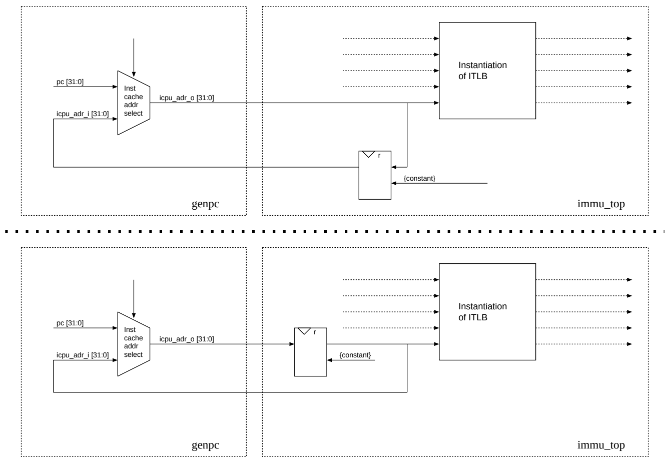

Figure 4.8: Test fix. Moving the latch gives the IMMU one clock cycle more

4.3.5 Summary of tests

This is a quick summary of the relevant tests and the speed results that we got. The most obvious part is when the MMU gets disabled the speed rises quite a bit. This could be regarded as evidence of a badly or very tightly designed MMU / cache back-end.
When the memories are introduced in the synthesis, it shows that the logic around the memories eat to much time of the cycles that should be available to memory lookups. This is a must to do something about.

Path	Module	Worst Case Timing	Redesign	Note
Original implementation	Original implementation	238MHz	none	low performance no memories
spr_pc_we	genpc	242MHz	Removed signal	Debug read/write disabled no memories
binsn_addr	genpc	276MHz	Moved adder	no memories
icpu_adr_i	immu_top	340MHz	disabled MMU's	no MMU's no memories
icpu_adr_i	immu_top	367MHz	without memories, moved latch	non functional CPU, no memories
icpu_adr_i	immu_top	200MHz	none	with memories & MMU's

Table 4.1: Redesign and implications

4.4 False paths

One problem in timing analysis is asynchronous behaviour. It is very hard to verify a asynchronous signal because of it's very nature. In the design there are some path that are closely related to being asynchronous. In some sense, they are asynchronous in contrast to the desired behaviour, but are still synchronous signals. This type of signals are often long false-path combinatorial signals.

False paths are non-critical paths in the design that can be ignored when performing timing analysis on the design. The first path, pc_we, runs through the CPU without being registered once. This signal controls several muxes in different pipeline stages. In the timing analysis this signal showed up as a critical path, but a investigation of the path showed that the timing probably aren't critical. False paths is a problem in a design because it can be hard to see if it actually is a false path or not. The difficulty in analysis of the signal leads to problems when trying to correct the behaviour. If you can't see what's correct in a signal path, then you will have a hard time splitting it into several paths, ignoring it completely or doing some other type of fixes to it.

4.5 Implication of the MMU units design

With the current design the MMU's are clearly the bottleneck. The functionality of the MMU's are right but the performance is low. When synthesised with memories the performance got even lower. One of the problems in the design is the hit/miss calculation. First the address is calculated in genpc, then the calculated address are compared against the previous virtual address to see if the address is in the same page. If this is a match the physical address is sent to the cache, the instruction gets fetched and all is fine. This is done in a zero latency manner. The CPU gets halted only when a translation is needed to do a lookup and eventually a cache-line fetch.

