KTH Computer Science
and Communication

Open Source Hardware Development and the OpenRISC Project

A Review of the OpenRISC Architecture and Implementations

JULIUS BAXTER

Master's Thesis at IMIT
Supervisor: Ahmed Hemani
Examiner: Ahmed Hemani

Abstract

Advances in the design and manufacture of microelectronic devices since the 1960s have enabled embedded computers that are ubiquitous. Microprocessors, the core component in modern computers, and their architectures have evolved continuously over this time, too. During the 1980s a new architectural approach, favoring a reduction in design complexity, emerged and became known as reduced instruction set computer, or RISC, architectures. The mid-1980s also saw the beginning of a widespread change in the attitudes towards computer software. The Free Software Foundation (FSF) was set up and aimed to foster the development of free (as in *free*dom) and open source software, as a reaction to increasingly protective measures software vendors were taking to restrict the use of their software. The concept of less restrictive software has proved successful but it took over fifteen years before this philosophy was applied to the discipline of electronic hardware design. One of the earliest and most prominent projects to do so was initiated in the late 1990s by a group of students aiming to develop an open source microprocessor architecture and set of implementations. Their goals were realised in the OpenRISC project, a RISC microprocessor specification and implementations. The initial development team then created OpenCores, an online open source hardware design community focusing on developing register transfer level (RTL) designs of functional cores based on the principles of the open source software movement. The application of open source principles to hardware design gathered pace throughout the decade that followed, but despite good progress early in the OpenRISC project, it slowed as the maintainers decided to pursue commercial interests and ceased development of the publicly released versions. Recent interest in the architecture, and increased uptake in use of open source hardware, has lead to a much-needed rejuvenation of the project. Twelve years on from the inception of the OpenRISC project, this work has lead to questions about the state of the open source hardware development movement, and about a possible successor to the first OpenRISC architecture. This document will discuss the underlying technologies and philosophies of the OpenRISC project, present the recent work on the platform, undertake a critical analysis of the project as a whole, and present a section on the future directions of the OpenRISC 1000 project in particular, and open source hardware development in general. Recommendations of specific work to be done on the project and arguments for the general direction of development are presented. Finally, the proposed successor architecture, OpenRISC 2000, is discussed.

Acknowledgements

I would first like to thank my parents for their support and providing me with the opportunities that I have had. I would like to acknowledge ORSoC for their commitment to open source technology and allowing me to do the work that I have done. I would also like to acknowledge the small but committed OpenRISC development community, without whom not nearly as much would have been achieved during the last three years. I would finally like to acknowledge the patience of my supervisor, Ahmed Hemani, who has been very helpful during my studies at KTH.

Contents

Chapter 1

Introduction

This document is a look at both the technical aspects of a microprocessor project and open source development. The technology involved in microprocessors and the philosophy and practices of open source development are first explained, before the OpenRISC project, a project combining the two, is presented. This project is then evaluated and the results of the development effort and the role open source has played are discussed.

This thesis was produced to satisfy the requirements of the Master of Science in Engineering at KTH, Sweden. The work was done while in employment at ORSoC AB, a fab-less design house based in Stockholm, Sweden.

The field of microprocessor architecture design has continued to evolve since the 1970s. The major developments of the technology are presented in the second section. This overview will attempt to select the important developments up until the late 1990s. This section will then look at the emergence of reconfigurable logic, and how that has played a role in digital design. Finally there is an introduction to the concept of open source and its genesis. This should provide enough of a background on the technology and philosophy the OpenRISC project is based on.

The third section further introduces the OpenRISC project, and the OpenRISC 1000 architecture. The basics of the architecture, the implementations and support tools are presented. This mainly presents the state of the project as it was before the work outlined in section five commenced. This should help frame the work outlined in the next section.

The fourth section will outline some of the advances in microprocessor design and the open source movement from the beginning of the OpenRISC project in 1999 to today, nearly twelve years on. In particular, the massive success of open source development is bought into focus and hopefully reinforces the motivation behind this development model.

The fifth section outlines the work done on the OR1K implementations and toolchain during the last three years. The work performed during the course of this thesis project is presented here. Work the author was directly responsible for is presented in greatest detail. The work involved in upgrading the toolchain and

Linux kernel ports, which the author was not wholly responsible for, is outlined.

The sixth section is a critical look at the OR1K architecture and implementations. It identifies issues with the RTL implementation, and limitations of the defined architecture. These details will form the basis of the recommendations presented in the final section. The issues faced by the open source hardware community will also be presented in this section. This is a discussion of the aforementioned intersection of open source and hardware design.

The final section begins by making recommendations regarding the OR1K implementations. Primarily this is a list of features that could be added to increase the potential of the platform. The lack of testing, identified in previous sections, is considered to be an issue that should be addressed to increase the appeal of the platform by ensuring its functionality is well proved. Ways of improving the accessibility of the platform are also presented. A critical point for the the toolchain is to ensure an effort is made to submit it upstream, similarly with the Linux kernel port, which will increase both the visibility, accessibility and ease of use of the platform. Finally, it is argued that modifying OR1K to add or alter features is not the best option, and that instead a new architecture should be considered. The initial features being discussed for this successor architecture, named OpenRISC 2000, are presented.

The author was responsible for all RTL modifications to the OR1200, and the entire re-implementation of the OpenRISC Reference Platform SoC (ORPSoC) project outlined in section five. The author also contributed to the or1ksim project in the floating point unit and testsuite work described here. Finally, the author has been responsible for, or involved in, many elements of the project too numerous to name here, but has at the very least read, if not tested and reviewed, every line of code that has been edited or contributed to the project over the last three years. This was done with support by, and in conjunction with, the core group of OpenRISC contributors, all of who's work I've attempted to acknowledge accurately in this document.

Chapter 2

A Historical Perspective

2.1 The Dawn Of The Microprocessor Era

The dawn of the microprocessor era was ushered in by advancements in design technology enabling Large Scale Integration (LSI), or circuits containing up to 10,000 transistors. The first microprocessor emerged in 1971, with the term coined a year later by Intel upon the introduction to the market of its 4004 CPU (1).

Early implementations of microelectronic systems typically consisted of various discrete components acting together to form a system capable of arithmetic calculation and system control. With increased integration of circuit elements onto a single silicon substrate, solutions emerged where these discrete components could be contained in a single chip. The microprocessor, in a strict sense, is capable of performing three basic tasks: arithmetic or logical operations, memory transactions and control decisions(2). The earliest example of this is the aforementioned 4004 CPU, which Intel developed when tasked with developing a processing system capable of being used in an array of business calculators. After being presented with an overly complex proposal for a calculator design, Intel's engineers decided to create a simpler, yet more versatile solution to the problem.

It is argued that the advancements enabling early microprocessors were inevitable, and that the microprocessor's rapid acceptance was in many ways predetermined (1). The technology's popularity is then thought to have driven the advancements in LSI and VLSI design for non-memory applications, advancements that would not have happened without the innovation of the microprocessor (2).

The microprocessor design and manufacture revolution was underway by the middle of the 1970s. Greater potential for processor and system architecture was realised immediately. Within two years there was an array of implementations offering a great deal of variability among them in terms of instruction sets, capability and speed(3). This period witnessed innovations that still occupy the cutting edge of technological development today. The design of micro-architectures, or sub-microprocessor level design, became a common design method when partitioning blocks within the microprocessor. One such micro-architecture approach is that of

microcode programs, consisting of simpler instructions, triggered by more complex, code-dense instructions. The complexity of the designs ultimately increased with the the capability to implement them.(4).

It would be remiss not to mention the infamous *Moore's Law* pertaining to advancements in fabrication and the number of transistors that can be implemented on an integrated circuit over time. Gordon Moore observed in 1965 that reduced cost is one of the big attractions to integrated electronics, and as process technologies advance, the per component cost plummets.

> The complexity [of a manufactured circuit,] for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years(5).

This prediction remains accurate 40 years later, although perhaps not much longer. It is progress in this area (among the leaders of which is the company co-founded by Moore, Intel) that has driven the computing revolution and changed literally every facet of life.

Advancements in software design were also to come. Even though high level languages had been in use during the 1960s, the arrival of LSI and higher density memories meant the increased potential of software. Even by the mid-seventies, in most microcomputer applications, software development accounted for fifty to eighty percent of total development costs(6). This resulted in a shift away from programming in machine code toward higher level languages which typically generated, at the time, a ratio of five to ten assembly instructions per source code line (6). This spurred a renewed focus on compilers for the rapidly improving hardware and instigated a shift in the approach to both software and hardware. There was a questioning of the value of having such complex hardware if the compiler were able to synthesize streams of simpler instructions to do the same job. Others, such as researchers at IBM, wondered if most of the instructions in sets, designed with an emphasis on code density, were even used all that often. At a time when the sophistication of compilers was considered less than that of the hardware, a different approach was taken to hardware-software interaction in an attempt to address the balance. Other motivations were in breaking free from the microprogram trend, and removing what was seen to be an expensive overhead in performing the most frequently executed instructions(7).

At the time of the mid-seventies the acronyms CISC and RISC had yet to be termed, but the realisations that would lead to these distinctions in architecture approach were being made. John Cocke of IBM's Thomas J. Watson Research Center worked with a group that took the approach of simplifying things and stripped back hardware and high-level language to their elementary components. Hardware features barely used by existing compilers, such as the ability to use operands from main

memory, were scrapped. Their chosen high-level programming language (PL/I) also had those features scrapped which appeared to defy reasonable translation to machine code(7). This work resulted in the 801 microprocessor, and with it they confirmed their predictions regarding this approach on overall system performance. The 801's capability to be on par with, if not surpass, microcode architectures in performance terms was due to compilers which had greater opportunity for optimization due to simpler instruction formats and a greater number of registers. Also put to rest were concerns about path-length, or the number of instructions taken to perform specific operations, with results indicating execution times on par with microprogrammed implementations. According to the paper released by IBM's research group, "on the whole, the code generated for the 801 confirmed our belief that an exposed vertical-microcode machine was a very cost-effective, high-performance machine"(7). And so the RISC architecture approach was gathering momentum.

Much was to be made of the difference between the RISC and CISC approaches, and their respective benefits and limitations. The simplest distinction that sets them apart is the number of opcodes supported by the instruction set. The idea of a complex instruction set architecture, is to move complicated software functions into hardware. This complicated software function can then be represented by a single machine code instruction. By itself this seems like a good idea until every tricky problem is moved from software to the hardware, which then suffers under the burden of then being a highly complex implementation. Early RISC designs tended to have just 30-40 instruction opcodes, whereas CISC machines of the time had hundreds.

Two of the most influential public RISC architecture research projects occurred during the early 1980s. The first project chronologically, and that which coined the acronym RISC, was undertaken at the University of California's Berkeley campus by David Patterson and Carlo Sequin. This work resulted in the RISC-I processor.

The Berkeley team's motivation for the research came from the observation that the architectures of the time were increasing in complexity commensurate with advancing implementation technologies, and that this lead to unnecessary increases in design implementation time, inconsistency and errors(8). Better use of the scarce resources (less silicon used up by control logic means more space for cache or registers) they argued, could be had by an architecture with an instruction set, reduced in size and complexity, which would also result in reduced design time, errors and execution time of each instruction.

With the self-imposed restrictions on instruction width, memory accesses and execution time (a single cycle), they predicted results tending towards performance worse than existing architectures. Despite their prediction they found comparable performance largely, they argue, due to the use of register windows(8). Register windowing requires a large number of registers, typically over one hundred, and when performing a function call, instead of memory accesses to save register values, a register window moves to a different set of registers and the called procedure then uses those within the newly shifted window.

Other architectural choices were to use a *delayed jump*. When a program jumps

or branches to a new address for execution, there is typically a latency involved. A *delayed jump* feature always executes the instruction following a branch, and therefore uses up some of the time required to switch address and perform a fetch. In implementation terms, it reduces the temptation to add complexity to the fetch logic to reduce wait times when a program branch is taken. Techniques such as this, although appearing to burden the user by having to fill an extra instruction slot for each branch, taken or not, had the execution time penalty largely mitigated by compilers that could optimize around the problem by using the instruction after a branch efficiently.

Like other RISC projects, it was essential that a high level programming language was supported to aid in programming the new architecture. The *C* language had come into fashion by the early eighties and was chosen primarily due to its popularity at the time.

The implementation of the first RISC processor at Berkeley showed that its control logic occupied only 6% of overall silicon area, compared with at least forty in implementations of its contemporaries. The large number of registers, though, ultimately made it a larger chip than most, but with six percent of the logic being the irregular control logic, and over 90% a highly regular memory layout, time to manufacturing was well below the others. Performance benchmarks indicated it was ahead of the rest in a standard set of C language algorithms.

Just as the microprocessor was seen to be a more efficient approach than the usual electronic systems of its day, the UC Berkeley work on RISC architectures showed how a reduction in complexity can be a better solution in every aspect.

Around the same time as the work at Berkeley was proceeding, a similarly RISC-based research project was underway, headed by John L. Hennessey, at Stanford University. The processor they implemented aimed to be of modest size, low power dissipation and high regularity. The regularity aspect focused on keeping the five stage pipeline as full as possible at all times. The compiler was an integral part of this, aiming to optimize the machine code to achieve high pipeline utilization(9). This architecture became known as MIPS, short for "Microprocessor without Interlocked Pipeline Stages".

As the work on the MIPS architecture progressed, they identified the trade-off of silicon area between processor logic and memory cells as being critical to performance(10). They noted that as instruction rates increase, the bandwidth and latency of memory systems becomes a important factor in determining overall performance. By increasing the amount of cache available on-chip, this limiting factor is then mitigated somewhat. Additionally, when compared with CISC, RISC instructions typically perform less complex operations, and thus a continuous stream is required to match the performance of CISC ISAs. This makes on-chip caching crucial if any level of performance, comparable to CISC machines, is to be reached.

By the end of the 1980s there were several RISC architectures in production targeting everything from embedded to desktop computers. The architectural design from the work done at Berkeley was the basis for the Scalable Processor Architecture, or SPARC, developed by *Sun Microsystems*. The goal of SPARC was to

define an architecture and have the cost/performance ratio of implementations scale with, or at least track, improvements in fabrication technology, while staying ahead of CISC architectures in terms of performance(11). *Sun* released the specification publicly (later turned into the IEEE Standard 1754) and encouraged other companies to implement it.

Early implementations of SPARC microprocessors included the Cypress Semiconductor CY7C601 and the Fujitsu/Sun MB86900, and were first used in Unix workstations in 1987, replacing processors from Motorola's 68000 family of CISC microprocessors(12).

The first commercial offering of the MIPS design was from the company spun out from the research done at Stanford, *MIPS Computer Systems*, and was named the *R2000*. This chip implemented the MIPS-I ISA. These early MIPS processors were adopted by US based firm *SGI*, a market leader in graphics processing at the time.

Another notable RISC architecture emerging from the 1980s was by a British company named ARM. In 1989 their paper announcing the ARM3 CPU indicated it was targeted at low-cost high-performance personal workstations(13). Despite this attempt, and others, to target the desktop market they would all largely lose out to Intel's dominant x86 CISC architecture during the final decade of the twentieth century century. ARM soon targeted their designs toward embedded computing, focusing more on efficiency than performance. This would prove to pay off in the long run.

Despite all of the progress on the RISC architecture front, Intel's *x86* CISC architecture took over from RISC-based designs in a majority of desktop computer implementations at the beginning of the 1990s. Beginning with Intel's 8086 CPU from 1978, a string of backward compatible microprocessors were released over the next decade and a half, gaining increasing market share. Around the early 1990 there was a boom in personal computer ownership, mostly beginning with Intel's 386 and 486 chips. Their first superscalar chip (ability to execute more than one instruction at a time due to multiple pipelines) known as the Pentium, further enforced their market dominance. Intel's work toward addressing compiler complexity, and advancements in fabrication technology gave them a level playing field on performance terms with low- to mid-end RISC-based workstations(15). This factor, in combination with the large legacy code base that was entirely compatible with the Pentium, meant it improved its already dominant market position for desktop systems.

In an attempt to produce a RISC-based competitor for the commodity PC market, in 1991 two large firms, IBM and Motorola, and an up-and-comer, Apple, formed an alliance with the goal of creating such a family of microprocessors. Citing the eighty five percent market share Intel enjoyed with its x86 architecture in this area, they saw no reason why a RISC-based competitor couldn't be introduced(16). Despite initial benchmarks indicating an up to five times performance increase over the Pentium architecture, and relative success in Apple desktop PCs, the PowerPC architecture failed to significantly dent the popularity of x86 machines. This has

been put down to a more advanced fabrication process available to Intel, and a large legacy x86 codebase.

Throughout the 1990s, Intel's CISC implementations started to blur the line between pure CISC and RISC. The core execution units in the micro-architecture began to resemble RISC machines. Instruction decode began taking a CISC instruction and issuing a sequence of instructions from a RISC instruction set(14).

While Intel and the PowerPC consortium pushed the limits of processor architectures for commodity PCs and servers, the embedded microprocessors market began to grow rapidly. The architectural considerations for embedded applications and workstations are quite different. Mobile embedded applications have energy and power efficiency, and thus battery power use efficiency, as their highest design priority. Real time embedded applications require architectures designed with low latency as a priority. Mobile multimedia applications present a challenge in the attempt find the right balance between performance and low-power features.

Tightly defined applications of microprocessor-based systems, where the machine is typically not easily reprogrammed or retasked, are referred to as embedded systems. These systems are used regularly in telecommunications, networking, multimedia and appliance applications. Typically, if an application is physically small, the system is designed specifically for that application only and the implementation is generally not reusable or easily disassembled into its constituent parts.