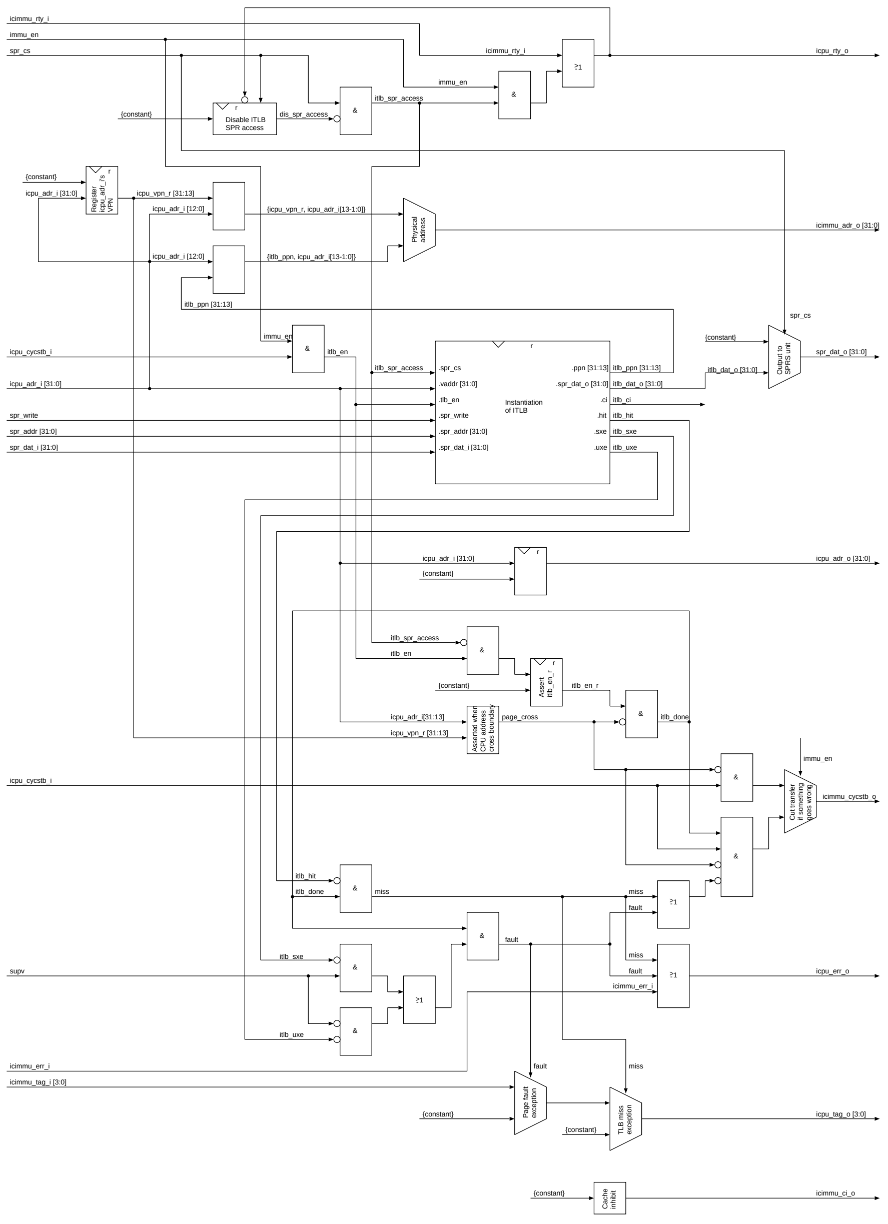

Figure 4.9: OR1200 IMMU module.

4.6 Program counter generation problem

Another problem with the implementation is the *genpc* module. Every long path occurred through this module. Generating the program counter in it self shouldn't be any problem but with the control signals from except, freeze and ctrl the timing gets critical. In the design its hard to follow the signal flow in the CPU and the entire module is designed with all sorts of combinatorial paths through it. But the main problem in the PC generation lay in the connection against the MMU. With a unclocked output address from module *genpc*, which should be clocked for the next cycle, the timing inside the MMU's become very critical.

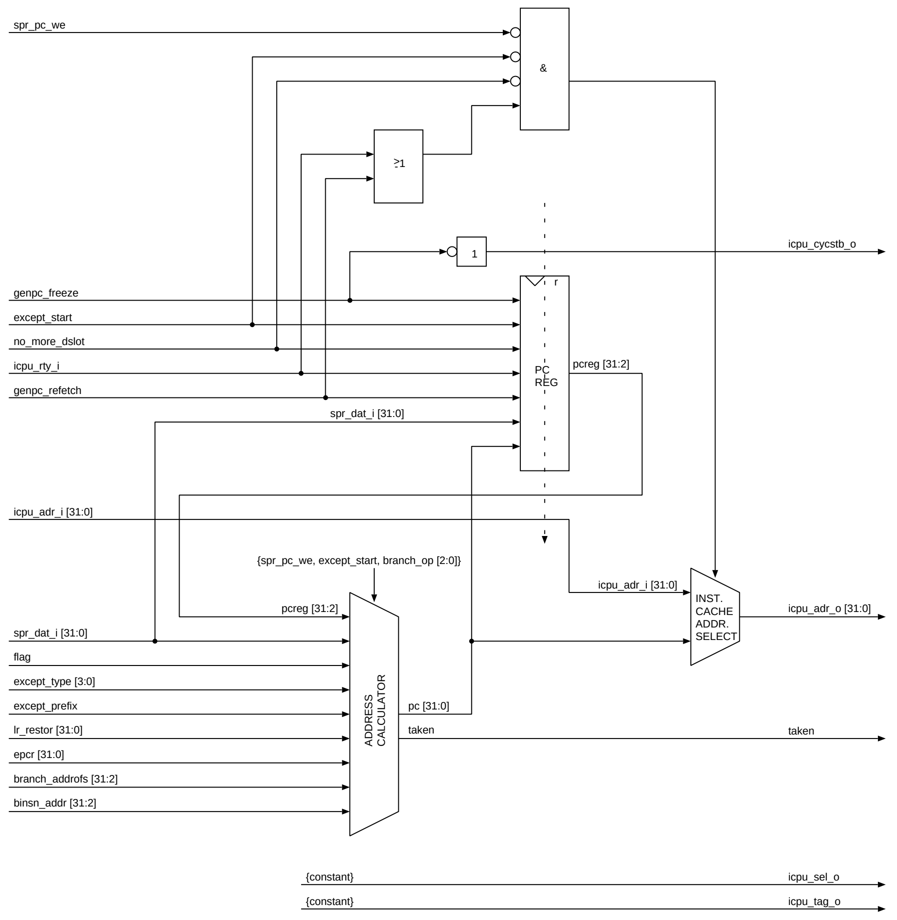

Figure 4.10: OR1200 genpc module. Obvious combinatorial paths.

4.7 Cache memory block simulation

Since we didn't have access to memory blocks of certain sizes to use for our CPU, we had to write our own memories to simulate timing delays within a memory. The memories where written with a timing library format file syntax used by Ambit Build

Gates[3].
The simulated memories where written with a timing delay on 2.5ns and a setup time of 1.0ns. This is a fairly long time but standard cell memories of today doesn't perform much better. Because of the low memory performance it's important that the memory has a clock cycle on its own without being encapsulated with to much logic. In the OR1200 implementation there is a lot of logic in the same clock cycle, for example address translation. This meaning that except the time in the memory also the time to do a address translation has to be done in one clock cycle.
A synthesise run with memories proved the bad design choices and the performance dropped hard to a worst case clock frequency of 200 MHz which put us right back at step one.

```
Cell(timing_spram_1024x32
PIN(clk
PINTYPE(input)
CLOCK_PIN
CAPACITANCE(0.5)
)

BUS(addr[9:0]
BUSTYPE(input)
CAPACITANCE(0.5)
)
BUS(di[31:0]
BUSTYPE(input)
CAPACITANCE(0.5)
)
BUS(do[31:0]
BUSTYPE(output)
CAPACITANCE(0.5)
)
PIN(we
PINTYPE(input)
CAPACITANCE(0.5)
)
PATH(clk *>do[31:0] 01 01
Delay((Const (2.5)))
Slew((Const(0.3))))
PATH(clk *>do[31:0] 01 10
Delay((Const (2.5)))
Slew((Const(0.3))))

SETUP(addr[9:0] *>clk 01 Posedge (Const(1.0)))
SETUP(addr[9:0] *>clk 10 Posedge (Const(1.0)))
SETUP(di[31:0] *>clk 01 Posedge (Const(0.5)))
SETUP(di[31:0] *>clk 10 Posedge (Const(0.5)))
SETUP(we *>clk 01 Posedge (Const(0.5)))
SETUP(we *>clk 10 Posedge (Const(0.5)))

HOLD(addr[9:0] *>clk 01 Posedge (Const(0.5)))
HOLD(addr[9:0] *>clk 10 Posedge (Const(0.5)))
HOLD(di[31:0] *>clk 01 Posedge (Const(0.5)))
HOLD(di[31:0] *>clk 10 Posedge (Const(0.5)))
HOLD(we *>clk 01 Posedge (Const(0.5)))
HOLD(we *>clk 10 Posedge (Const(0.5)))
```

Table 4.2: Excerpt of a timing library format file showing a 32KB memory

[3]Now owned by Cadence, Build Gates is one of the market leading ASIC synthesis tools to date

4.8 Timing results

The original implementation of OR1200 surely isn't the best implementation if speed is important. The major problem lies in the implementation of the MMU and the program counter generation. In the default configuration a clock speed of 150MHz was achieved using 0.18 μ technology which later was confirmed by the OpenRISC team with regards to their high frequency figures in the flyer and the implementation manual.

When we choose to use a 0.13 μ technology the performance went up to 200 MHz with simulated memories in the end. The performance of the CPU should be better than this but the implementation is unfortunately not god enough. To be able to reach 400 MHz a redesign of MMU and program counter is essential. Because of the low memory performance it's important that a new design has pipelined MMU's and caches. A good parallel lookup implementation, hit decision and memory lookup should be done in more than one cycle. The drawback of this will be that the penalty for cache access will be one clock cycle more. This is a trade-off likely done because the gain in frequency which will make the performance of the CPU increase anyway.
The other alternative, instead of redesign MMU's and genpc, is to implement the CPU from scratch. This is probably a better solution and could be less time consuming. The current implementation contains too much complexity and to redesign MMU's and genpc will cause more changes through out the CPU. These changes won't be easy. 400MHz is hard to reach but could be expected with the architecture of OR1000 implemented with a five stage pipeline. One of the problem will be finding standard cell memories that can match the performance needed at 400MHz with only one stage for memory access. A more normal figure should be about 300MHz with good enough memories.
When designing a CPU all pipeline stages should be clearly separated from each other. An example could be that all modules in the design should have registered outputs. This will make it easier to find the slow parts of the design and to redesign parts of the CPU. OR1200 has a designed with different modules but doesn't benefit from this because the outputs from the modules aren't registered. This will make it hard to redesign certain modules without redesigning a big part of the CPU. If registered output is used the size of the core will presumably grow and its a trade-off that has to be done if the design should be easy to analyse. Of course some signals will be hard to register but the effort should be to minimise these.

Chapter 5

Conclusions

When designing a new SoC processor there are several things to keep in mind. First off, you need a high quality and simple ISA to last longer than the actual implementation of the current generation. This has proved itself over and over again. If the architecture gains foothold, it's probably going to last longer than you think. The simplicity of a ISA becomes a virtue in the long run. Of course a complex ISA can be kept alive for a long time, but that costs tremendous amounts of resources. When those resources can't be mobilised in a small company the choice of implementation becomes easy. Keep it simple.