The large market that opened up during the 1990s for embedded computers saw ever increasing use of previous generation microprocessor architectures relative to what was selling in desktop computers. The typical embedded processor of the time were power and size conservative versions of chips that originated in the 1980s, such as the Motorola 68k CISC platform and assorted 32-bit RISC machines. The size of this market by 1995 was valued at around one billion dollars(18). Significant in this market were the chips based on a processor by the UK-based firm *ARM*. ARM's ISA was their own take on the RISC approach to microprocessors and proved to be rather popular during this period. Other competitors in this market were Motorola's Coldfire 68000-based CISC microprocessors and embedded variants of PowerPC and MIPS processors.

As the advancements in VLSI manufacturing technology continued, geometries were made ever smaller resulting in far more design potential on a single die than ever before. By the mid-90s VLSI implementations of high-end processors were around the ten million transistor mark. Just as the LSI advancements of the seventies saw the consolidation of the discrete components making up a processor onto one chip, so too did the fruits of VLSI advancements in the 1990s provide increasing room on chips to incorporate more complex logic. Known as system-on-a-chip, *SoC*, these designs typically combined the processor, memory controllers and peripherals onto a single die, and further shrunk the physical size of embedded system designs.

Over the final three decades of the twentieth century the technological advancement made by the semiconductor industry had commoditized high performance processors and ushered in a wave of personal computing and telecommunications that changed the way we worked and lived. There is no doubting there was an

important role played by the microprocessor in this transformation.

2.2 FPGAs and Soft-Core Microprocessors

Semiconductor device fabrication is a highly complicated, expensive and lengthy process. The expense involved typically means designs intended for silicon chip implementation get few opportunities to have prototypes fabricated before high-volume production begins. This places a great deal of importance on the testing and verification phases of a design before fabrication.

Simulation of a design intended for fabrication, at various levels of abstraction, has traditionally been the method used to confirm functionality according to specification, and that it meets timing (requirements relating to signal propagation time throughout the design) and other physical constraints. Computer simulations of large designs have their drawbacks. One is that the simulations occur at a rate many orders of magnitude slower than the circuits actually operate at. This means the testing of functions which require large simulation time frames are cumbersome and difficult to test.

Breadboarding was one technique for testing the logical function of a system, however this became impractical as designs grew in size, and more appropriate modeling of the final integrated system was desired(19). Programmable logic solutions before the early 1980s included those implementing combinatorial logic functions from a ROM's address inputs. Devices with sequential logic elements eventually emerged. However these were typically small and of little use when prototyping larger designs.

At the time of the early eighties, although silicon real estate was considered extremely valuable and thus strictly rationed, Ross Freeman bet that LSI advancements according to Moore's Law would enable chips with enough transistors to allow arrays of programmable logic blocks. The company he founded, Xilinx, offered its first chip in 1984, containing logic cell arrays (LCAs), programmable by the user into just about any configuration they wished. These were referred to as field programmable gate arrays (FPGAs) and provided increased capacity over the other programmable logic devices (PLDs) of the day.

Dual roles awaited FPGA technology - one as a final implementation target of a design, and as a prototyping stepping-stone on the way to final fabrication of a design.

Designs implemented in FPGA almost always have inferior area use, performance and power characteristics when compared to an application-specific IC (ASIC) implementation. This is due to the inherent nature of FPGA architectures. The fundamental components of FPGA fabric such as configurable connection lines (routing) and configurable logic blocks (CLBs), impose an inescapable area overhead. Increased interconnect lengths, and the use of configurable look-up-tables (LUTs) implementing combinatorial logic, result in increased propagation time of signals between registering elements, and thus degraded speed performance overall. These

additional resources implementing configurability contribute to both static and dynamic power consumption of a design implemented on FPGA.

The great benefit of FPGA implementations is that there's no significant costs involved when altering a design, whereas they may be crippling if a change is required to an ASIC design ready for production. This changes the development cycle of FPGA-targeted designs, as they can be prototyped early and often. FPGA prototyping can have a positive effect on time to market for a product, possibly eliminating entirely at least one prototype run and the time spent waiting for wafers to return from fabrication. It must also be mentioned, however, that this applies to the purely digital realm of microelectronic circuit design - FPGAs with arrays of analog primitives do exist but are rarely used to prototype analog parts of a mixed-signal design.

If a design is not intended to sell in large volumes, and neither complexity, performance nor power consumption are a concern, FPGAs can potentially be an option for implementation than ASIC fabrication. Although one's competitive advantage is usually lost the moment a highly innovative solution is revealed to the marketplace, a concept might be proved at relatively low cost and time-to-market on FPGA and implemented in ASIC at a later time to provide an improved, cheaper, solution.

VLSI manufacturing advances continued throughout the 1990s and so too increases in FPGA capacities. Large digital system designs which were once only able to be implemented as ASICs, then had the option of targeting implementation on FPGA instead.

The microprocessor, either as a discrete component or alongside other logic on the same chip, was an obvious candidate for implementation on FPGA. This introduced greater potential for design space exploration by having custom computing logic implemented alongside a standard microprocessor(20).

Digital circuit designs are typically partitioned into functional blocks, referred to as modules, or *cores*. A core will consist of sub-blocks that help implement the functionality. Cores can range in size up to that of an entire microprocessor. A core might occupy an entire FPGA in one implementation, while only being *instantiated* among others on a larger FPGA or in an ASIC. They are typically described using a hardware description language (HDL) at a level of abstraction known as register transfer level (RTL).

The process of taking the RTL description of a design and converting it into a list of gates and connections between them, then allowing implementation on the target technology, is known as *synthesis*. This can be thought of as akin to the compilation of software - that is taking a program in a higher-level language such as C, and converting into machine-specific primitives, or machine instructions in the case of the software. For "hardware" designs done in RTL, there can be a slightly varying level of abstraction, however it is the synthesis step which will generate the list of primitives, or logic gates, for the targeted technology. The result of synthesis, known as a *netlist*, is at a level of abstraction referred to as *gate* level. Simply put, it is this netlist that is then used for further processing into an FPGA configuration

or into a layout for an ASIC.

Cores can be, and often are, developed independently by design houses and licensed or sold individually. They would be licensed or bought by firms who would then typically use the core as a component to be instantiated in a larger design. The commodity in this case is often referred to as an *IP core* (*intellectual property core*) in the sense that the design is the IP of the third-party developer and the right to use it is being licensed to the customer. The terms *IP* and *core* are used interchangeably and in combination to mean the same thing.

IP can be in a variety of forms when licensed. When it is in the form of synthesizable RTL then the IP is referred to as a *soft core*. If it is in a less abstracted form, such as a netlist or a post-layout format ready for fabrication, it is known as *hard core* IP.

Continuous innovation in semiconductor fabrication technology has seen the available "real estate" on chips increase in line with the 1965 prediction by Gordon E. Moore. The ability of digital design engineers to make use of these extra transistors has not kept pace with this increase in fabrication capability(21). The time to market requirements of these increasingly complex designs has remained static, if not tightened. This has lead to the emergence of the IP core industry made up of firms specialising in developing and licensing IP to people building systems for FPGA or ASIC implementation. This allows design teams to assemble a system consisting of commodity components developed by third parties to implement support for standard communications protocols such as Ethernet, IIC or SPI, while concentrating their design efforts on what it is that makes their design unique or particularly valuable.

This model has proved successful. The market for silicon IP (SIP) is valued at three hundred and twenty million U.S dollars in first quarter 2010(23). Despite revenues receding following an international financial crisis in the last few years of the first decade of the twenty-first century, the electronic design automation (EDA) and IP industry has maintained quarterly revenues in excess of a billion dollars since the beginning of the twenty-first century(22).

The best performing microprocessor IP company over this time by revenue has been ARM. Although they provide a vast library of IP, their success has primarily been with their soft core RISC microprocessors. In most cases a microprocessor is a complicated and essential part of a SoC, making the use of pre-developed IP a sensible choice. ARM pioneered the so-called *fabless* semiconductor technology company - meaning they designed IP but were never involved in fabrication, only the licensing of it to other firms who instantiate and fabricate it. Although they originally preferred selling *hard* IP, that is a core which is already synthesized and layed out for a fabrication technology, they have since started to offer *soft* IP for processors to select customers alongside a considerable library of other IPs for memory and system interconnect.

Other vendors of microprocessor IP targeted for implementation in ASICS include Synopsys, MIPS, Freescale and Tensilica. Combined, they account for volume in the tens of billions of microprocessors sold globally each year (23)(24)(25).

There is also a slightly different group of soft microprocessors, primarily targeting reconfigurable hardware. Three of the largest FPGA vendors, Xilinx, Altera and Lattice, all offer their own thirty-two bit RISC microprocessor cores. The two largest FPGA device vendors, Altera and Xilinx, provide the Nios and Microblaze cores, respectively. They are considered hard cores in that the source RTL is not made available and they can only be implemented as netlists on their respective FPGA technologies.

A group of *open source* cores exists which are not restricted by technology, and are inherently soft cores. This group is usually developed by enthusiasts within open source communities, or in a few cases, developed by commercial entities before being *open sourced*. The notable thirty-two bit microprocessor cores in this category are the *OpenRISC 1200*, the *LEON SPARC* processors, and the *LatticeMico32* core from the aforementioned Lattice Semiconductor firm.

For the majority of developers targeting a reconfigurable implementation their options for thirty-two bit microprocessors are between their FPGA vendor's offering and the freely available open source soft cores. However, when it comes to being able to develop and sell a product based on these cores, there are additional considerations regarding the licensing of the designs. These licensing related issues will be discussed in a later section.

The advantages of a true soft core over a hard core, obfuscated via netlist encryption or similar, are to do with the openness of the design, and freedom from restrictions on what one can do with the work. With a truly open source design there is the option of customising the RTL description to implement optimisation or desired functionality. Portability and product end-of-life concerns also do not arise with the RTL description of the design.

As FPGAs and CPLDs have become cheaper and more power efficient they are being considered for use instead of ASIC solutions in certain applications. If criteria such as time to market or field updatability are crucial, with peak performance and power use less so, then an FPGA implementation may be suitable.

For an appropriate application targeting typically mid-to-low volume and non-mobile use (and thus relaxed area and power constraints) FPGA-based systems using softcore processors are of great use as they allow a high degree of customisation and flexibility. Apart from strictly market-oriented uses of such cores, there exists significant hobbyist and academic use of such cores in uses ranging from tinkering to design space exploration of new processing concepts.

2.3 Free and Open Source Software

A simple interpretation of what is meant by the term *open source*, when used in the context of describing a software program or hardware design, is that the design's sources are somehow made available to look at. Although it is a broad and potentially ambiguous term, a commonly agreed upon definition is in a document called the Open Source Definition (OSD) published by the Open Source Initiative (OSI).

The definition isn't an open source license itself, rather something to measure distribution terms against to determine if they comply and, if they do, can then be said to be open source(36). What is not immediately clear is what could, or should, be done with a copy of the design source, practically and legally. When *free* software is used to describe open source software, it is referring not to the cost to the user, but the rights of the user. The Free Software Foundation (FSF) provide a definition to show clearly what must be true about a piece of software for it to be considered free(35). The term free open source software (FOSS) is used to refer to software adhering to both the OSD and FSD. Free and open source software is an inclusive term which covers both free software and open source software which, despite describing similar development models, have differing cultures and philosophies.

Free software focuses on the philosophical freedoms it gives to users while open source focuses on the perceived strengths of its peer-to-peer development model(37)(38). However, the proponents of free software and those of open source software do not see eye to eye. In practical terms the license for software is controlled by the original author and indicated via inclusion of license text with the design source.

The gamut of interpretation of the *open source* concept has emerged over the past twenty years. Most adopt one of two positions, which can be described as either pragmatic or ideological, in relation to the issues of enforcing the sharing of code and the acceptance of the use of FOSS in conjunction with proprietary software. It is these differences that cause proponents of free software to distinguish themselves *within* the open source community. With the number of published open source licenses approaching the hundreds(34) there are many examples of how different organisations interpret the meaning, and make use of, open source.

Computer software has predominantly been, and continues to be, the subject of open source licensing. Although the concept of open source licensing has now been applied to other areas, it all started with computer software. When IBM began computer sales on a large scale in the 1960s, their software came bundled as source code. A decade later, however, they began to "unbundle" the software, and it became usual for computer manufacturers to provide only software that did not have its source disclosed, limiting the ability of competitors to run the same code, but also eliminating the ability of the code to be *free* to modify and share (26). It has also been helped by the fact that the Internet practically elminates the cost of distribution, and the technology to use and developed software is relatively inexpensive.

The genesis of two major open source licensing schemes occurred in the early 1980s. First, at MIT in Boston, Richard Stallman resigned from his position over a disagreement with the ever increasing use of proprietary software in his department. He begin work on the GNU project which focuses on implementing a free and open operating system. Stallman argues that when forcibly restricted from modifying or disclosing how proprietary software works one is unable to cooperate or help other users of the software and are forced to beg the proprietary software developer for any desired changes. His view is that this is antisocial and unethical. He challenges the stance of proprietary software developers that they have a natural right to

control the actions of the users of their software and argues this cannot be so and gives the example of the case where the software causes harm (the users cannot object), and that it is an established legal view that the rights of the software producer are not natural, they are actually artificially imposed by Government(27). Another point he makes is that the perception that there would be no software without the rights of the developers to control who sees the source and how it is used is also shown to be inaccurate by the masses of software produced by open source contributors. Stallman founded the *Free Software Foundation* (FSF) in 1985 to promote computer user freedom and to defend the rights of all free software users(28). The FSF sponsors the GNU project.

The second open source licensing group emerged out of work done by the Computer Science Research Group (CSRG) of the University of California at Berkeley. They were developing many applications for a proprietary form of Unix. Despite producing a great deal of code under their own license, there wasn't enough to constitute an O/S free of the AT&T UNIX source code license that covered the remainder of the kernel. However, William Jolix, a Berkeley alumni, was working on the equivalent components that were missing to make an entirely unencumbered O/S. By 1993 these parts were finally merged and released as *386BSD*, under what is called the Berkeley Software Division (BSD) license, which places very little restrictions on the code's reuse.

The BSD license and the GNU Project's General Public License (GNU GPL) are two of the first such open source licenses. Both provide the freedom to use open source software for any purpose and permit the modification and distribution of its source code without having to pay any royalties. The differences between the two highlights an ideological difference among the proponents of open source.

A significant point of difference between the BSD and GPL licenses is that the latter allows you to

> modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications ... provided that you also ... cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License. (GPLv1) (29)

The GNU GPL is referred to as a *viral* license, in that any design making use of code already licensed under the GNU GPL must then, itself, be licensed under the GNU GPL, or any license judged as equally unrestrictive by the FSF. Put simply, a condition of use of GPL'ed code is that your design must then be licensed under the GPL, or a compatible license. Licenses deemed compatible with the GPL by the FSF are typically similar in the freedoms it ensures for the software. In other words, you are forced to share, if you were planning not to, and to be subject to the GNU's license, or an approved equivalent. The license, in effect, *infects* the code making use of it. The GNU GPLv3 requires that when a project adopts this

license the source code must be made available and that patents or digital rights management (DRM) do not inhibit others from using the design. The original BSD license, in contrast, simply requires recognition be paid to the original authors by the inclusion of their name in the source code and all advertising. Later versions of the BSD license dropped the requirement of names in the advertising material.

The GNU GPL is in a way,ironically, *more* restrictive than the BSD license regarding ones freedom to do what one will with source code. It stipulates the modified code *must* be made available, and any design used with GPL code must also come under the GNU GPL. However this is no different to any proprietary license, written by an employee of a company; all of the code they modify or create comes under the company's proprietary license. In the GNU license's case, however, users are forced to keep their design open and as free as the GNU GPL makes it, in the same way the employee is forced to keep their code proprietary and secret from anyone who is not the company.

Another point of contention is the combined use of designs where each is under a different license, and has lead to the concept of *license compatibility*. The meaning of *use* here is ambiguous and depends on the context, however in this instance it will mean *using* a precompiled binary format version of a design. In the case that one design uses a GPL'ed binary library, although no source code is seen or modified by the user of the GPL'ed design, the GNU GPL specifies that it cannot be used unless the other design is also under the GPL. The use of precompiled libraries, comprised of common functions, in computer software is very common and is equivalent to the example given here. In the case that a library is under the GPL, anything that uses it (also known as creating a *link* to the library, or *linking against* the library) must also come under the GPL.

However, the question of the actual inclusion within a compiled application of GPL'ed binary (known as *static* linking) is not so contentious - this is considered to be the equivalent of including and compiling the original source - the debate is over *dynamic* linking. This involves using a precompiled software library residing elsewhere (not within a compiled application) when an application is executed. Whether an program that dynamically links to a library is considered a derivative work is a debated topic. The GNU Project considers those applications as derivative works and requires them to adhere to the GNU GPL license requirements. An alternate view is that these programs using dynamic linking are not derivative works(30)(31). A solution for those wishing to write libraries, and not have the strictest interpretation of derivative work apply to them, was proposed by the GNU Project in their Lesser General Purpose License (LGPL). It is a trade-off, allowing them to demonstrate that GNU Project licensed technology is high quality and thus encouraging people's participation in the project, while still retaining some their freedom requirements. The GNU Project, however, prefers developers to release libraries under the GPL, forcing those who use it to contribute their work to the body of GNU GPL licensed software.

These differences of opinion in relation to what constitutes a derivative work, and the ambiguity around other aspects of open source licensing could have consequences

for fields such as hardware design and will be discussed in a later section.

The main difference of opinion, however, stems from the fact that the Free Software Foundation wish to make it impossible for proprietary software to use software released under the GNU GPL. The FSF asks why it is those who are not willing to allow others to freely see or modify their code take advantage of those who do. Other open source licenses, however, are more permissive of the use of their designs, either as source of libraries, in proprietary applications. These more permissive open source licenses are at the more pragmatic end of the ideological spectrum.