Second off, you need to have a very good idea of what you are targeting in technical implementation. What type of cache / MMU configuration do you build? This is inherently important between generations of implementations. No CPU run particularly well without a well designed and well balanced memory access hierarchy. You need to know the balance between stage implementation and stage complexity. The tradeoff between complexity and speed gains should be remembered in contrast to Amdahl's Law of diminishing returns. The more effort and flashy solutions you put in your implementation the more likely you are to end up with a CPU that is way to big (physically) for your demands. It's very easy to miss a targeted size if you are not careful.
The actual technical modelling could be very simplified by good high level implementation modelling. For example, programs like SIMICS could help much if used properly. Caches and MMU's can be configured and reconfigured on the fly with new testing runs completed within minutes of a reconfiguration.
If Axis decides that a new ISA should be implemented after all. Then a extremely simple classical RISC ISA would be the best choice. No SIMD and certainly no VLIW. Although both technologies are very tempting to use and worth their effort for some type of CPUs the embedded segment is just not there yet. SIMD and VLIW could however be used in special co-processor units however without any problems.
The classical RISC has several advantages. It's easy to maintain and easy to scale. There is plenty left for speed in the classical RISC design for embedded purposes for at least another decade to come. So it should be a natural choice for new embedded processors.

Third and finally, you need a very good understanding of what your code becomes in terms of hardware. By looking at the OR1200 implementation we have found sev-

eral deficiencies in the implementation leading to general asynchronous behaviour and long false paths.
If you don't know how to design a proper pipeline in the form of coding experience, you shouldn't expect extreme speeds either. Simply because it's probably very hard to verify the CPU functionality if the design proves to have asynchronous elements within.

Chapter 6

Summary

There is no doubt that the need of performance will increase in the future. During our thesis we have found that the following is valid on Axis part.
A standard RISC ISA processor with good design is enough to carry need for processor power for a long time ahead. This applies to both the SPARC Leon processor and the OpenRISC 1200 processor. Both are valid choices.
When compared to the Leon, OpenRISC 1200 might seem like an unwise choice to make, in these days when short term investments and short termed projects are primary goals for an entire industry. We did however chose the OpenRISC 1200. Because we firmly believe in a partnership between OpenCores and Axis is for mutual benefit. This community has however a long way to go but with a very promising future if things turn out well.

Unfortunately, we could not meet the performance requirements due to lack of time and due to the complexity of redesigning an entire implementation to match the need for speed at 400MHz. So we settled for a performance tuning and estimation of architectural fitness for Axis next generation.
We have however learned a lot during our thesis here at Axis. From architectural knowledge to tinkering with tools that normally aren't available to the public, it has been a great pleasure with a good deal of failures as well as a good deal of successes. This has enabled us to become more broad in our knowledge concerning ASIC technology and semiconductor manufacturing in general. Failures have ranged from simple scripting mistakes to complex unsolvable problems in the given time-frame. Successes has mostly come in the form of knowledge on how the coding style affects your final layout and how coding should be done from the first place to enable speedy designs.