The GNU Project's goal of implementing an entirely open and free operating system, was progressing well by the early 1990s, but was missing key lower level components. By this time a Finnish university student, Linus Torvalds, had written a replacement *kernel* (central hardware interface component of an operating system) for MINIX, an inexpensive minimal Unix-like operating system restricted to educational use only. Once Torvalds' kernel had reached a relatively stable state the operating system applications from MINIX were replaced with those made available by the GNU Project. Torvalds then re-licensed the kernel (as the copyright owner this was permissible) to the GNU GPL and the first functioning and wholly GNU GPL operating system came into existence(32).

The combination of Torvald's kernel, known as the *Linux* kernel, and the GNU Project's software applications and libraries has evolved into the most-used server operating system in the world. It is referred to most often simply as the *Linux* operating system, although the FSF prefers it to be known as the GNU/Linux OS. Its adoption among desktop, workstation and mobile platforms isn't as high, but are increasing rapidly as operating system packages, known as *distributions* or simply *distros*, become more widely available and provide equivalent, if not better, user experiences than proprietary operating systems.

Its success demonstrates the potential of the free software development model. It has not only disproved derision, such as that it is a disincentive for innovation, or is incapable of providing a commercial business model(33), but prospered and continues to gather momentum. Examples of other successful and widely adopted open source projects are the web server Apache, office utilities suite OpenOffice.org and the Mozilla project which creates email and web browser software. Although, the latter pair were not originally open source, the release of their source code under open source licenses was significant, and they continue as popular projects today.

Following increased adoption of open source software throughout the 1990s, in 1998 an organisation named the *Open Source Initiative* (OSI) was started by some software developers who set out to convince people that free software (as it was commonly referred to at the time) had a place in the commercial industry. One of the founders, Eric Raymond, wrote a seminal paper on the way open source development works named The Cathedral and the Bazaar, that gained his ideas and subsequently the open source movement, a lot of publicity and interest.

The paper uses a bazaar as a metaphor for open source development communities consisting of large numbers of developers casually working on various problems

coordinated only by the Internet. A release strategy involving a release each week of work and getting feedback "created a sort of rapid Darwinian selection on the mutations introduced by developers" and appeared to keep the quality of the developed code at a high standard (39). The success of the model surprised a lot of people, and proved it was a viable development model. It is in contrast to highly coordinated and centralised projects, there was "no quiet, reverent cathedral-building here - rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches"(40). Raymond continues on in his essay to recount initial confusion at the success of the model and his eventual efforts at running an open source project coordinated in a similar fashion, and his success at doing so.

The paper, presented in 1997, triggered a mass of interest in open source and the largest projects around at the time, GNU/Linux among others. Almost immediately, there was a wish to participate, with the unprecedented announcement by the Netscape company that it would release its popular web browser as an open source project. In early 1998 the momentum due to incrased interest in open source caused the initiators to attempt to make the most of the attention people were paying and made an announcement requesting that the community refer to the software as open source, rather than free(41).

The OSI was conceived as a general educational and advocacy organisation, and the initial members agreed to promote the term "open source", and adopt the rhetoric of pragmatism and market-friendliness that Eric Raymond had been developing. Of course, this ran slightly contrary to the ideological position of Richard Stallman's GNU Project. In response to one of the OSI founders, Stallman is quoted as saying "Free software and Open Source seem quite similar, if you look only at their software development practices. At the philosophical level, the difference is extreme. The Free Software Movement is a social movement for computer users' freedom. The Open Source philosophy cites practical, economic benefits. A deeper difference cannot be imagined."

Despite this most fundamental of disagreements of the motives and goals of free and open source software, the proponents of each are sympathetic to the other's cause.

As the popularity and utility of the Internet has grown, so too have online communities for various causes. It is a little bit of a chicken or the egg question about the growth of online open source communities; did the facility of the Internet allow tinkerers to open up their solo endeavors to others, or did the advent of mass Internet uptake and online communication provide the spark for large open source communities. Regardless, what has resulted is countless communities and loose knit groups contributing to open source development of almost anything.

Large web sites for communities focused on computer application software development, such as *SourceForce*, *freshmeat*, *ohloh*, and *CPAN* host upwards of tens of thousands of projects. A group named *Freenode* provide Internet relay chat (IRC) servers where tens of thousands of open source developers gather to interact.

There are a number of smaller free project hosting services aimed at group

development of software such as *Google Code*, *Launchpad*, *GitHub*, *GNU Savannah*, as well as the aforementioned community sites.

One site, starting just after the year 2000, named *OpenCores* focuses on providing a site for the open source hardware community. It was among the first to provide for the hardware development community and is currently the largest, with over one hundred thousand users and almost one thousand projects.

Chapter 3

OpenRISC Overview

The OpenRISC project was started in 1999 by a group of Slovenian university students. Their aim was to create an open source microprocessor architecture specification and implementation. Two years later they had produced an architectural specification, architectural simulator and Verilog HDL implementation complete and made publicly available through their new open hardware community, OpenCores. It has seen use by industry, academia and hobbyists in both FPGA and ASICs. This section will look at the OpenRISC project, the first architectural specification, the OpenRISC 1000 (OR1K) family of processors, and the first HDL implementation - the OpenRISC 1200 (OR1200) processor.

3.1 Beginnings

The first public mention of the OpenRISC project was early in the year 2000 via the EE Times, an electronics industry news media publisher. One of the project's initiators, Damjan Lampret, was interviewed and mentioned that by the end of February designers should be able to download VHDL files for an OpenRISC 1000 core, to go with the already available C compiler(43).

The article continued on to say that offerings such as this could potentially alter the SIP market landscape similar to the way Linux altered the operating systems market. Lampret continued on to announce the *OpenCores* site which was designed to host a community for online open source HDL development.

Lampret explained that the OR1K's architecture was inspired by the DLX and early MIPS RISC architectures, both prominent in the seminal text from RISC CPU pioneers John Hennessy (of the Stanford MIPS project) and David Patterson (UC Berkeley's RISC project).

The last section of the article saw Lampret mentioning the potential of replacing a single module in the design, the decode stage, to enable support for the MIPS or ARM instruction sets. The article finished ominously by quoting some major processor design houses indicating they would vigorously protect their IP.

The project attracted interest from around the world, with various IC manu-

facturing companies eager to check out the capabilities of the first architecture and implementations.

3.2 The OpenRISC Project

The OpenRISC project aims to define and implement free and open source families of RISC microprocessor architectures. It provides the instruction architecture specification (ISA) for free under the GNU GPL, and implementations under the LGPL.

It is the flagship project of OpenCores, an online community set up to host and help development of open source hardware projects.

3.3 OpenRISC 1000 Architecture

The first OpenRISC architecture defines an instruction set, addressing rules, exception system, register set, fast context switch mechanism, cache interface, MMU interface and software ABI.

3.3.1 Instruction set

The OR1K instruction set consists of uniform width instructions grouped into 5 subsets.

All instructions are 32-bits wide and 32-bit aligned in memory.

- ORBIS32 - OpenRISC Basic Instruction Set operating on 8-, 16- and 32-bit data.
- ORBIS64 - OpenRISC Basic Instruction Set operating on 8-, 16-, 32- and 64-bit data.
- ORFPX32 - OpenRISC Floating Point Extension Instruction Set operating on 32-bit data.
- ORFPX64 - OpenRISC Floating Point Extension Instruction Set operating on 8-, 16-, 32- and 64-bit data.
- ORVDX64 - OpenRISC Vector/DSP Extension Instruction Set operating on 8-, 16-, 32- and 64-bit data.

Within each instruction subset, there is the further distinction of class I and II instructions. Only class I instructions are *mandatory* to implement.

Instruction Subset Features	
ORBIS32	32-bit integer instructions Basic DSP instructions 32-bit load and store instructions Program flow instructions Special instructions
ORBIS64	64-bit integer instructions 64-bit load and store instructions
ORFPX32	Single precision floating point instructions
ORFPX64	Double precision floating point instructions 64-bit load and store instructions
ORVDX64	Vector instructions DSP instructions

Table 3.1. OpenRISC 1000 Instruction Subset Features

31 26	25 0
6-bits	26-bits
Opcode	Immediate

Table 3.2. OpenRISC 1000 Immediate Instruction Format (I)

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10 0
6-bits	5-bits	5-bits	5-bits	11-bits
Opcode	rD	rA	rB	Opcode

Table 3.3. OpenRISC 1000 Register-to-Register Instruction Format (R)

Instruction formats

As is common with RISC architectures, there are only a few instruction formats. Opcodes are six bits wide. The remainder of the instruction accord to the following formats in tables 3.2-3.7.

These are the Immediate type (I), Register-to-Register type (R), Register-with-Immediate type (RI) and second Register-with-Immediate type (RI2), and types for setting the flag bit in the status register Set Flag Register-with-Immediate (RSFI) and Set Flag Register Format (RSF).

Opc.	Mnemonic	Opc.	Fmt.	Function
0x00	l.j	-	I	PC <- exts(Immediate < < 2) + PC
0x01	l.jal	-	I	PC <- exts(Immediate < < 2) + PC; LR <- PC + 8
0x03	l.bnf	-	I	PC <- SR[F] ? PC + 4 : exts(Immediate < < 2) + PC
0x04	l.bf	-	I	PC <- SR[F] ? exts(Immediate < < 2) + PC : PC + 4
0x05	l.nop	-	I	-

Opc.	Mnemonic	Opc.	Fmt.	Function
0x06	l.movhi	-	RI	rD <- Immediate « 16
0x08	l.sys	-	I	PC <- system-call exception
0x09	l.rfe	-	I	PC <- EPCR; SR <- ESR
0x11	l.jr	-	R	PC <- rB
0x12	l.jalr	-	R	PC <- rB; LR <- PC + 8
0x21	l.lwz	-	RI	rD <- [rA + Immediate][31:0]
0x22	l.lws	-	RI	rD <- [rA + Immediate][31:0]
0x23	l.lbz	-	RI	rD <- extz([rA + Immediate][7:0])
0x24	l.lbs	-	RI	rD <- exts([rA + Immediate][7:0])
0x25	l.lhz	-	RI	rD <- extz([rA + Immediate][15:0])
0x26	l.lhs	-	RI	rD <- exts([rA + Immediate][15:0])
0x27	l.addi	-	RI	rD <- rA + Immediate
0x28	l.addic	-	RI	rD <- rA + Immediate + SR[CY]
0x29	l.andi	-	RI	rD <- rA AND Immediate
0x2A	l.ori	-	RI	rD <- rA OR Immediate
0x2B	l.xori	-	RI	rD <- rA XOR Immediate
0x2C	l.muli	-	RI	rD <- rA * Immediate
0x2D	l.mfspr	-	RI	rD <- SPR[rA OR Immediate]
0x2E	l.slli	0x0	R	rD <- rA < < Immediate
0x2E	l.srli	0x1	R	rD <- rA » Immediate
0x2E	l.srai	0x2	R	rD <- rA »> Immediate
0x30	l.mtspr	-	RI2	SPR[rA OR Immediate] <- rB
0x35	l.sw	-	RI2	[rA + Immediate][31:0] <- rB
0x36	l.sb	-	RI2	[rA + Immediate][7:0] <- rB
0x37	l.sh	-	RI2	[rA + Immediate][15:0] <- rB
0x38	l.add	0x0	R	rD <- rA + rB
0x38	l.addc	0x1	R	rD <- rA + rB + SR[CY]
0x38	l.sub	0x2	R	rD <- rA - rB
0x38	l.and	0x3	R	rD <- rA AND rB
0x38	l.or	0x4	R	rD <- rA OR rB
0x38	l.xor	0x5	R	rD <- rA XOR rB
0x38	l.mul	0x6	R	rD <- rA * rB
0x38	l.sll	0x08	R	rD <- rA < < rB
0x38	l.srl	0x18	R	rD <- rA » rB
0x38	l.sra	0x28	R	rD <- rA »> rB
0x38	l.mulu	0xb	R	rD <- rA * rB
0x39	l.sfeq	0x0	RSF	SR[F] <- rA == rB ? 1 : 0
0x39	l.sfne	0x1	RSF	SR[F] <- rA != rB ? 1 : 0
0x39	l.sfgtu	0x2	RSF	SR[F] <- rA > rB ? 1 : 0
0x39	l.sfgeu	0x3	RSF	SR[F] <- rA >= rB ? 1 : 0
0x39	l.sfltu	0x4	RSF	SR[F] <- rA < rB ? 1 : 0

Opc.	Mnemonic	Opc.	Fmt.	Function
0x39	l.sfleu	0x5	RSF	SR[F] <- rA =< rB ? 1 : 0
0x39	l.sfgts	0xa	RSF	SR[F] <- rA > rB ? 1 : 0
0x39	l.sfges	0xb	RSF	SR[F] <- rA >= rB ? 1 : 0
0x39	l.sflts	0xc	RSF	SR[F] <- rA > rB ? 1 : 0
0x39	l.sfles	0xd	RSF	SR[F] <- rA =< rB ? 1 : 0

Table 3.8: OpenRISC 1000 Class I ORBIS32 Instructions

The table 3.8 outlines the class I ORBIS32 instruction set, that is the set of instructions that must be supported in an implementation dealing with up to 32-bit data. This set of instructions encompasses a majority of the functionality of the instruction set.

3.3.2 Addressing Modes

Memory Access

As is a standard RISC design feature, memory addressing features are limited to simple loads and stores between memory and registers.

Memory accesses are always performed indirectly with the address calculated from the addition of the contents of a register (indirect) and a sign-extended immediate value embedded in the instruction.

Unaligned accesses are not permitted on OR1K.

Memory Operand Conventions

On the OR1K architecture, words are 4-bytes in length, and double words are 8-bytes in length.

Despite there being much reference to OR1K being potentially bi-endian, all implementations thus far have been big-endian (BE).

Program Control

Program execution control with conditional branch instructions occurs by calculating the target address from the addition of a sign-extended immediate value embedded in the instruction to the program counter (PC). This is known as PC relative addressing.

Jump instructions are unconditional and non-PC relative. All instructions must be word-aligned.

3.3.3 Register Set

The set of supporting registers aiding system control is comprehensive. Operating system support is provided by the use of user- and supervisor-level access restrictions

31 26	25 21	20 16	15 0
6-bits	5-bits	5-bits	16-bits
Opcode	rD	rA	Immediate

Table 3.4. OpenRISC 1000 Register-with-Immediate Instruction Format (RI)

31 26	25 21	20 16	15 11	10 0
6-bits	5-bits	5-bits	5-bits	11-bits
Opcode	Immediate	rA	rB	Immediate

Table 3.5. OpenRISC 1000 Register-with-Immediate Instruction Format Two (RI2)

31 26	25 21	20 16	15 11	10 0
6-bits	5-bits	5-bits	5-bits	11-bits
Opcode	Opcode	rA	rB	Reserved

Table 3.6. OpenRISC 1000 Set Flag Register-to-Register Instruction Format (RSF)

to certain system registers. All registers, aside from the GPRs, are referred to as special purpose registers (SPRs).

Register File

The register file registers, otherwise referred as general purpose registers (GPRs), are either 32- or 64-bits wide depending on the implementation. Although it's possible to have just 16 registers, all publicly available implementations of hardware and compiler implement and make use of 32 GPRs. However, the facility to support only 16 is available.

Unit Dependent Registers

The unit dependent registers implement control interfaces to the optional units of the architecture. These units are the instruction cache, data cache, instruction memory management unit (MMU), data MMU, debug interface, performance counters tick timer, floating point unit, multiply accumulate unit (MAC), power management unit, programmable interrupt controller (PIC) and tick timer (TT). Each of these sub-modules will be outlined in following sections. All of these

3.3.4 Privilege Modes

The processor can operate in either supervisor or user mode. The ability to execute in user mode, with reduced privileges, enables operating systems to support execution of code in a controled environment. All processing of exceptions occurs in supervisor mode. The MMUs can control permission of accesses to abritrary regions of memory depending on the execution mode. Almost all special purpose

31 26	25 21	20 16	15 0
6-bits	5-bits	5-bits	16-bits
Opcode	Opcode	rA	Immediate

Table 3.7. OpenRISC 1000 Set Flag Register-with-Immediate Instruction Format (RSFI)

registers are not accessible in user mode, with just a few status registers available for reading.

3.3.5 Fast Context Switch Support

The capability to switch between contexts instantly is supported by the OR1K ISA. This technique reduces the overhead when a context switch is forced due to an exception or interrupt by providing multiple processor resources to save storing the state to memory.

In order to provide a new processor state within a single cycle, multiple register files must be implemented, as well as supporting logic to implement fast context switching.

The current context is indicated in the supervision register in the bits SR[CID]. If fast context switching is in use, exceptions switch back to CID 0, the main context. The context in which the exception occurred is stored in the CXR register.

Although there may be many more contexts than implemented hardware context support, and thus some context may been to be saved back to the stack, it is expected multiple context support can speed up performance by eliminating some delay due to context switching.

3.3.6 General Purpose Registers

The OR1K ISA has a register file consisting of 32 registers either 32- or 64-bits wide, depending on the implementation. Register zero, (mnemonic `r0`), is constantly zero.

It is possible that an implementation can support fast context switching, in which case multiple physical register files will exist in an implementation.

Certain implementations may have a register file consisting of less than 32 GPRs, which is acceptable so long as the software behaves appropriately. It is not clear what should happen if an instruction addresses an out-of-bounds GPR.

3.3.7 Essential SPRs

The special purpose registers (SPRs) are 32-bit wide registers used to interface with the modules defined by OR1K. The following is a list of the essential registers required in an OR1K implementation.

Opc.	Mnemonic	Opc.	Fmt.	Function
0x06	l.macrc	-	RI	rD <- MACLO; MACLO/HI <- 0
0x08	l.trap	-	I	PC <- SR[K] ? trap exception : PC + 4
0x08	l.msync	-	I	mem sync.
0x08	l.psync	-	I	pipline sync.
0x08	l.csync	-	I	context sync.
0x13	l.maci	-	RI2	MAC with immediate
0x1C	l.cust1	-	I	Cust. 1
0x1D	l.cust2	-	I	Cust. 2
0x1E	l.cust3	-	I	Cust. 3
0x1F	l.cust4	-	I	Cust. 4
0x2E	l.rori	0x3	R	rD <- rotate(rA, Immediate)
0x2F	l.sfeqi	0x0	RSFI	SR[F] <- rA == Immediate ? 1 : 0
0x2F	l.sfnei	0x1	RSFI	SR[F] <- rA != Immediate ? 1 : 0
0x2F	l.sfgtui	0x2	RSFI	SR[F] <- rA > Immediate ? 1 : 0
0x2F	l.sfgeui	0x3	RSFI	SR[F] <- rA >= Immediate ? 1 : 0
0x2F	l.sfltui	0x4	RSFI	SR[F] <- rA < Immediate ? 1 : 0
0x2F	l.sfleui	0x5	RSFI	SR[F] <- rA =< Immediate ? 1 : 0
0x2F	l.sfgtsi	0xa	RSFI	SR[F] <- rA > Immediate ? 1 : 0
0x2F	l.sfgesi	0xb	RSFI	SR[F] <- rA >= Immediate ? 1 : 0
0x2F	l.sfltsi	0xc	RSFI	SR[F] <- rA > Immediate ? 1 : 0
0x2F	l.sflesi	0xd	RSFI	SR[F] <- rA =< Immediate ? 1 : 0
0x31	l.mac	0x1	R	mac(rA, rB)
0x31	l.msb	0x2	R	msb(rA, rB)
0x38	l.ror	0x38	R	rD <- rotate(rA,rB)
0x38	l.div	0x9	R	rD <- rA / rB
0x38	l.divu	0xa	R	rD <- rA / rB
0x38	l.extbs	0x1c	R	rD <- exts(rA)
0x38	l.exths	0x0c	R	rD <- exts(rA)
0x38	l.extbz	0x3c	R	rD <- extz(rA)
0x38	l.exthz	0x2c	R	rD <- extz(rA)
0x38	l.cmov	0xe	R	rD <- SR[F] ? rA : rB
0x38	l.ff1	0x0f	R	rD <- FindFirst1(rA)
0x38	l.fl1	0x1f	R	FindLast1(rA)
0x3C	l.cust5	-	I	Cust. 5
0x3D	l.cust6	-	I	Cust. 6
0x3E	l.cust7	-	I	Cust. 7
0x3F	l.cust8	-	I	Cust. 8

Table 3.9. OpenRISC 1000 Class II ORBIS32 Instructions

Group Num	Reg Num	Reg Name	User Mode Permis.	Supv. Mode Permis.	Description
0	17	SR	-	R/W	Supervision Register
0	32-47	EPCR0-EPCR15	-	R/W	Exception PC Registers
0	48-63	EEAR0-EEAR15	-	R/W	Exception EA Registers
0	64-79	ESR0-ESR15	-	R/W	Exception SRs

Table 3.10. OpenRISC 1000 Mandatory SPRs

The supervision register (SR), exception program counter (EPC), exception status register (ESR) and exception effective address register (EAR) are the only registers mandatory for an OR1K implementation. These are outlined in table 3.10.

The multiple exception PC, EA and supervision registers are required when fast context switching is required, and thus the SR[CID] are used to determine which one corresponds to that context.

Supervision Register

The supervision register's contents are described in table 3.11.

Exception SPRs

These registers provide copies of important registers when exceptions occur. The ESR(s) contain a copy of the SR when an exception occurs. The EPCR(s) contain the PPC where an exception occurred. The EEAR(s) hold the effective address in the event that an exception is caused by an address-related fault. Read access when in user mode is granted if the SR[SUMRA] bit is set.

3.3.8 Memory Management

OR1K defines an instruction and data MMU interface and address translation mechanism from the perspective of the programming model. It provides support for three page sizes, page-based protection and demand-paged virtual memory. Translation look-aside buffers are usually implemented and keep the most recently used translation cached. It is possible to have multiple-way associativity on these buffers. The MMUs, using an exception-based processing mechanism, and implementing permission-based page access, provide enough functionality for modern operating systems' paged virtual memory systems. Implementation of the MMU system on either instruction or data buses is optional.

3.3.9 Cache System

OR1K defines a data and instruction cache control interface and multiprocessor coherency model. Implementation of cache is optional. A Harvard cache model should implement both instruction and data caches, a Stanford implementation should only

Bit	Identifier	R/W	Reset	Description
31 28	CID	R/W	0	Context ID (optional)
27 17		R/O	0	Reserved
16	SUMRA	R/W	0	SPRs User Mode Read Access
15	FO	R/O	1	Fixed One
14	EPH	R/W	0	Exception Prefix High
13	DSX	R/W	0	Delay Slot Exception
12	OVE	R/W	0	Overflow Flag Exception
11	OV	R/W	0	Overflow in last op.
10	CY	R/W	0	Carry Flag
9	F	R/W	0	Flag
8	CE	R/W	0	CID Enable
7	LEE	R/W	0	Little Endian Enable
6	IME	R/W	0	IMMU Enable
5	DME	R/W	0	DMMU Enable
4	ICE	R/W	0	Insn. Cache Enable
3	DCE	R/W	0	Data Cache Enable
2	IEE	R/W	0	Interrupt Exception Enable
1	TEE	R/W	0	Tick Timer Exception Enabled
0	SM	R/W	0	Supervisor Mode

Table 3.11. OpenRISC 1000 Supervision Register

use the data cache interface. The cache system can control from direct-mapped up to an eight way, set associative, cache. The control interface allows for write-through (CWT) or write-back (CWB) strategies to be implemented. Cache bypass is possible on data accesses when the top-most address bit is asserted. In conjunction with the MMUs, pages can be marked as caching-inhibited (CI), cache coherent (CC), or write-back. Multiprocessor systems are aided by a page-based cache coherency system in conjunction with the MMU, whereby pages can be marked as CC ensuring that the cache system across the processors will enforce coherency. This is most useful among cache systems implementing any sort of data caching.

3.3.10 Power Management

OR1K defines a power management interface that can control an external clock divider, control doze mode where all clocks in the processor are gated except to the TT and PIC, control sleep mode where all clocks are gated, and potentially the core voltage is lowered, or control suspend mode which must then be reset to enable continuation of execution.

3.3.11 Interrupt System

OR1K defines a programmable interrupt controller (PIC) which is optional to implement. It provides a masking function for up to 30 interrupt request lines, and up to two optional non-maskable interrupts. Edge-triggered, level-triggered, and hybrid latched-level-triggered interrupts are supported.

3.3.12 Timer

OR1K defines the tick timer (TT) unit which is optional to implement and provides a timer counting at the core CPU frequency and can be programmed to interrupt the processor on reaching a programmable value. The TT can be programmed to count in single run, restartable or continuous modes.

3.3.13 Debug Unit

OR1K defines a debug unit providing watchpoints conditional on fetch address and load/store address and data. Watchpoints can trigger breakpoints or increment counters. Counting watchpoints can also be specified. Several watchpoints can be combined to create complex watchpoints. The unit can also be programmed to stall the processor on any exception.

3.3.14 Configuration Registers

OR1K defines a group of SPRs indicating the configuration of the microprocessor implementation. As much of the architecture is optional, and various revisions of implementations may exist, these registers exist to assist software, such as a complex OS, in appropriately controlling the microprocessor. All configuration registers are read only.

Version Register

OR1K defines a version register (VR) consisting of a 6-bit revision number, an 8-bit configuration value and an 8-bit version number. All are used to help identify the processor's capabilities, and any specific software workarounds required.

Unit Present Register

OR1K defines a unit present register (UPR) for identifying which of the many optional units are implemented in the design.

CPU Configuration Register

OR1K defines a CPU configuration register (CPUCFGR) which is used to identify the CPU capabilities and configuration. It holds the number of shadow GPRs (for fast context switching), and if the GPR is less than 32 registers in size.

MMU Configuration Registers

OR1K defines data and instruction MMU configuration registers (I/DMMUCFGR) indicating the number of TLB ways, number of sets, number of entries and if other supported features of the MMUs are available.

Cache Configuration Registers

OR1K defines data and instruction cache configuration registers indicating the capabilities and capacity of any implemented caches.

3.3.15 Software ABI

OR1K defines the software application binary interface (ABI). Defined are the physical sizes of standard scalar types used in the ISO/ANSI C language, the physical layout of structs and arrays, and register usage for function calls and stack management.

The OR1K has two execution modes; user and supervisor. All exceptions are switched to with supervisor mode enabled. Operating system functions requiring supervisor execution privileges can be called via the system call mechanism using the `l.sys` instruction.

3.3.16 Exception Interface

OR1K defines an exception interface describing the cause of exceptions, and state of the system when they occur. Exceptions cause the PC to jump to a pre-defined location in memory to execute handler code. Table 3.12 shows the exceptions defined by OR1K.

3.3.17 Omissions

For brevity, overviews of OR1K features left out include the ORFPX32/64 and ORVDX64 instruction sets, which instructions can cause which exceptions, SPR details for almost all unit dependent registers and parts of the ABI such as position-independent code formats.

For complete information see the OR1K specification (44).

3.4 Comparison

OR1K is much like early incarnations of the MIPS ISA. The main differences are the extra instruction formats (RI2, RSF, RSFI) in OR1K and the flag setting mechanism which sets a flag in the supervision register instead of a '1'/'0' in a destination register as on MIPS.

Exception Type	Handler Address Offset	Cause of Exception
Reset	0x100	Hardware reset event
Bus Error	0x200	Error signal from bus asserted due to nonexistent address, parity error, etc.
Data Page Fault	0x300	Unmapped data location or protection violation in data MMU
Insn. Page Fault	0x400	Unmapped insn. location or protection violation in instruction MMU
TT Interrupt	0x500	Tick timer enabled and hit programmed count limit
Alignment	0x600	Access to unaligned instruction or data
Illegal Instruction	0x700	Illegal or unsupported instruction
External Interrupt	0x800	External IRQ asserted
DTLB Miss	0x900	DMMU TLB miss, reload required
ITLB Miss	0xa00	IMMU TLB miss, reload required
Range	0xb00	Arithmetic overflow
System Call	0xc00	System call instruction executed
Trap	0xd00	Trap instruction executed as software breakpoint

Table 3.12. OpenRISC 1000 Exceptions

3.5 First Implementations

By 2004 the architectural simulator *or1ksim*, and first RTL model in Verilog HDL had been largely completed.

3.5.1 Architectural Model

Transactional level models of microprocessor architectures are very useful for early development of software, and later on as a golden functional reference for other implementations. One such model was developed at the onset of the OpenRISC project, named *or1ksim*.

It is a custom built model, written in C, capable of near-cycle accurate simulation of the CPU pipeline, and transactional level emulation of memory mapped peripherals. It is highly configurable at runtime making it very versatile. It provides a remote debugging interface for the GNU debugger (GDB). It is a high speed model, and allows early code analysis and system performance evaluation.

It comes with a suite of models of peripherals commonly found in SoC designs. All of these cores are models of designs available on OpenCores.

It originated with the beginning of the OpenRISC project in 1999. Initially,

it supported the ORBIS32 instruction set, and emulated all optional modules of OR1K. Including peripherals, support modules and tests, the codebase of the simulator comprised fifty thousand lines of ANSI C code.

3.5.2 RTL Model

The first synthesizable implementation, according to early reports on the OpenRISC project, was a 32-bit single-issue model written in VHDL that achieved 100 MIPS at 100MHz (43). The only implementation made publicly available by 2001 was the 32-bit OpenRISC 1200 (OR1200) written in Verilog HDL. Originally, each major architectural feature included in the implementation would mean an increased model number, OR1300 and OR1400 etc. However, regardless of the configuration, the core has only been referred to as the OR1200.

By 2004 the implementation supported most ORBIS32 class I and II instruction, except for synchronisation instructions and various instructions not generated by the compiler, but still capable of being generated by the assembler.

The additional modules included direct mapped instruction and data cache (write-through scheme only) of up to eight kilobytes in size. The data cache was not fully functional and the instruction cache's bus interface produced invalid accesses occasionally. Instruction and data MMUs were implemented, each with sixty-four entries. The tick timer and programmable interrupt controller were fully implemented. The software debug module did not appear to be correctly implemented and features such as hardware breakpoints were not working. The ALU contained a pipelined multiply accumulate (MAC) unit.

The model was targeted at implementation on Xilinx FPGAs, and was also produced in ASICs by Flextronics from 2003(45).

3.5.3 Support Tools

The software support tools, such as compilers and operating systems are just as important in enabling use of the platform as the hardware implementation. As is typical for most microprocessor architectures, work towards implementing the architecture-specific parts of the GNU binutils and compiler collection (GCC) started soon after the first hardware was working, in around the year 2000. In 1999, Damjan Lampret made the first submissions to the GCC mailing list to announce the architecture(61). In early 2001, the first contributions to *binutils* for OpenRISC support were made by Johan Rydberg(62). By 2002, posts made to the GCC's mailing list indicated the entire toolchain looked to be complete, and further submissions to the respective tools' repositories were made.

By the time of the demonstration of the first OpenRISC ASIC in 2003, there were working ports of the GNU tool suites such as the *binutilities* package, GCC and the GNU debugger (GDB). Additionally, there were ports of the uClibc C library, the full Linux kernel, lighter uClinux kernel, and eCos operating system.

A reference SoC implementation, called the OpenRISC Reference Platform System on Chip (ORPSoC), was released which provided the RTL and software to implement an OpenRISC-based SoC. This platform was intended to be the test harness for the OpenRISC processor, as the OR1200's project came with little in the way of RTL-level testbenches. ORPSoC was targeted at the XESS XSV800 board, containing a Xilinx XCV800 Virtex series FPGA, and a range of peripherals such as Ethernet, audio, VGA, and USB controllers.

3.6 Commercialisation

By the end of 2004, the core of the original developers and contributors had largely stopped their open source contributions to the project, and were instead focused on commercialised version of the OpenRISC platform. In 2005, the company Beyond Semiconductor was launched in Slovenia and employed the bulk of the OpenRISC contributors up to that point. Unfortunately, the company decided to re-license the OR1200 for their own purposes, and re-branded it the BA1200, and ceased all contribution to the project on the OpenCores site.

In 2007, they decided the OpenCores website was to be closed down if suitable backers for it were not found(46). Fortunately some engineers who had worked with the OpenRISC platform at Flextronics, and who had recently started ORSoC, a Swedish fabless digital design house, decided to take over the operations of the OpenCores site(47).

Unfortunately, by this time, the significant progress made on the OpenRISC project in its first four years of development had largely come to a halt. However, the next few years would see a revival of the project, thanks to ORSoC and a small, but dedicated, group of contributors.

Chapter 4

A Contemporary Overview

This chapter will give an overview of the technological and market developments continuing from the time frame covered in the historical overview section. The two central topics are microprocessor design, and the open source movement.

4.1 Microprocessors

The trends in microprocessor design and fabrication that have emerged over the first decade of the twentieth century were largely related to reducing the power usage of processors. High performance processing has been constrained by the inability to cost effectively dissipate the energy generated when designs are pushed to the limits of their operating frequency, and embedded computing has focused on decreasing power usage in an effort to increase the available usage time when operating on battery power.

Despite the various proposals of different circuit technologies in the early 1990s, it became clear by the mid 1990s that CMOS technology had the best characteristics for voltage scaling as process geometries shrank(48). Lower voltages in the circuit result in less dynamic power consumption. Fabrication processes advanced, designs were pushed to operate faster, and by the year 2000 the first one gigahertz processor was revealed by IBM(49). Three years later Intel surpassed the three gigahertz mark(50).

Significant roadblocks to increasing operating frequency soon began to be identified. Primarily the issue was power dissipation, with machines predicted to reach 100 watts of power usage, the limit of cost-effective cooling solutions would quickly be reached(48). As a result, peak operating frequencies were to plateau for the rest of the decade. This meant for performance to increase, alternate routes to increase processing capacity had to be found.

Similarly for embedded processing technology. Consumer electronic trends have been such that there has been a continual and rapid rise in demand for cutting edge portable devices such as mobile phones and media players since the mid 1990s. The net value of consumer electronics sold by the U.S in the 1990s doubled from around

forty billion dollars in 1999 to eighty billion at the end of the decade, and again to one hundred and sixty five billion dollars by the end of the next decade(51)(52). There has been a continual increase in the capabilities provided by manufacturers, and demanded by consumers of portable electronics. The integration of media and telecommunication features into portable devices has been a notable trend that has lead to the demands on processor performance to increase greatly. However, being portable means that must rely on battery supplied power, and the longer the device can go before requiring recharging the better. Thus the two major but conflicting goals for designs are are high performance and low power consumption.

How little power is considered *low* power? One paper from 2005 gives examples of power targets for various mobile applications; playing video at 250mW, playing audio at 75mW, O/S scheduling at 50mW, and standby mode of 3mW(53). Measured power consumption of a microprocessor implementation from Apple released in 2010 indicate the estimated combined microprocessor and DRAM (in single package) power consumption is between 250mW and 520mW during audio and video playback applications(54). However these are relatively high performance designs, capable of running multi-tasking operating systems and compute-intensive media applications. The other end of the mobile, embedded microprocessor spectrum, there is the need for processors to perform basic synchronisation and maintenance functions only, and may be limited to less than 1mW peak usage, and spend most of the time on standby, consuming only micro or nano watts of power.

This, however, looks at *dynamic* power consumption, or primarily the power dissipated due to digital signals throughout the design toggling (charging and discharging gates.) Power use considerations have shifted somewhat as advances in the manufacturing processes have lead beyond the nominal 90 nanometer fabrication process point. As the smallest possible feature size of these fabrication processes decreases with advancements, a quantum effect referred to as *leakage* has emerged, which sees mobile charge carriers tunnel through ever thinner insulating materials (potentially from an active line to a grounded substrate) and ultimately resulting in increased current draw by the circuit, whether it is toggling or not. The static leakage of current this way increases exponentially as the feature size of processes become smaller, and by middle of the first decade of the twenty-first century, was seen as an inhibitor to simply printing ever-growing circuits for mobile or embedded uses. Leakage adds a third constraint to designs targeted at mobile use; area, as the larger the design, the larger the static current component or the designs power usage.