Appendix A

Pictures

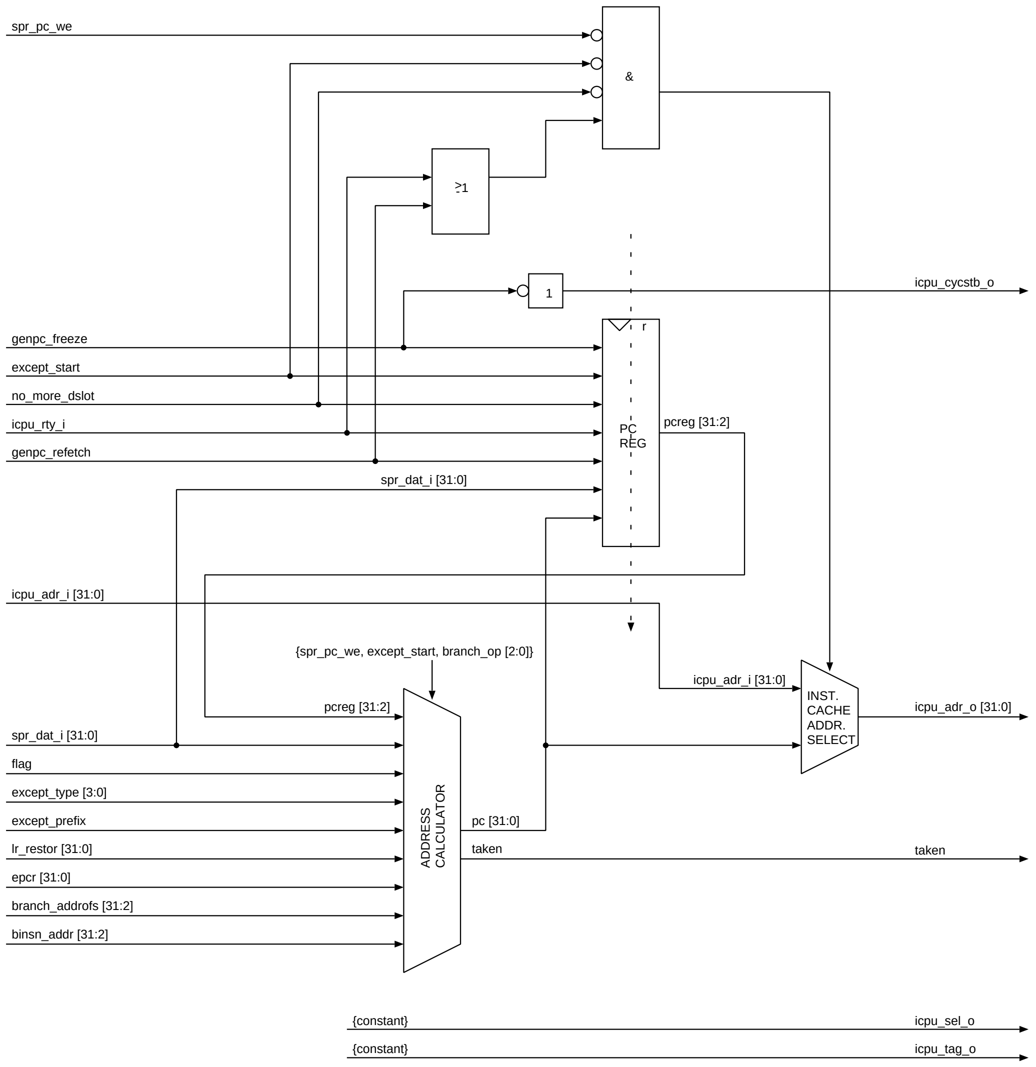

Figure A.1: OR1200 PC Generation.

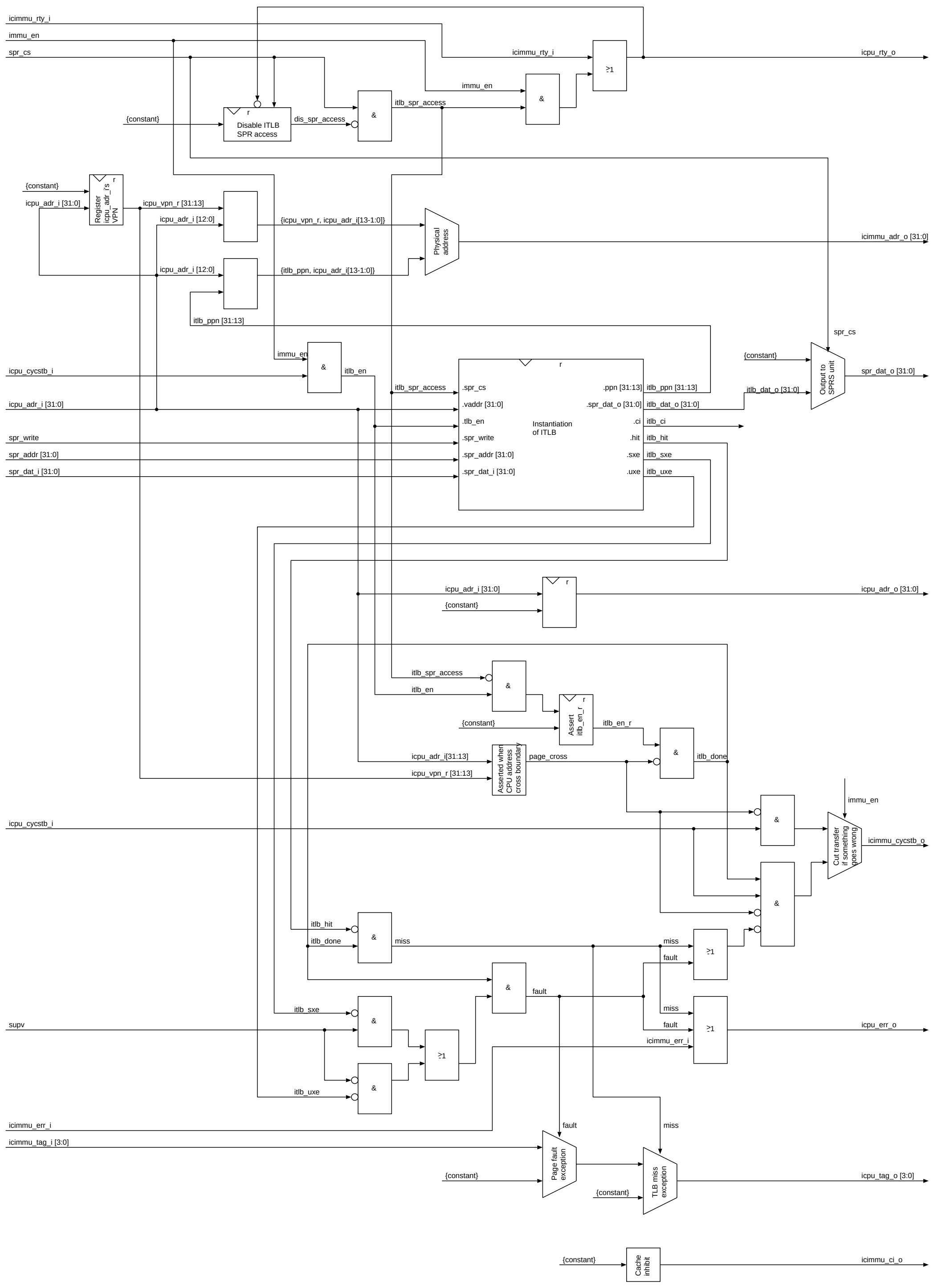

Figure A.2: OR1200 IMMU.