Implementations using the 90 nanometer process node began to see leakage power become more dominant than dynamic power in some designs(55). Mitigation techniques for overall power consumption, including a complete power off for unused modules, among others others, became commonplace and result in up to forty times leakage power saving(56). However, there is still a need to continue increasing performance of these systems while keeping energy use down. Architectural approaches to this include re-approaching certain tasks and implementing logic that can complete algorithmic work that might have otherwise been done on a generic processor.

Another solution is to lower the power use of otherwise unused logic, gaining some room in the power budget which can be spent on other operations(57).

With frequency limited due to power concerns, a trend after the 1990s was architecture-level parallelisation of processing. Various techniques had been used previously, such as fast context switching, hardware threading and superscalar implementations to improve throughput but one widely-used approach was multi-core implementation. This design has provided increases in performance while keeping power costs at a minimum(55). Purpose-designed co-processor units for audio and video encoding and decoding (CODEC), and single instruction multiple data (SIMD) processing modules are now very common modules to include, reducing the time taken to perform common algorithmic tasks, thus reducing overall dynamic power consumption of the system. Microprocessors with the ability to scale their frequency, and in some cases voltage, have been shown to result in up to eighty percent reduction in dynamic power consumption for scheduling tasks, and up to forty percent reduction for MPEG4 playback(53). This means things such as custom processing blocks and power minimisation control logic are must-haves for a modern digital design.

Desktop and server-targeted microprocessor architectures were the first to implement multi-core designs, with embedded and mobile applications processors soon following their lead. For mobile applications requiring increased processing capacity, a multi-core approach provides better power economy than frequency scaling. Despite the software design challenges imposed by new requirements on load-balancing among processes and threads, multi-core implementations ultimately provide more processing capacity.

FPGA-targeted microprocessor designs do not have the operating frequency ranges of an ASIC-implemented designs. This makes multi-core implementation on FPGA a necessary step to increase general purpose processing capabilities. Despite FPGAs and customisable soft-core microprocessors being prime candidates for custom processing blocks to speed up specific tasks, there is still a desire to increase processing capacity.

On the market front, ARM are forecast to become the leading 32-bit microprocessor architecture, by sales volume, in the year twenty eleven(58). ARM specialise in microprocessors for mobile and embedded platforms, highlighting the focus and demand on these areas. Additionally, Intel are focusing on the embedded and portable market as it identifies it as a strong growth area.

The methods of power-conscious design, and multi-core architectures have become, and are likely to remain, standard practice for SoC designers.

4.2 Open Source

The increased development and use of open source software during the first decade of the 21st century has seen it become so common that there is likely to be open source software running on almost every modern computer system. The use of

desktop computer systems and servers running entirely open source and free (as in freedom) software is now commonplace. The development model has been proved, and has been adopted by some of the largest corporations and governments.

Although the world does not yet run on entirely open source software, increasing amounts of computer software, once in the domain of proprietary software vendors only, is now available free of restrictions and, largely, for free. The distinction between the two uses of *free* here is significant. The first "free", as in freedom, is used to indicate the software has no restrictions and the user at liberty to do as they wish, usually in regard to modification and distribution. This is the meaning of the "free" in the Free Software Foundation, and free and open source software (FOSS). It just also happens that a lot of this software is made available for free, where the "free" is used to mean no cost, in the way that the "free" in "free beer" can only mean one thing. However, despite the free of cost of a lot of the software, the "free" is rarely used to indicate this.

Software such as advanced multimedia software, office productivity, graphics tools, operating systems, and communications software all have open source alternatives, developed in the open, and usually free to download and use. Governments and large corporations are increasingly leveraging existing open source software and are becoming significant contributors to the projects of the software they adopt. Recent trends have done much to dispel the image of the legions of lone open source developers being the sole contributors of work. As large commercial entities increase their adoption and utilisation of open software projects, they are becoming, by far, the most frequent contributors. The Linux kernel now has the largest proportion of its code contributions coming not from individual hobbyists or enthusiasts, but instead from commercial entities either working with the Linux kernel in their products, such as Red Hat, or wishing to ensure support for their hardware in the kernel, such as Intel, AMD, and IBM. The same is true for project such as Apache and MySQL.

The plethora of open source software available provides the choice between adopting FOSS but having to release their derivative work, and developing a solution internally or purchasing a proprietary solution and revealing the work to no one. The GNU GCC and binutils project's development and maintenance is being performed largely by those employed by companies with an interest in keeping support or their platforms in top condition. The health of the project, then, obviously is driven by the widespread use of the GCC tools, which is due to the initial strong implementation. The increased commerical potential of open source software has resulted in enterprises contributing resources to these projects rather than, developing and maintaining their own proprietary compiler suite in the case of GCC, for example. This is one of the aims of open source's originators - to provide a base of open source software so rich that developers are better off adopting FOSS and releasing their derivative work and thus increasing the existing mass of software rather than develop from scratch or purchase a proprietary solution. Once a critical mass is reached the participation and contribution back to the project increases and in effect "snowballs", for lack of better term. Although simply described here,

the motivations for adopting and contributing to open source would be many and varied and would depend greatly on the actual functionality of the open source project. Despite its success in software, open source hardware design has not seen the successes of the software world, and the reasons for this will be explored in later sections.

Despite open source software's seemingly unstoppable rise in popularity and use, there have been many detractors of open source development to voice their opinion in recent times. Obviously there's going to be opponents of open source who's established dominance of a niche market may be threatened by an open source alternative, and may suffer diminishing revenues because of it, but these opponents rarely argue interesting or useful points of the debate and suffer mainly from failing to innovate sufficiently, a point which will be discussed further.

There are arguments that open source development impedes competition by reducing the chances a proprietary developer has of developing a similar, potentially better, solution because the open source variant usually gets used regardless of its failings largely because it is initially free of cost to the user. Thus, it is argued, that innovation is stifled because smaller companies can't get a foot in between the open source solution and the products of larger established firms whereas, previously, they might have had an opportunity to prove their technology. The counterpoint to this is that any sufficiently superior proprietary implementation will of course attract users, and all that is occurring is a raising of the required quality-of-implementation bar due to the presence of a free-of-cost alternative. One suggestion here would be that these innovators leverage the existing open source implementation to demonstrate their innovation, and thus improve the open source implementation, hopefully allowing it to compete with any dominant proprietary player. In doing so the developers become knowledgeable about the project they've contributed to, become known among the users of the software, and develop a business providing development or support services for the project they've improved. The obvious down side to this is that they have exposed their innovative technique, and cannot reap the rewards as done traditionally in proprietary software products. However, this is offset by their reduced overhead in developing their product (large proportions of the supporting infrastructure adopted from open source implementation), and their improved potential as a gun-for-hire on the newly improved open source project which, in turn, attracts other contributors who decide to work with the project.

Other standard complaints about open source projects are in regard to the quality of implementation. It is true that open source implementations do vary greatly in quality and functionality and this is typically due to a limited contributor base that is implementing only enough so that it works for their application. For the uninitiated, the barriers to entry for contributing to an open source project can be significant and this is another common criticism of open source development. Larger, better managed projects, typically do not have these issues as they adopt a more professional approach to the development, and typically have full time maintainers who will ensure any new features do not cause errors elsewhere, or *regress* the project's functionality. However, on smaller projects, with only a handful of

contributors, unfinished features can be common. On the other hand as the design is completely open, and although there's a relatively steep learning curve, missing features can be added and problems can be fixed by anyone, as they're required.

There's no doubting some very useful software has been written and contributed to open source software projects, however open source should not be seen as an innovative force, rather a step on the way to further commoditising a technology. Entrepreneurship and the profit motive typically drive the high risk and high innovation firms which are involved in cutting edge technology implementation. The whole premise of the GNU Project (GNU is a recursive acronym for "GNU is Not UNIX") was to develop, in essence, a free, open source copy of UNIX applications and operating system. This was not innovation, rather imitation but with a different goal for the resulting work. The Linux kernel was begun for similar reasons. It demonstrated engineering capability but not ingenuity, at least not at that time. It is not true to say that what has been developed in and around those projects doesn't have its innovative elements. It is one thing to wish to re-implement an existing application for largely academic purposes, and another to wish to invest large amounts of time and money to develop a new concept wishing to see a return based on the innovation, rather than the accessibility, of the design.

One motivation for open source development from a commercial point of view, as already eluded to, comes from the ability to provide and charge for services relating to the open source project. Despite the fact that the IP is publicly available, implementing or customising the project typically requires experience in the relevant discipline and with the project specifically. Companies adopting and maintaining an open source solution usually have only the cost of the skills required to do so, with no additional royalty or licensing fees being paid. The information technology boom of the late 1990s saw a large increase in technology workers, and the lack of any open source solutions meant companies typically had costs of both workers and license fees from large software vendors. Ten years later there now are open source alternatives to the proprietary IT software systems of five to ten years ago, resulting in the cost for the same functionality now being just the necessary expert support staff. This need for expertise could potentially be shared between in-house support staff and contracted workers from specialised consultants, or even the project's developers themselves. In summary, the open source developers give up certain claims over the IP, which potentially was largely not their own idea or considered particularly innovative to begin with, for reward in the form of continued employment performing services related to the project.

Despite open source not being a traditional driver for innovation it does not preclude it from being the chosen development strategy for an innovative technology. Any sufficiently innovative design will usually see worthwhile returns for the investment required to bring it about. However, with open source alternatives springing up relatively quickly, their advantage may not last for long as others take note of what innovative developments have been made and whether they're worth pursuing as open source solutions. It may end up being a choice for the developer whether they pursue open source development to begin with, based on how long any com-

petitive advantage may be maintained with a proprietary implementation versus the ability to provide long term support for an implementation which will potentially see more use (the open source version.) The best of both worlds might be a good approach, where initially the product is licensed for a fee, before the source code is released allowing others to develop and hopefully improve the project. As a product's life cycle draws to a close, it may be advantageous to release the source to allow any extended users to fix problems arising from the inevitable platform updates which occur. There is still a problem here, though, as the proprietary implementation maintains its dominance while there isn't an open source equivalent, the designer may never choose to release the source, but then runs the risk of a different implementation with equivalent functions gaining more popularity, and thus the original designer's idea lives on but not their ability to provide support for it.

This brief look at the current state of open source has indicated that open source has become a force to be reckoned with in the software world, but this is definitely not the case at present for open source hardware development. Reasons for this will be explored in the final sections. However, the modern approach to open source development, and the benefits and motivations of it were presented and indicate there is certainly a case for adopting the open source development approach. There is little doubt that there will continue to be increasing adoption of the open source development and licensing model in the future. As the wealth of available open source designs increases, and the understanding of the approach spreads, it will surely continue to prove itself as a worthwhile approach to the development of technology.

Chapter 5

OpenRISC Developments

This section will outline the developments made by the OpenRISC project since 2008. First the RTL and architectural model improvements and bug-fixes will be outlined, and then the significant improvement in the testing for both of these will be explored. Following this, the progress on the supporting toolchain and software will be presented. Finally, the upgrades to ORPSoC, the reference implementation, will be explained.

5.1 RTL Model

The OR1200 has remained as the main OR1K implementation available on OpenCores.org. Despite the core's specification document indicating no updates since 2001, the RTL implementation was maintained until 2005. The period between 2005 and eight saw no maintenance work on the version of the OR1200 core available in the OpenCores repository.

Renewed interest due to commercial applications of the core saw an increase in contributions to the project in 2009. It is almost certainly the case that the core was developed further by third parties during the years it appeared to lie dormant in its repository, however none of the work was contributed back to the public repository.

The following is some detail on the fixes applied to the OR1200 since 2008. The symptom, or how the issue arose, will be explained, so too the implemented solution.

5.1.1 Interrupt priorities

As outlined in section 6.3 of the OR1K architecture specification, there is a defined order in which exceptions and interrupts, should they occur simultaneously, are handled. Table 5.1 displays the exceptions and their priority. The exception processing in the OR1200 is done in the `or1200_except` module.

The observed symptom, leading to the discovery of the incorrect implementation of exception handling priority, was the seemingly random failure of system calls

being performed in the Linux kernel software. This, of course, does not lead one to immediately check exception handling priorities, and it was through a process of elimination at every level from software application to RTL that it was determined the issue was with exception handling.

System calls are typically used in an operating system to perform operations which require permissions to be checked before operations can be performed. There are, at present, hundreds of different system calls supported by the Linux kernel and they typically implement a fundamental operation such a reading or writing data via I/O, control of a process such as starting and stopping or forking, and inter-process communication primitives. In OR1K the system call instruction, `l.sys` implements support for an operating system's system calls function. Its behavior is defined as signaling an exception causing the processor to interrupt and start executing from the exception handler address, `0xc00`. As such, the required system call handler code is arranged so its entry point is at `0xc00`. As mentioned already, this is not unique to Linux and is a common feature in other operating systems. The fact that the system call instruction typically forces a change of context is used by other operating systems, typically RTOSes, to force a context switch at a specific location in the software to allow the scheduler to execute.

The system calls in the Linux kernel on OR1K which were identified as failing were related to data I/O on peripherals. However, the first major indication or what to investigate came when it emerged that altering the rate of the tick timer correlated with the rate of system calls failing to be performed.

To test whether simultaneous tick timer and system call exceptions were being handled correctly, a simple software test was implemented, and it exposed the issue immediately. The RTL fix was implemented within minutes.

The OR1200's exception module is in charge of determining which exception should occur and when, issuing the appropriate signals to control the pipeline in the event of an exception and saving the appropriate state of the processor. Its top level block and relevant ports can be seen in figure 5.1.

The separate inputs prefixed with `sig` are generated throughout the processor. For example, `sig_itlbmiss` and `sig_immufault` are generated in the instruction MMU module. The signals `sig_int` `sig_tick` and `sig_syscall` come from the user interrupt lines, tick timer, and instruction decode stage, respectively. It is the exception unit's task to synchronise these signals and indicate when the processor's usual execution flow should be interrupted and the state of important signals saved, such as the supervision register (SR), program counter and effective address (EA) into the exception versions of those registers, ESR, EPCR and EEAR, respectively.

For a tick timer interrupt, the contents of the EPCR are specified as "Address of next not execution instruction". The observation within the OR1200 was that even though a system call instruction was in the execute stage and `sig_int` was asserted, if the `sig_tick` was asserted at any point leading up to the exception evaluation cycle, the tick timer would take precedence, and the EPCR would be set to the address of the instruction following the system call, as it would be expected that instruction would reach the write back stage and be finished as the exception

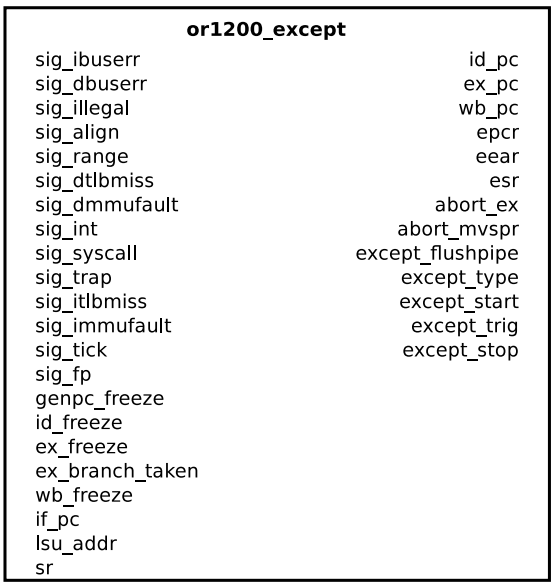


Figure 5.1. OR1200 Exception Module Block Top Level

is called.

Once it was observed that the tick timer was taking precedence over the system call instruction, in violation of the architecture specification, the interrupt priority case statement within the `or1200_except` block simply had to have its cases rearranged. This appeared to be the only incorrectly prioritised exception within the case statement. Perhaps this incorrectly arranged case statement was leftover from some exception testing of the tick timer which was never reverted. Once amended, system calls in the kernel were observed to operate correctly. The test for this, however, remains in the set of ORPSoC suite.

5.1.2 Data Cache

The data cache has received an update to first fix functionality, and next provide an alternate caching scheme.

The OR1200's data cache is direct mapped, single way with a configurable number of sets. Lines are physically tagged. The cache is bypassed if bit 31 of the address is set, or if the page mapping in the DMMU has the cache inhibit bit set. The address bit 31 bypass feature is convenient to ensure all peripherals, which are typically mapped above the `0x80000000` address, do not have their accesses cached.

See appendix A for a diagram of the top level of the OR1200's cache module.

The initial functionality issues were related to the data cache's FSM block and the relaying of accesses through to the Wishbone bus bridge. Trivial fixes were applied to ensure correct behavior with the external bus interface and the internal

Exception Type	Priority	Cause of Exception
Reset	1	Hardware reset event
ITLB Miss	2	IMMU TLB miss, reload required
Insn. Page Fault	3	Unmapped insn. location or protection violation in instruction MMU
Bus Error	4	Error signal from bus asserted due to nonexistent address, parity error, etc. during instruction fetch
Illegal Instruction	5	Illegal or unsupported instruction
Alignment	6	Access to unaligned instruction or data
DTLB Miss	7	DMMU TLB miss, reload required
Trap	7	Trap instruction executed as software breakpoint
System Call	7	System call instruction executed
Data Page Fault	8	Unmapped data location or protection violation in data MMU
Bus Error	9	Error signal from bus asserted due to nonexistent address, parity error, etc. during data access
Range	10	Arithmetic overflow
Floating Point	11	Floating point calculation exception
TT Interrupt	12	Tick timer enabled and hit programmed count limit
External Interrupt	12	External IRQ asserted

Table 5.1. OpenRISC 1000 Exception Priorities

load-store unit interface.