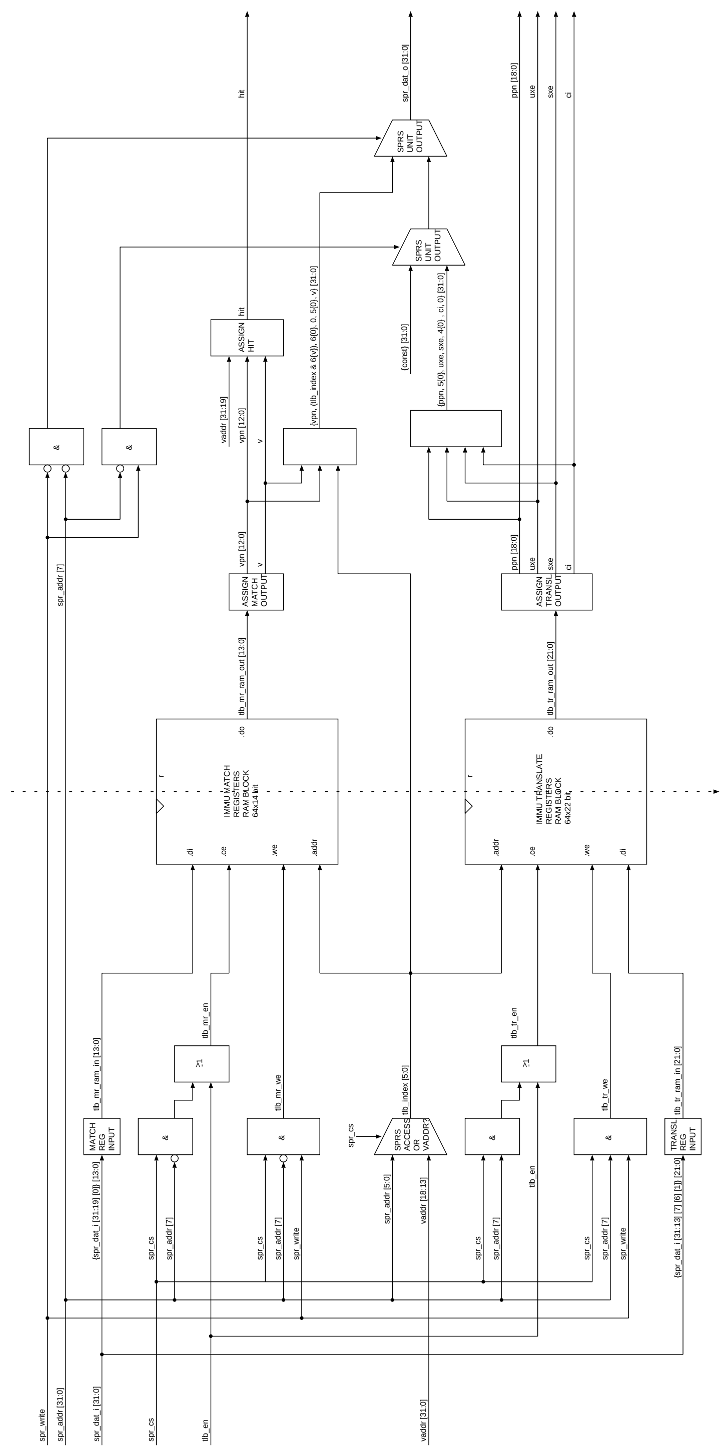

Figure A.3: OR1200 Instruction TLB.

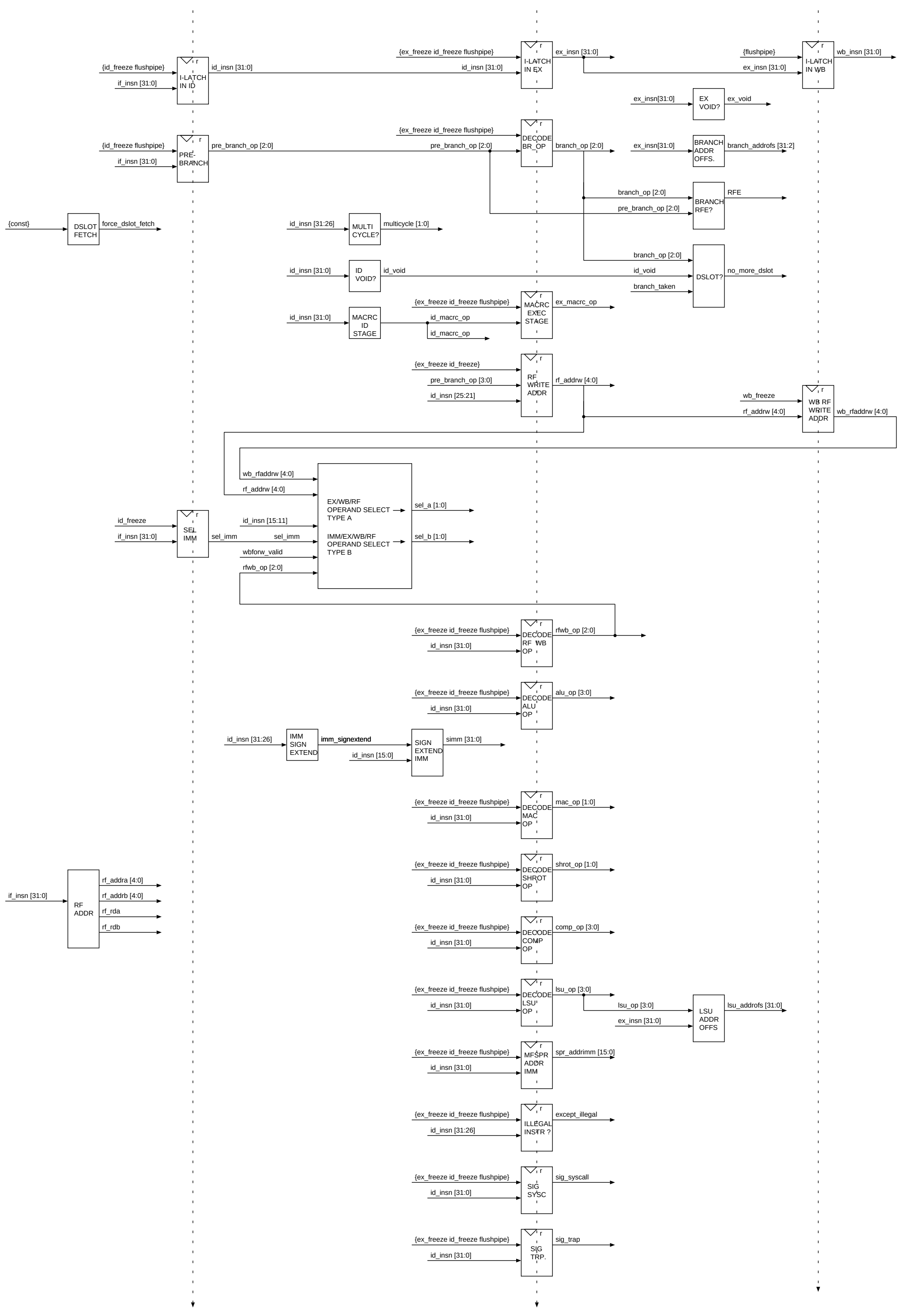

Figure A.4: OR1200 Instruction Decode.