Enabling

Enabling the data cache improves performance in the CoreMark testbench by about 40%. On an Actel ProASIC 3 board, at 20Mhz, without data cache enabled the system receives a score of 11.97 from the CoreMark 1.0 test. Enabling the data cache improves performance to a score of 19.01. The scale of this improvement depends on the level of caching of data elsewhere in the design. On a Xilinx Virtex 5 part, with caching in the interface to the memory controller, the improvement is only from 52.05 to 66.78 CoreMarks, so only about 10% improvement. This still indicates that the data cache is an important factor in the performance of the processor module and has most impact in smaller designs with less caching elsewhere in the system.

Benchmarking

The CoreMark testbench has been developed by the Embedded Microprocessor Benchmark Consortium (EEMBC) to help provide benchmark tests which are better targeted at specific parts of embedded platforms. The scoring of the testbench is related to the number of iterations of the algorithms performed within the time, and is intended to be a single number to allow easy comparison. The CoreMark test attempts to solely target the processor's integer unit by running list sorting, matrix calculation and FSM modeling algorithms. They also develop and provide benchmarks for larger system profiling, such as MultiBench which stresses memory subsystems and operating system schedulers and synchronisation mechanisms, and various testbenches for digital entertainment platforms, networking platforms and telecoms platforms. A popular benchmark for embedded processors has been the Dhrystone benchmark - an integer version of the Whetstone benchmark which focused on floating point arithmetic. However it is no longer considered a reliable benchmark and large vendors are shying away from using it(64). One criticism of Dhrystone is that it may not accurately represent improvements of processor design because it's too susceptible to compiler optimisations. CoreMark ensures compiler optimisation is limited by passing initial values as volatile values and in a way that the compiler is unable to optimise in. The CoreMark test should give a better indication of the true performance of the processor and was considered to be a better choice of benchmark software during processor development. Porting of CoreMark to OpenRISC was performed as part of this processor development work.

Resource Usage

If added to a build of the OR1200 configured with instruction cache, timer and all hardware integer arithmetic, the data cache accounts for an additional (3266 - 2984) 9% of LUTs and (19 - 10) 9 block RAMs (for 32 kilobytes) after synthesis, and no timing penalty, on Xilinx Virtex 5 technology.

Size Enhancements

Later work involved increasing the number of words per line from 4 to 8, or 16 bytes to 32, and increasing the maximum number of sets to 1024, the maximum supported in OR1K. This improvement was also made to the instruction cache. The OR1200's Verilog HDL design relies on a single configuration file included by almost every other file in the design. In this file there are defines for things such as the optional modules, instructions and arithmetic units as well as cache configuration, among others, which define the features of the processor at synthesis time. Although it appeared as though a single value could be altered to implement a different line size, `OR1200_DCLS`, it did not immediately work. Large portions of both the instruction and data cache FSM modules required rewriting to allow a variable line size.

This has increased the configurability of the cache system, making possible an additional level of granularity. The cache can now be configured in size up to 32

kilobytes, much larger than the previous maximum of 8 kilobytes. With all 32 ways implemented, it would be possible to have a cache of 1 megabyte for each instruction and data cache. For now, 32 kilobytes is acceptable for most FPGA implementations.

Write Back

The final work on the data cache block involved implementing the option of using write-back strategy, rather than write-through. The implementation, as it was, performed every write on the bus even if it was a cache hit. This write-through strategy removes the overhead of recording dirty information about lines, and performing write-back when flushing a line. However, when the processor is storing data, it must wait for the write to occur over the bus to main memory before continuing. The ability to simply write to the cache and continue removes this bottleneck and will hopefully result in less time with the pipeline stalled. On the other hand, there is an overhead of storing a line whenever a miss occurs and it is marked as dirty. On a system with burst capability, and subject the appropriate use, write-back strategy should result in performance increases.

The additional logic overhead for write-back strategy on Xilinx Virtex 5 technology, after synthesis is (3293 - 3266) 27 LUTs or about 0.8%. The improvement in CoreMark score when using write-back is 10, (up to 98 from 88 for a 66Mhz design with both instruction at data cache of 32 kilobytes) or a 9% improvement. The performance increase due to this, relative to the logic overhead, is appreciable.

Experimental Feature

An experimental feature was implemented which looked at providing a trade-off between write-through and write-back. Named no-stack-write-through, or stack-write-back, it enforced write-back for all stores using the address in GPR1, which is defined as the stack pointer by the software's ABI. In this way, function prologues and context saves, which typically use the stack pointer, will hopefully execute faster, as the writes are not propagating through to the system bus. Another mechanism in the OR1200 exists for speeding up writes - the other is the store buffer, implemented in the `or1200_sb` module but this has remained largely untested and undocumented and it remains to be seen if data hazards are avoided when the store buffer is in use. The stack-write-back strategy hopefully reduces resource usage to achieve a similar performance boost, as the store buffer requires at least 272 bits to buffer just 4 stores. Employing the write-through strategy in the data cache, which is likely to be implemented, uses no extra memory resources and marginal logic overhead and will potentially be more efficient as lines are burst to and from memory, rather than individually written. However, a case could be made for the store buffer in an extremely low-resource implementation which optioned off the data cache.

The CoreMark result for the same design as above with stack-write-back strategy is 93.6. This indicates an improvement over standard write-through strategy (88), but only half as much as the full write-back strategy (98), this suggests the store instructions from the stack pointer constitute half the performance overhead when performing all write-through.

5.1.3 Carry Flag

The carry flag in OR1K is bit 10 of the supervision register (SR), and is used in computation by the two add-with-carry instructions, `l.addc` `l.addic`, which add two registers, or a register and an immediate, including the carry in. The carry flag is optionally implemented in the OR1200, and these instructions using it do not appear to be emitted by the compiler, and thus was never an issue during execution. There were issues identified in (68) relating to the capturing of the carry flag by the SR during execution. This pointed out the carry flag was not controlled by the pipeline's synchronous stall signals, and so may be written to the SR at the wrong time, corrupting later calculations. It also pointed out the generated carry was not present in the sensitivity list of the combinatorial result generation table, which may have caused mismatch between simulation and synthesis. These issues have been fixed, and software tests ensuring correct carry behavior have been developed and included within the ORPSoC test library. Although the add-with-carry instructions were implemented, there were issues with the way carry was calculated, and these will be discussed in that section.

5.1.4 Overflow

The overflow flag in OR1K is bit 11 of the supervision register (SR). It is specified as capable of being set by integer addition, subtraction, multiplication and division arithmetic instructions, and is used to generate a range exception. This feature of the processor was implemented as part of the work undertaken recently. It was not a critical part for system operation as the OR1K compiler port does not have the facility to make use of it, and thus compiled code runs with or without it. Despite the C compiler's inability to use it, having a proper implementation ensures those who wish to perform arithmetic operations in assembly code have this facility available to check for overflow.

The overflow flag in the SR can also be used to cause an exception. This behavior is controlled by bit 12 of the SR, or the *overflow flag exception* bit. When set, and an overflow occurs, this transfers execution to the address `0xb00`, or the range exception vector, for handling. This is most likely useful for operating systems which wish to catch arithmetic overflow.

Divide By Zero

As will be discussed in the architecture section, the flags set by OR1K's divide instructions, according to the specification, make little sense. A common exception

when using divide instructions is when the divisor is zero. As the OR1K specification indicates currently, this should set the supervision register's *carry* bit. The specification indicates that both the carry and overflow bits can be set by this instruction, however integer division cannot produce overflow of the kind seen in other integer arithmetic instructions. So it is assumed a mistake has been made in the OR1K specification, and that the carry bit is not used, and instead the overflow bit is used to indicate divide by zero. This fits in perfectly, as it's likely a divide by zero exception should cause an exception in most operating systems, which it can by causing a range exception if mapped to the overflow bit. This specification clarification will be made in upcoming reviews of the OR1k architecture specification document, but the implementation in the OR1200 has been updated, along with the implementation of the capability to perform range exceptions, to make divide by zero set the overflow bit of the supervision register.

5.1.5 Overflow and Carry Generation

The integer addition instructions on OR1K are defined as being for signed values. It is common in ISAs to provide separate signed and unsigned addition, subtraction and multiplication instructions so as to distinguish between cases of carry, occurring in unsigned numbers, and overflow, occurring in signed arithmetic. The OR1K addition, subtraction and multiplication instructions are specified as being able to set both carry and overflow. This is an issue of clarity for the architecture, most certainly, but the implementation of carry, as generated by the addition logic, considering all additions were supposedly signed, is confusing. In implementation, the addition ignores any sign and operates as a standard 32-bit adder. As signed negative values are always two's complement, this isn't an issue for the results of these additions, however it is not clear, as there is no distinction between supposedly signed and unsigned additions, which of the flags should be set, and so, as some architectures do, both carry and overflow are set as required and which flag is to be used should be left to the user. Now that the overflow logic has been implemented, this option exists. However, the addition instructions should probably be amended in the architecture specification to avoid any confusion. As the ALU implementation simply negates the second operand for subtraction and uses the same adder, and the OR1k's subtraction is also specified as signed but altering both carry and overflow flags, this discussion applies to it as well.

There are both signed and unsigned variants of integer multiplication on OR1K and both have been implemented to appropriately set the overflow flag. As discussed earlier, although integer division is listed as affecting both carry and overflow, this is assumed to be a mistake and overflow is now the only affect flag which is set on divide by zero. This has been implemented within OR1200's `or1200_mult_mac` unit, which, with the ALU results, indicate to the SPR unit when an overflow flag should be raised. A test program of the corner cases for each arithmetic operation which triggers overflow and carry has been implemented in the ORPSoC test suite.

5.1.6 Instructions

l.fl1

During development it was discovered, and also mentioned in in (68), the find-first-1 instruction was implemented but not the find-last-1, `l.fl1`, instruction. Although it was not implemented, it was not not but causing an illegal instruction exception when being executed. Instead, the find-first-1 result was generated. This has been amended and a test ensuring its operation was developed and added to the ORPSoC test suite.

Zero and Sign Extension

As discussed in the Master's dissertation by Ahmed Waqas at KTH(68), the set of value extension instructions in `l.extbs`, `l.extbz`, `l.exths`, `l.exthz`, `l.extws`, and `l.extwz`, were not implemented correctly. Not only were they missing, they appeared to execute the `l.movhi` instruction instead of causing an illegal instruction exception. This was a major error, but was not noticed often due to the compiler not being capable of emitting these instructions, although users could still assemble and use these instructions.

The implementation of the instructions was simple enough as the OR1200's decode stage and ALU is easily extended. These instructions have been added to the OR1200's code, and have been made optional via a Verilog `` `define `` in the `or1200_defines.v` file. Test software has been added to confirm their functionality.

5.1.7 Pipeline Indefinite Stall

There are indications that the OR1K ISA attempted to encode the number of cycles taken to execute integer arithmetic in their instructions. The indication lies in the OR1200's Verilog HDL defines file which contained a second ALU operation decode field for bits 9 and 8, interpreting them as the number of cycles for the integer multiplication and division instructions.

However, no processor implementation uses this, and it's unlikely they would want to. One problem is that the multiply with immediate instruction `l.muli` does not contain this field. Second is that implementations, such as the OR1200 may not take the encoded number of cycles (3, into bits 9 to 8 of the instruction) to execute the instruction, as the multiply may be a 32-cycle serial or 1-cycle parallel implementation. Third, all floating point and vector instructions, which quite likely require multiple cycles to execute, do not contain this field. For these reasons, and for additions to the processor implementation that require it, a the processor's pipeline has had a mechanism added which allows it to be stalled for arbitrary lengths of time.

The implementation required the addition of a signal generated during instruction decode in the `or1200_ctrl` module. This signal, `wait_on`, used in the `or1200_freeze` module, now selects the appropriate signal to be included in the stall logic, keeping

the pipeline from progressing until the appropriate module signals it has finished its operation.

At present, this is used for FPU instructions which have a variety of cycle lengths, and when causing a cache flush from the data cache with write-back strategy as it will take a variable number of cycles to perform the bus accesses require to write out a dirty line. An alternative solution to this is to use the `l.msync` instruction to ensure all memory operations are complete before complete before continuing, which should also rely on the same mechanism, but currently does not need to.

Another benefit of this approach is to simplify the work required to attach other computation units which may require the processor to remain stalled until its computation is complete. Examples of this may be a vector processing unit, or an external, custom computation unit exercised by the OR1K's custom instructions.

5.1.8 Multiply, Divide and MAC unit

One issue that arose when selecting which optional integer arithmetic operations to support, via the OR1200's defines file `or1200_defines.v`, was that integer divide could not be included without the inclusion of support for the multiply-accumulate (MAC) instructions.

Upon inspection of the RTL implementation within the multiply and MAC unit, `or1200_mult_mac`, it was clear that this constraint was not necessary. Some rearranging of Verilog HDL `` `ifdef `` statements meant that requirement was no longer in place.

The inclusion of the MAC functionality does require that the multiply feature be enabled, but forcing it to be enable to allow integer divide added an additional 64-bit adder/subtractor required for the accumulate or subtract stage of the `l.mac` and `l.msb` instructions, respectively. As these instructions are useful only in DSP applications, and integer divide is useful in any application, it was an unnecessary resources overhead that has now been removed, allowing greater flexibility.

This unit, too, was largely re-written to aid clarity and the addition of the serial multiply unit which is outlined in the next section.

5.1.9 Serial Multiplier

To aid in achieving a middle ground between a parallel multiplier implementation, and relying on software libraries to implement multiplication, an optional serial hardware multiplier was added to the OR1200. The main motivation arose when synthesizing the OR1200 for the Actel ProASIC technology, which does not contain any arithmetic macros such as multipliers, and finding area use was too high. The full parallel multiplier, then, took up 6% of an A3PE1500, nominally 1.5 million "gates", percent of the core cells, and this was not considered reasonable resource usage.

The "parallel" multiplier implementation in the OR1200 is a multiplication operator with two 32-bit operands and 64-bit result vector. There are two registering

stages of the 64-bit result after this to provide some flexibility to the synthesis tool regarding its implementation.

A scaling accumulator design was chosen for its simplicity to describe and relatively small amount of resources required to implement - mostly an adder and two shift registers of the length of the multiplicands. This process takes 32 cycles for two 32-bit words.

The difference in resource usage in Actel technology is presented in table 5.2. The approximately two-thirds reduction in logic use does, of course come with the added latency of performing the multiplication, but timing analysis also indicates its critical path is longer, as the serial device had an estimated maximum frequency of 70Mhz, while the parallel implementation could, supposedly, be clocked at 270Mhz.

The advantage of this implementation is diminished on FPGA technologies such as Xilinx, where the difference in implementation size is marginal - both use approximately 200 LUTs, with the serial implementation using 72 FFs and the parallel 100. The parallel implementation, however, makes use of 3 Xilinx DSP48E macro cells, containing various hardware arithmetic circuits, no doubt being used to attempt to calculate the multiplication as efficiently as possible. Again, the speed benefit appears to be the with the full multiplier implementation, which XST claims can reach 750Mhz, compared to the serial's 250Mhz. Whether these values remain even remotely the same after backend processing remains to be seen, but they at least give an indication of the length of the critical paths after synthesis, which appears to be longer with the serial implementation.

In all, this helps provide a area-saving option for designers who might be using older FPGA technologies, or alternate ones such as Actel's, which may not contain hardware arithmetic macro cells to help implement operations such as multiply. Despite the multiply instructions being class I in ORBIS32, the compiler and OR1200 allows for hardware multiply support to be removed, and instead replaced with software routines to perform multiplication. Despite multiplication taking approximately ten times longer with serial divide (three cycles for usual multiplication versus thirty two for scaling accumulator), this is still much less than the overhead of storing, fetching, and executing the software multiplication routines required to achieve the same result.

5.1.10 Single Precision FPU

The OR1K architecture defines two sets of instructions to support floating point arithmetic, the ORFPX32 and ORFPX64 for single and double precision operands, respectively. The ORFPX32 instruction set is presented in table 5.3. The ORFPX64 instruction set is equivalent to the ORFPX32 instruction set except that it operates with double precision, and thus requires a 64-bit register file to operate. The goal of this instruction set is to allow support for the IEEE 754 standard for floating point computation on OR1K.