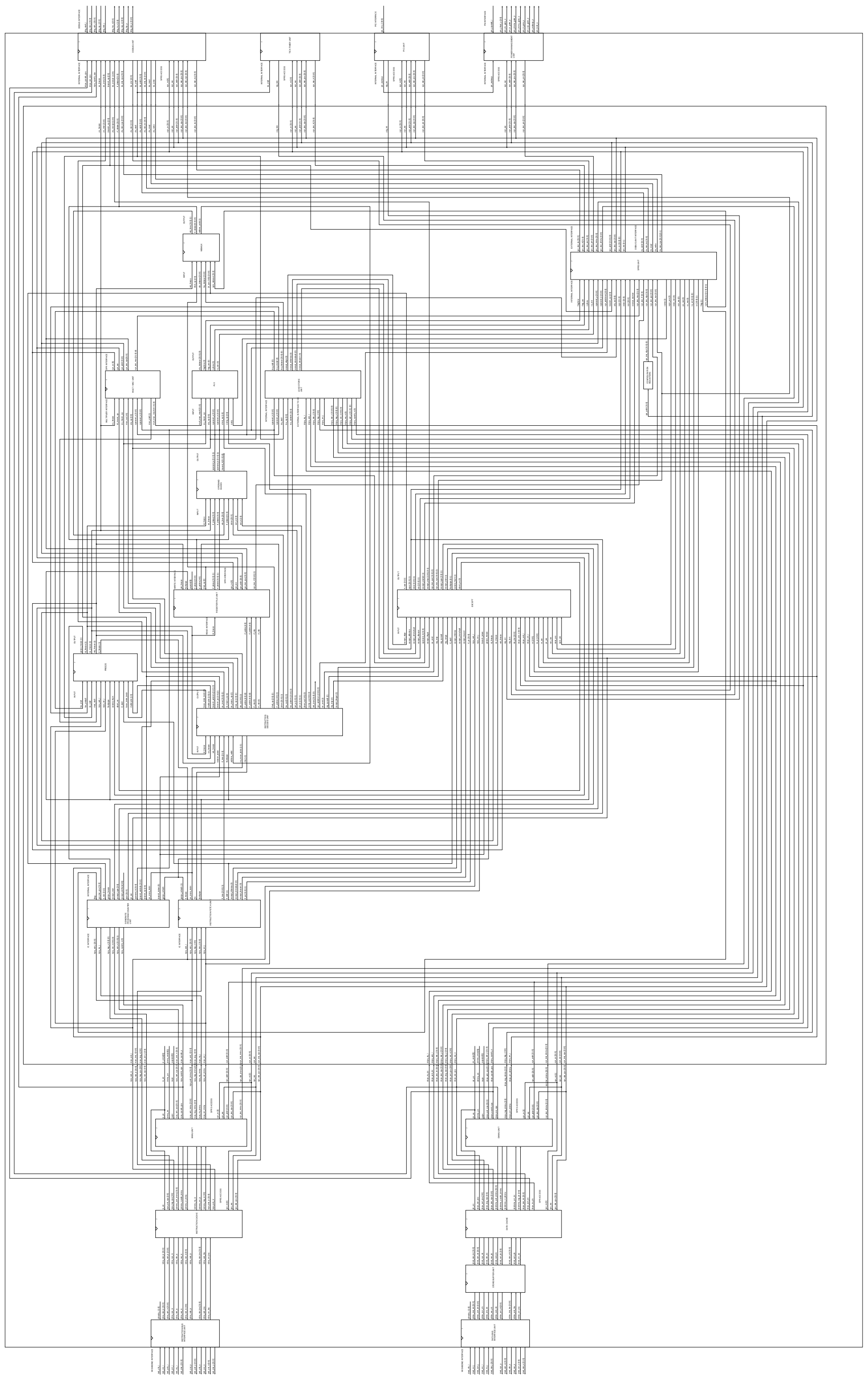

Figure A.5: OR1200 CPU overview.

Bibliography

[1] *Comprehensive Verilog Issue 5.0.* Doulos Ltd.

[2] *Comprehensive Verilog Workbook Issue 5.0.* Doulos Ltd.

[3] *The Verilog Golden Reference Guide Version 1.1.* Doulos Ltd.

[4] David A. Patterson & John L. Hennessy. *Computer Architecture. A Quantitative Approach, Second Edition.* Morgan Kaufmann Publishers, Inc 1996.

[5] David A. Patterson & John L. Hennessy. *Computer Architecture. A Quantitative Approach, Third Edition.* Morgan Kaufmann Publishers, Inc 2002.

[6] Sites, Witek. *Alpha AXP Architecture Reference Manual, Second Edition.* Digital Press 1995.

[7] *ARCompact Technical Backgrounder.* ARC International.

[8] *ARCtangent-A5 Processor, Product Brief.* ARC International.

[9] Dave Jaggar. *Advanced RISC Machines Architectural Reference Manual.* Prentice Hall PTR.

[10] *CRIS Programmers Manual.* Axis Communications AB, 21 Jan., 2003

[11] Gerry Kane & Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall PTR.

[12] Donald E. Knuth. *MMIX.* 15 Jun., 2002.

[13] *NEC VR4121 64-bit MIPS Processor, Product Brief.* NEC Electronics Inc.

[14] *OpenRISC 1000 Architecture Manual.* OpenCores.org 12 Jan., 2003.

[15] *OpenRISC 1200 IP Core Specification Revision 0.7 Preliminary Draft.* 6 Sept., 2001

[16] *The SPARC Architecture Manual Version 8 Revision SAV080SI9308.* Sparc International, Inc.

[17] Jiri Gaisler. *The Leon-2 Processor User Manual Version 1.0.10.* Gaisler Research, 2003.

[18] *The SPARC Architecture Manual Version 9 Revision SAV09R1429309.* Prentice Hall PTR.

[19] *SH4 SuperH RISC Processor, Product Brief.* Hitachi Europe, Ltd.

[20] Tom Shanley. *PowerPC System Architecture.* Addison-Wesley Publishing Company.

[21] Tom Shanley. *Pentium Pro and Pentium II system architecture, Second Edition.* MindShare, Inc Dec, 1997.

[22] *Xtensa Architecture And Performance.* Tensilica, Inc. Sept., 2002.

[23] *AMBA Advanced Micro-controller Bus Architecture Revision D.* Advanced RISC Machines Ltd, 1997.

[24] *AMBA Specification Version 2.0.* Advanced RISC Machines Ltd, 1999.

[25] *The Core Connect Bus Architecture.* International Business Machines, 1999.

[26] *WISHBONE System-On-Chip (SOC) Interconnection Architectures for Portable IP Cores Version B.3.* OpenCores.org 7 Sept., 2002.

[27] Rudolf Usselmann. *OpenCores SoC Bus Review Revision 1.0.* OpenCores.org 9 Jan., 2001.

[28] *CodePack: Code Compression For PowerPC Processors Version 1.0.* International Business Machines, 1999.

[29] Larry Wall, Tom Christiansen & Randal L. Schwartz. *Programming Perl, Second Edition.* O'Reilly & Associates, Inc.

[30] *BuildGates Users Guide Release 2.3* Cadence Design Systems, Inc.

[31] *BuildGates Command Reference Release 2.3* Cadence Design Systems, Inc.

[32] *The GNU Lesser General Public License.* http://www.gnu.org/licenses/lgpl.txt