The architecture has a FPU control and status register (FPCSR) mapped in SPR group 0, number 20. It allows the ability to enable one of four rounding

Serial				Parallel			
Cell	Count	Cell	Count	Cell	Count	Cell	Count
AND2	9	NOR2B	59	AND2	16	NOR3A	38
AND3	72	NOR3	39	AND3	72	NOR3B	94
AO1	45	NOR3A	13	AO1	31	NOR3C	8
AO13	4	NOR3B	30	AO13	30	OA1	11
AO18	2	NOR3C	12	AO18	8	OA1A	5
AO1A	13	OA1	11	AO1A	6	OA1B	32
AO1B	6	OA1A	5	AO1C	2	OAI1	1
AO1C	3	OA1C	3	AO1D	1	OR2	21
AO1D	3	OAI1	2	AOI1	5	OR2A	34
AOI1	4	OR2	18	AOI1B	10	OR2B	45
AOI1B	9	OR2A	12	AX1	4	OR3	6
AX1	12	OR2B	19	AX1A	2	OR3A	9
AX1A	3	OR3	2	AX1B	1	OR3B	24
AX1B	1	OR3C	6	AX1D	3	OR3C	15
AX1C	2	XNOR2	92	AX1E	14	XA1	3
AX1D	8	XNOR3	6	MAJ3	195	XA1A	2
AX1E	3	XO1A	1	MIN3	188	XA1B	6
MAJ3	9	XOR2	23	MX2	55	XA1C	1
MIN3	6	XOR3	4	MX2C	39	XNOR2	112
MX2	70			NAND2	5	XNOR3	92
MX2C	88			NOR2	48	XO1	2
NAND2	1			NOR2A	164	XO1A	4
NOR2	70			NOR2B	304	XOR2	142
NOR2A	78			NOR3	41		

	FFs	Count		FFs	Count
	DFN1C1	6		DFN1C1	98
	DFN1E0P1	1		DFN1P1	1
	DFN1E1C1	64			
	DFN1P1	1			

	Total			Total	
	Count	Device%		Count	Device%
	954	2		2352	6

Table 5.2. OpenRISC 1200 serial versus parallel multiplier synthesis results in Actel ProASIC3 technology

Mnemonic	Function	Mnemonic	Function
lf.add.s	rD[31:0] < - rA[31:0] + rB[31:0]	lf.cust1.s	Custom instruction
lf.div.s	rD[31:0] < - rA[31:0] / rB[31:0]	lf.ftoi.s	Convert float to int
lf.itof.s	Convert int to float	lf.madd.s	Multiply accumulate
lf.mul.s	rD[31:0] < - rA[31:0] * rB[31:0]	lf.rem.s	rD[31:0] < - rA[31:0] % rB[31:0]
lf.sfeq.s	SR[F] < - rA[31:0] == rB[31:0]	lf.sfge.s	SR[F] < - rA[31:0] >= rB[31:0]
lf.sfgt.s	SR[F] < - rA[31:0] > rB[31:0]	lf.sfle.s	SR[F] < - rA[31:0] =< rB[31:0]
lf.sflt.s	SR[F] < - rA[31:0] < rB[31:0]	lf.sfne.s	SR[F] < - rA[31:0] != rB[31:0]
lf.sub.s	rD[31:0] < - rA[31:0] - rB[31:0]		

Table 5.3. OpenRISC 1000 ORFPX32 Instructions

modes, and signal all standard exceptions raised according to IEEE 754. The only unsupported instruction in OR1K required by IEEE 754 is the square root function.

Motivation

It was identified during some profiling of video encoding software that extensive use of floating point types caused significant amounts of time to be spent in software functions performing floating point operations. As the OR1200's ALU and decode logic is relatively simple to expand, it was deemed a worthwhile exercise to look at adding a floating point unit to the processor implementation. As the OR1K's floating point instructions use the same register file as integer arithmetic instructions, and the OR1200 is a 32-bit implementation, with no native support for 64-bit data, the implementation was limited to single precision support. It has recently been observed, though, that the OR1K compiler handles 64-bit values in such a way that adjacent register pairs are used for double precision values and this could easily be used to implement a double precision FPU. This was not known at the time, and as will be discussed in the architecture analysis section, the OR1K instruction set does not contain the right instructions to implement double precision as efficiently as possible.

Implementation

The first implemented solution was to provide a generic port from the OR1200's CPU to attach any floating point unit. The additional infrastructure required to handle floating point operations had to be added, too. This included additional logic in the exception unit to provide the ability to cause floating point exceptions when required, implementation of the floating point status and control SPR (FPCSR), instruction decode and operation forwarded to FPU wrapper, addition of MUX input to register file writeback stage, and the FPU wrapper itself which contained logic to set flags in the FPCSR based on the operation and result, and generate comparison results to set the flag for the `lf.sfXX` series of instructions. All

floating point operation can be excluded by the removal of a Verilog HDL `‘define` at synthesis time.

There are several floating point arithmetic cores available on OpenCores.org, and the one initially chosen was Rudolf Usselmann's, named simply *fpu*. It is in Verilog HDL and provides all required operations except for the remainder function. It was designed to be an IEEE 754 compliant FPU, and also claimed to be compatible with the OR1200 core, even though no implementations of this existed. The arithmetic unit in Usselmann's FPU is separate from the comparison unit, and as such, the OR1200's FPU wrapper instantiated it along side the arithmetic core.

Once instantiated the floating point operation was checked via some test software which existed previously in the or1ksim testsuite. The OR1K compiler port, then was tested to confirm basic FPU operations were working. However, upon checking with some basic C code using floating point operations, it was noticed that the OR1K compiler generated many double precision instructions, even though the `float` type was used for the variables. This upgrading of precision occurs commonly with floating point arithmetic in the C compiler, and so the OR1K GCC port was modified to require an option to be passed at compile time if double precision instructions were to be generated. This involved modifying GCC's OR1K machine description, and adding a new option in the `or32.opt` file, requiring the machine option `-mdouble-float` to be passed for it to emit double precision opcodes. Once in place, this allowed the GCC machine option `-mhard-float` to be passed at compile time, and all single precision calculations would be performed with an instruction instead of a software routine.

This implementation worked, but Usselmann's implementation could not be synthesized directly as it relied upon the the arithmetic primitives being implemented as the user chooses. These primitives which required implementation are multiply, divide and modulo-divide for the remainder. So, despite the core functioning in simulation, it could not be synthesized as no synthesis tools support integer division with a divisor that isn't a constant, base-2 number (so that it could be simplified to a shifter.) At this stage the infrastructure was complete, but unfortunately the design required implementations of these arithmetic primitives.

An alternative core was then evaluated. The OpenCores *fpu100* core by Jidan Al-Eryani was chosen as it claimed to be IEEE 754 compliant, and appeared to have all of its required arithmetic blocks in synthesizable form. Although it took significantly longer to perform FPU operations then Usselmann's core, 35 cycles for multiply or divide and 7 for add or subtract(69) compared to just 3 for all operations in the *fpu* core, it was synthesizable and made use of serial multiply and divide circuits so was area efficient.

The *fpu100* core was attached to the OR1200 and tested to confirm its basic functionality, and that it was synthesizable. Once this was confirmed, and its operation found to be satisfactory it was decided that this core was to be used for the single-precision floating point arithmetic in the OR1200. Comparison logic could still be done via Usselmann's *fpu* project's `fcmp` module, however the arithmetic unit would come from the *fpu100* project.

This presented a problem, however, in that the *fpu100* core was in VHDL, and the OR1200 is in Verilog HDL, and all of the OR1200's testbench and testing tools only supported Verilog HDL. After several VHDL-to-Verilog tools were evaluated and found to be lacking for various reasons, the *fpu100* core was then transposed by hand from VHDL to Verilog HDL. This code has now been included in the OR1200's source for distribution.

or1ksim implementation

The floating point test program included with or1ksim was found to test some, but not all, corner cases designed to exercise the floating point exception logic. A more thorough set of stimulus was needed to ensure the operation of the core. A well written and thorough test package is provided in John Hauser's *SoftFloat*. This software, written in ISO/ANSI C, has two independently written floating point unit emulation algorithms and a test vector generator. These two implementations of an FPU, written entirely differently, help ensure the function is accurate. The alternate, less efficient, FPU computation algorithm is included in the TestFloat application, and is called SlowFloat, and the more efficient FPU emulation is implemented in the SoftFloat library. SoftFloat can be used to simply simulate floating point operations, but is also used against TestFloat. In this way, SoftFloat can be tested against TestFloat, and any issues during compilation which cause incorrect behavior by one of the FPU calculation libraries is bound to be picked up by the other. To test an architecture's floating point capability, architecture-specific code is added to TestFloat, which is then compiled and executed.

Adding architecture-specific components to TestFloat can be achieved by simply filling in skeleton functions called to perform standard floating point operations. There also needs to be a way for TestFloat to set and test the architecture's floating point flags.

TestFloat's test vector generation software can perform hundreds of thousands of tests, checking exception state and result for each one. This was considered to be a suitable test, then, for the newly implemented OR1200 FPU. It was initially run against or1ksim to check the TestFloat software was ported correctly. This, however, showed up problems with or1ksim's single-precision floating point emulation. The actual arithmetic and the behavior of the floating point flags varies between host platforms or1ksim can run on, and for accurate emulation and access to floating point flags would require architecture-dependent assembly language inlines in C, all of which was considered too much. The obvious alternative was to implement the SoftFloat library to emulate single precision floating point in a host architecture-independent way. The library was then implemented in or1ksim, and the floating point instructions in or1ksim result in calls to SoftFloat's emulation functions. Running TestFloat, ported for OR1K, on or1ksim, is then a more elaborate way of running TestFloat with SoftFloat, but it ensures all architecture-specific parts of the port are functioning correctly, and ready to test on the hardware. A mismatch in behavior is still possible, however, as running TestFloat on or1ksim

Operation	Software			Hardware
	Instructions	Icache misses	Dcache misses	FPU cycles
Add	177	11	8	10
Subtract	151	9	3	10
Multiply	206	32	3	38
Divide	401	20	4	37
Int to Float	190	24	4	7
Float to Int	93	18	4	7

Table 5.4. OR1200 hard versus soft single precision floating point execution

using SoftFloat as its floating point emulation library could potentially have a bug, as the SoftFloat library is compiled with the host machine's compiler, in this case GCC for x86, and TestFloat is compiled with GCC for OR1K. To confirm there are no issues with OR1K's compiler and it is compiling the TestFloat library correctly, TestFloat testing against SoftFloat was compiled and executed in or1ksim and on FPGA target to confirm TestFloat's SlowFloat library is compiled correctly on OR1K. With these tests done, the degree of confidence in TestFloat and SlowFloat to test the OR1K's FPU is very high.

Testing

Finally, with TestFloat testing the hardware implementation against SlowFloat, some bugs in the exception handling wrapped around the FPU modules, and VHDL-to-Verilog transposition bugs were found, but none were discovered in the FPU100 arithmetic core, or in Usselmann's fcmp comparison module, either. Extensive testing of the FPU on Actel FPGA target was done.

5.1.11 Results

Execution speedup is obviously going to be significant. The overhead required by the software floating point calculation libraries is outlined in table 5.4. The exact number of cycles varies depending on the memory subsystem to fetch the code and process the data. But for a system with 4KB of instruction an data cache, with 16-byte (4 word) lines, the performance difference can be observed in table 5.4.

The synthesis results of the entire single precision FPU indicate that is quite large. In the Actel ProASIC3 FPGA technology, the unit takes about 8400 combinatorial gates and 1500 flip-flops, which is about 25% of a "1.5 million-gate" Actel A3PE1500 FPGA, and has a maximum frequency of 35MHz. On Xilinx's Virtex5 FPGA technology, the number of LUTs required is about 3400 and 1200 flip-flops and has a maximum frequency of about 100MHz, which is in line with the claims of the FPU100 project's developer. Although this is a very large module, it can provide significant execution speedups for floating point arithmetic.

5.2 Architectural Simulator

Similar to the OR1200, the OR1K architectural simulator, *or1ksim*, saw few if any updates posted on OpenCores between 2005 and 2008.

Since 2008, however, or1ksim has been the recipient of a large amount of work from a contributor using the OR1K platform to demonstrate the services of their consultancy. Jeremy Bennett of Embecosm in the UK has lead the contributions to or1ksim and implemented many new features, as well as giving the entire source code a touch up. or1ksim is increasingly useful tool for early code development and analysis for designs targeting the OpenRISC platform.

The significant portions of Embecosm's work on the simulator (and other OpenRISC related topics) are neatly and concisely presented in a series of application notes available on the company's website(59). Thanks largely to this work, or1ksim has seen two point releases since 2008.

In all, or1ksim has probably seen the most work of any model from the collective of contributors, and can now be seen as a very reliable model of OR1K. Outlined below are the significant bug fixes and feature improvements made on or1ksim.

5.2.1 Testsuite

The testsuite code included with the project was the most extensive of any code in the OpenRISC project, and shows the initial development of the simulator was attempting to be thorough. The code was restructured so as to provide tests of or1ksim itself, and tests of or1ksim as a library. Next, a test harness was added to enable testing with DejaGNU. This system is highly automated and provides the capability of quickly and easily performing regression testing, as well as making the addition of new tests relatively simple.

Platform Support

Contributors have been maintaining the simulator so it can be built on non-Linux POSIX-compliant systems, such as Mac OS X and Cygnus Windows. Not only can the simulator be built and run on these platforms but there has been efforts towards ensuring the regression testing suite using DejaGNU also functions correctly on different platforms.

5.2.2 Debug Interface

As will be discussed in a later section, a GDB stub, supporting RSP, to provide access to the debug interface was added. It provides GDB with a standard interface to the system, and the implementation is very portable across the various OpenRISC simulated models. As will be discussed later, essentially the same code that is in or1ksim was used in the cycle accurate model, and attached to RTL simulators via the Verilog procedural interface (VPI) to implement the same accessibility by GDB to OR1K systems. The RSP server, in this case, talks directly to the emulated

debug interface. As will be discussed later, in the models with less abstraction, the JTAG lines are driven directly by the GDB stub and supporting interface modules.

5.2.3 Floating Point Emulation

As discussed in the previous section, emulation of the ORFPX32 and ORFPX64 instruction sets was enabled by the use of John Hauser's *SoftFloat* floating point emulation library. The issues surrounding providing accurate floating point system emulation on any host system were presented in the previous section, but the use of this emulation library gets around those issues, and while there may be a performance penalty compared with executing floating point instructions natively, the inherent portability of the solution between host platforms is a worthwhile benefit.

The *TestFloat* library was added to or1ksim's testsuite to check all of the supporting infrastructure around the FPU in the or1ksim model.

5.2.4 Instructions and Flags

As discovered during the development of a thorough verification suite for the OpenRISC platform in (68), several instructions were not functioning correctly. As very few of the assembly language tests in or1ksim's testsuite were using these instructions, and the compiler did not emit them, these bugs were not picked up earlier.

The issues were with the `l.divu` instruction, as per bug 1770 on the OpenRISC bugtracker, carry and overflow behavior for addition and division instructions, implementation of the find first and last '1' instructions, the multiply and MAC instructions, alignment exception on jump instructions, and the rotate instructions were all corrected and the appropriate tests have been added to the testsuite. For further details on these bugs, see (68).

5.2.5 Peripherals

Several of or1ksim's peripheral models have received updates and fixes. Most notably, the Ethernet MAC model can now implement actual network connectivity from within the simulator.

Ethernet

As virtualisation of whole machines on PCs has become more commonplace, so too has the ability to implement virtual network interfaces to provide network accessibility to VMs. A method of doing this was investigated and the OpenCores 10/100 Ethernet MAC peripheral model in or1ksim was updated to be able to use a virtual network interface, and in doing so, can provide actual networking functionality to software on or1ksim.

This was primarily developed to assist with the OpenRISC Linux kernel port testing, as it's desirable to provide network access to access via NFS to large stores of files that cannot, practically, be embedded into a kernel boot image. It is a

feature that is common, as mentioned, in large virtual machine suites, and is how the QEMU simulator (discussed later), implements its network support.

The required network virtualisation feature is currently only supported under GNU/Linux operating systems. Commonly referred to as *TUN/TAP* devices, they provide software-emulated network interfaces operating at significantly lower levels of abstraction than is available via the standard sockets networking API. A *TAP* device supports OSI layer 2 information, such as Ethernet frames, to be processed and passed to the interface, and a *TUN* device supports one level higher, layer 3, such as IP packets. TAPs are typically used for network bridging and virtual interfaces, and TUNs are used for routing.

In the or1ksim Ethernet peripheral's case, a TAP is what was required, as the model is capable of generating Ethernet frames. The peripheral's code was entirely reorganised and had large sections reimplemented to provide TAP support. It had previously only supported file I/O. It was a new and exciting challenge for those involved as the result was a tangible and useful feature for the simulator. Several attempts were made at implementing the model, with unexpected interrupt behaviors being exhibited from time to time, but in the end the peripheral model proved well done as it now implements solid networking support for the Linux kernel running on or1ksim. This platform is being used by Embecosm to run multiple instantiations of or1ksim and the Linux kernel port which is being used to perform userspace software library regression test suites.

5.3 Toolchain

The OR1K port of the GNU toolchain, consisting of *binutils*, *GCC*, and *GDB*, was included in the project from as early as the 2000. These ports, although operational to an extent, were far from perfect and implemented just the essentials for a functioning C compiler. Despite the port being accepted into the mainline GNU repositories early on, it has suffered from no continued maintenance for many years.

From 2006-08 contributor Rich D'Addio, who's own site hosts a set of OpenRISC toolchain related work(60), released work updating the port to support the more recent releases of *binutils*, versions 2.16 and 2.18, and GCC versions 3.4.4 and 4.2.2. This was the main source for the most updated release of the toolchain. This was somewhat of a stop-gap as the OpenRISC project on OpenCores was in flux while the new owners, ORSoC, took over and reorganised the OpenCores.org site.

Unfortunately no full time contributors for *binutils* or GCC stepped forward after 2007, and again the development of the ports stagnated. As has been noted, toolchain problems were one of the major detraction of the architecture, and the lack of maintainers for an extended period did not help.

By twenty ten, commercial interest in the platform resulted in financial support for a complete renewal of the toolchain. The GNU toolchain components were improved and the port bought into line with the latest development versions of the tools.

The work done on the tool chain port has allowed the developers to gain a better understanding of the architecture and how features are implemented in the various tools. The issues which still remain in the tool chain hopefully have their days numbered as better-skilled contributors are able to weed them out.

5.3.1 binutils

Improvements to the *binutils* suite, which was updated to version 2.20.1, include the adoption of the RELA linking format, and various fixes improving consistency with the architectural specification. The assembler and linker testsuites were also made functional, and tests were added. However, there still remains some doubts about the binutils port which will be discussed in later sections.

5.3.2 GCC

The following is a list of improvements made to the GCC port.

- Fixes providing compatibility with GCC version 4.5.2
- C++ language enabled
- C and C++ regression suites enabled and all GCC-port related errors fixed
- ABI updates according to architectural specification
- Various optimizations enabled
- Data alignment issues resolved
- Corrected assembly initialisation sequences
- Integration with OR1K *newlib* port
- Addition of flag to specify double or only single-precision floating point instructions
- Identification of issue with floating point to integer conversion

5.3.3 GDB

Work on the GDB port was related to the implementation of a wrapper for or1ksim to be used as a simulation target, and porting gdbserver for OR1K Linux to allow debugging of Linux userspace programs over a network connection.

5.4 Software

The software available for the OR1K platform had been limited to a subset of the standard embedded operating systems, libraries, and statically linked applications. The initial OR1K development effort included ports of the embedded system-targeted C libraries *newlib* and *uClibc*, and operating systems *eCos*, *RTEMS*, as well as a Linux kernel port. Each of these was left in varying states of functionality in 2005.

Development since the original team's hand over to ORSoC was minimal, with maintenance performed only when the software stopped compiling. No significant amounts of development were undertaken and released publicly until 2009.

Contributor Rich D'Addio distributed patches keeping the uClibc port functional, and an initial port of the newlib library, by Jacob Bower of Imperial College, London, was significantly re-worked and bought up to date by Jeremy Bennett of Embecosm. Work on the toolchain progressed somewhat under contributions by the core community of developers. It was not until the significant reworking of the toolchain that *uClibc* and the the Linux kernel port were greatly improved.

The work on the OpenRISC ports of uClibc and Linux kernel has enabled greater functionality of the operating system. This has provided access to many of Linux's desirable features, such as its networking and file system capabilities. These features have been tested heavily by recent work, too. The OR1K kernel port's development is now keeping up to date with the mainline kernel development tree.

5.4.1 newlib

The OpenRISC port of newlib now implements a more complete support layer for the library. Significant work was made to ensure the port was thoroughly documented, ensuring it should be simple to customise for further implementation. Any programs requiring compilation and execution in the GNU tool regression test suites use newlib as their C library, and the underlying "board" support components of *libgloss*.

libgloss

Libgloss is a part of newlib, and implements the lowest level board and architecture-specific parts of the C library. This includes C runtime initialisation code, and system call implementations, that are usually not only architecture specific, but board-specific, too. Work towards implementing a simple API for the basic OR1K architectural features, such as timers and cache control, has begun and is likely to be implemented very soon. Work on implementing board support that is configurable at compile time is also largely finished and awaiting review by the contributors. This enables an easy way to pass parameters to configure the otherwise generic sections of libgloss at compile time, or, put simply, allows many different boards to be supported and selected at compile times.

At present, the libgloss system calls provide no operating system support, and are largely unimplemented, however the goal of the newlib and libgloss libraries is to provide for software to run on the *bare-metal*, that is without any underlying operating system layer. The idea behind a compiler providing bare metal support is that it allows users in the process of developing OpenRISC systems to quickly compile simple programs ad have them run on the board. By providing access to a basic C library and UART I/O, which can help test the basics of the system through hello-world type applications and other custom diagnostics C programs the developer can quickly determine if it is functioning as expected. It is anticipated all boards builds included in ORPSoC will have their boards also supported in libgloss, which provides a quick and easy way for users of the board ports to compile and run code on the system to check things are operational.

5.4.2 uClibc

The *uClibc* library is used to implement Linux operating system compatibility for applications. The architecture port sections implement the *glue* layer between applications and operating system. The major improvements have been reducing the complexity of the port by adopting as many generic features as possible, synchronising with interface changes in the kernel, and the addition of support for threading with *linuxthreads*. The most recent changes have seen the system call interface improved to become far more efficient.

5.4.3 Linux Kernel

Significant work has gone into bringing the kernel port back to life after approximately four years of inactivity of the port. It has had almost all architecture-specific sections either wholly, or largely, reimplemented. The dated nature of the port, as it was, meant large swathes of it, which are now implemented in architecture independent code, could be cleared away. Support for new kernel features such as device trees for configuration, and dynamic ticks have been added. The simpler *linuxthreads* threading support has been implemented. The architecture-specific signal handling and debugging mechanisms in the port have also been updated. As mentioned previously in the uClibc work, the system call interface has been re-written and is now more efficient.

5.5 Testing

Testing of each implementation, toolchain component and library has received improvements, particularly the GNU tools. The newly enabled GNU GCC regression test suite is considered a significant development in terms of the amount of testing the OR1K port, and implementations, now receive.

5.5.1 RTL Model

The current OR1200 is a rather monolithic implementation, making RTL model block-level testing difficult. A block-level approach to testing was never taken by the original designers, and has not yet been implemented. The primary testbench is the ORPSoC reference implementation, a minimal system-on-chip consisting of the OR1200, bus system, memory and debug interface. The OR1200 is tested by running the system from reset with different programs in the main memory. A monitor module checks the processor and bus state, halting upon detection of errors. The test suite then consists of the different individual software programs, written in a mixture of C and assembly language. Each software test program exercises a particular feature of the processor, and finishes with reporting a value indicating whether the test was successful.

The OR1200 software tests included in ORPSoC have been improved to increase the coverage of the processor's features that are tested. Specific tests for the newly upgraded MAC, multiply, divide and floating point units have been added, and tests tests for the data cache, MMU and exception handling have been improved.

The testing done by Waqas Ahmed, presented in his Master's dissertation(68) discovered some instruction errors with the implementation. The developed testbench, which took a grey-box verification approach, and provided constrained random stimuli, implemented a System Verilog wrapper for the OR1200, and paired the OR1200 with the golden model, or1ksim, to compare internal registers, among other things, after each executed instruction. The top level, tying it all together, has been done according to the Open Verification Methodology, OVM, and should be portable among compatible simulation environments. This approach discovered irregular behavior of some instructions and status flags. The testbench is now available with the OpenRISC project on OpenCores.org. This testing did not, however, go on to check the exception behavior, and that the execution flow is according to specification. However it is a valuable piece of work and the project, as a whole, has benefited from it.

GCC Regression Suite with Verilator-built Model

The newly enabled GNU GCC regression test suite also provides a large new set of software tests to execute against the RTL model. However, running these many thousands of tests on the RTL model in the open source Verilog simulator, Icarus Verilog, proves to be a rather time intensive method of testing.

The simulation rate of the open source Verilog HDL event-driven simulator, Icarus Verilog, is far from that of the commerical implementations. One method of increasing simulation rate of the RTL model is to use the Verilator tool to generate a cycle-accurate model of the RTL instead. This involves running the OR1200 Verilog HDL code through the Verilator tool, which generates a cycle-accurate C++ model, capable of running simulated systems in the hundreds of kilohertz range, as opposed to the sub-ten kilohertz range simulation experienced with Icarus Verilog.

This dramatically reduces simulation time, at the expense of sub-clock cycle timing accuracy. This has enabled the capability to run testbenches orders of magnitude greater, like the GCC regression suite, within a reasonable time frame.

This method of testing the OR1200 is a high level approach, essentially testing every possible combination of instructions the compiler is likely to generate. For the intended use of the OR1200, as a programmed core in a SoC, this is suitable as it is exercising the core in its most likely use case. If the ORPSoC test suite and GCC regression test suite programs all execute correctly on the RTL model then it is a good indication that the model is correct and likely to function as desired in other implementations. The lack of a formal verification suite for the OR1200 RTL description will be explored in the critique section. However for the primary intended uses of the OR1200 - as a processor in a SoC system - this method of testing verifies functionality in that capacity.

In one run of the GNU GCC testsuite for the C compiler of approximately 53,000 tests, it completed in five hours and twenty minutes, having simulated just under five billion cycles. Compared with the architectural simulator, in terms of speed, this turned out to be less than triple the amount of time for the same testsuite, mainly due to the fact that execution time is only about fifteen percent of the whole time, with compilation and linking and test harness overhead accounting for the rest of the time(70). This test suite did report some errors on the cycle-accurate model that were not present on the architectural simulator, and these reasons are still being investigated.

5.5.2 Architectural Simulator

The architectural simulator, or1ksim, had a testsuite of C and assembly code developed to test the processor model, and the various peripherals, while it was being implemented. The recent work on the simulator testing included adding scripts to allow use of the DejaGNU testing system to run the regression testing, addressing any test failures, and adding new tests as new features were added to or1ksim.

The GNU GCC regression testing suite was initially run against this simulator, and it is used as the golden reference model for execution behavior by the RTL implementation. With this in mind it is important to ensure the correct behavior of the simulator's modeling of an OR1K-compliant processor. For this, a set of tests written in assembly ensure the processor behaves according to specification. This regression test suite is continually expanding as bugs or missing features are fixed and implemented, respectively.

At present or1ksim's testsuite consists of 2174 tests of basic OR1K operation and peripheral features, and 262 tests of or1ksim as a library and its interface.

5.5.3 Toolchain

The standard regression tests for each tool have been enabled to run against the OR1K toolchain by recent work. This exposed many problems with the toolchain

and enabled them to be fixed. As previously mentioned, this test suite is run against both or1ksim (the golden reference) and the the RTL model.

Binutils

The binutils test suite, combined, contains 337 test, of which all but 16 linker tests pass or are not applicable to the OpenRISC port. The missing sections of the OR1K binutils port will be discussed in the future work section.

GCC C

The GCC C code compiler test consists of 53768 tests, 52869 of which pass, and the remaining 809 are not applicable to the OpenRISC platform or fail due to causes external to the OR1K port.

GCC C++

The GCC C++ code compiler tests consist of 21111 tests, 20721 of which pass, and the remaining 390 are not applicable to the OpenRISC platform or fail due to causes external to the OR1K port, or optional functionality not being supported.

Failures

The only true failures of the C and C++ compilers are related to the linker not supporting the garbage collection mechanism (the option `--gc-sections`) and thus are not really a failing of the compilers at all, rather the linker. This indicates the status of the OR1K GCC port is that of an industry grade compiler. This is a great advancement in the quality of the compiler.

GDB

The GDB test suite consists of 11934 tests, 11758 of which perform as expected, 42 of which fail due to a recent switch in debugging format used by GCC, 19 untested testcases, and 115 unsupported tests. This indicates the debugger, too, is in a quite robust state.

5.5.4 Software Libraries

The newlib regression suite is quite small and consists of just 24 tests, 23 of which pass and one of which fails. However all of the above tests were compiled using the newlib library so each program could execute on the "bare metal" of the simulated models, indicating the newlib library provides the basics well.

At the time of writing the infrastructure required to run the uClibc library's test suite was in the state of being assembled.

5.6 Accessibility

This section discusses the work that has helped make the OpenRISC platform more accessible and easier to use. It will outline the standard debug interface now present on every model, and the way ORPSoC provides an easy-to-use system for implementing OpenRISC-based systems.

5.6.1 Debug Interfaces

A standard GDB stub with RSP interface was developed by contributor Jeremy Bennett and included in all of the OR1k implementations as a way to provide a standardised interface to each of the systems.

GDB and RSP

GDB, or the GNU Debugger, is a widely used software debug tool developed as part of the GNU Project. It provides the capability to investigate and control an executing program, the hardware it runs on and almost any other part of the system that is memory mapped. It has access to memory and in this way can be used to load programs onto a *target*, or the system it *connects* to, to debug. When running natively, *connecting* involves the operating system giving GDB control of the software process that is being debugged. In embedded debugging, connecting usually means connecting via some channel to issue commands such as read and write to system registers and memory.

The protocol used when connecting over network sockets is the remote serial protocol. It is a simple packet-based protocol consisting of ASCII character control words and raw data payloads. Almost all of GDB's functionality can be extended over RSP to control remote targets such as FPGA boards or simulators. Implementing a system to provide GDB access and control is relatively straight forward and requires just the parsing of commands and data from incoming network sockets packets.

An RSP *server* provides GDB with the ability to connect via network sockets and should support the necessary subset of RSP commands to give GDB control of the target system. In the OR1K's case, most of the capability of GDB is limited to memory accesses, CPU register access, software breakpoints, single stepping and a stall or unstall command.

Figure 5.2 shows the various targets and their mechanisms for connecting to the OpenRISC systems. The common element between them is the RSP server, which is largely the same code between or1ksim, the cycle accurate model, the Verilog simulator VPI functions, and the USB debug proxy,*or_debug_proxy*.

JTAG

In all targets except for or1ksim, the system's primary interface is via JTAG. The debug interface module within the design is on the JTAG scan chain side the device.

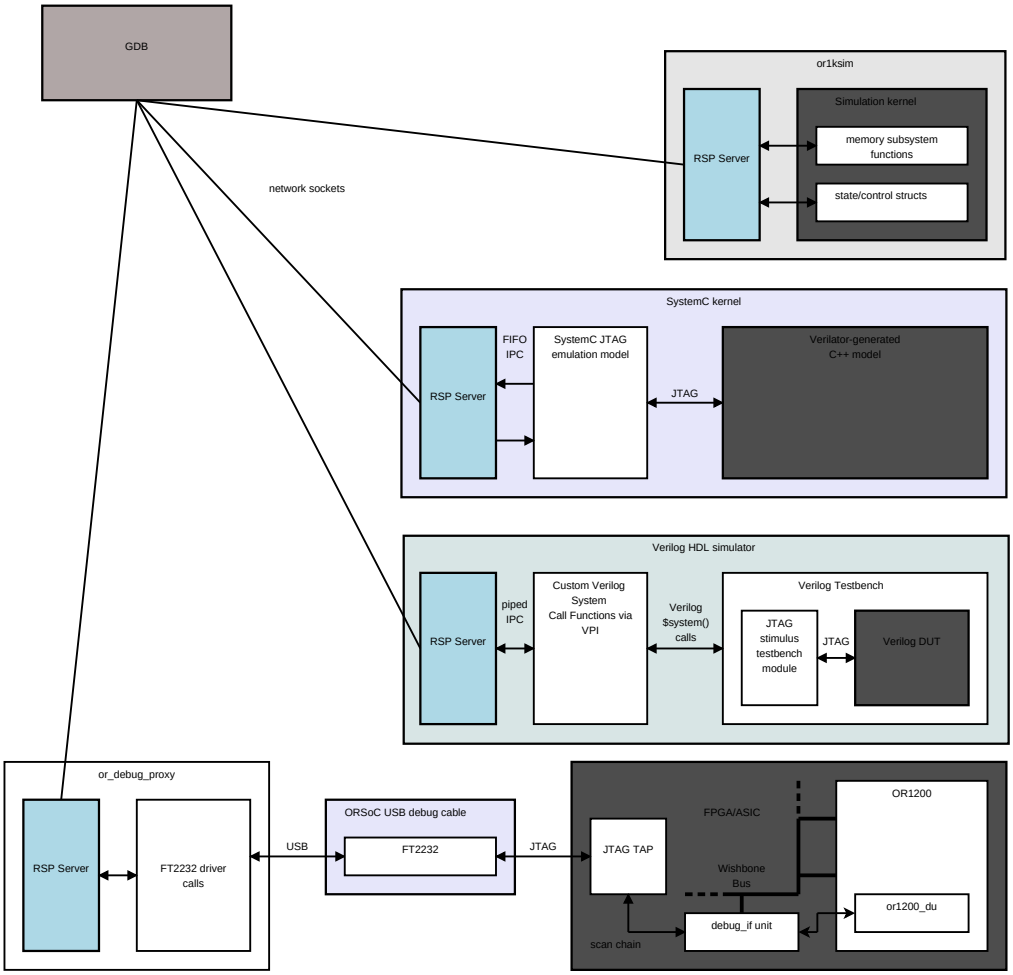


Figure 5.2. Debugging targets available via GDB and RSP over network sockets

Once the TAP has selected this module, it can be controlled via the use of a simple protocol, which is the basis of all communication with the target. In all cases the RSP server has no idea of how to actually perform transactions on its target, and simply calls a simple function, or transmits a simple message to the handler for each specific target target. In the case of or1ksim, the RSP server code actually has direct access to the control and status structs of the processor, but in the case of the Verilog and SystemC models, commands must be translated into JTAG transactions with the debug interface module.

In the case of the cycle accurate model, a SystemC JTAG module is passed basic JTAG operations to performed by this interface function between it and the RSP server. The JTAG lines into the device are emulated and driven appropriately to implement the transactions. A similar system is in place in the Verilog model,

which uses Verilog's system calls, or special simulator-implemented functions accessed via function names with dollar-signs preceding their name, such as `$time` and `$display`. In this case, the RSP interface functions are accessed via functions added to the simulator at runtime via the Verilog Procedural Interface, or VPI. This allows custom system calls to be implemented, and thus custom C code to be run within the simulator. In this case, the JTAG emulation module in the testbench polls a link with the RSP server, which is running concurrently in a separate process. It performs any request transactions and returns data as required via another system call. In all, the interface is similar to the cycle accurate model's SystemC JTAG driver module, except this one is written in Verilog.

Interfacing to a physical target is done using a standard FTDIChip to perform JTAG transactions on lines connected to the appropriate ports of an FPGA or ASIC. The FTDI device is driven from a program implementing an RSP server and appropriate debug interface protocol functions and driver functions.

Debug Interface

The debug interface unit in use in most OpenRISC platforms at present, although not all, is the original module written by Igor Mohor. This device, sits on the JTAG scan chain and receives and performs instructions, returning any data over JTAG as required. Figure 5.3 indicates how a write command on the JTAG TDI is performed. An initial low bit is followed by a command woord, in this case `GO`. Previous command words would have configured the type, length and address of access in a similar fashion. The result of the `GO` command is that it performs the previously configured transaction and returns data if performing a read with, or just, a status response message. In the case of a write, the data is shifted in here, and once the `data_len` bytes have been written, the debug interface will shift out a 4-bit status indicate if any issues with the writes occurred, such as a bus error, overrun or underrun. Both ends of the transaction are validated with a 32-bit checksum value.

Both transactions to the system bus and processor go through this debug interface module. The debug interface is a master on the Wishbone bus, and thus has access to all peripherals and memory that is mapped on the data bus.

Targets

Each simulation model, and physical target incorporating the appropriate debug interface, can be connected to GDB to have software downloaded and executed. This standardised interface is the only way to connect to, and debug, a physical target such as an FPGA or ASIC, and can help recreate issues seen on target in simulation, by allowing the exact same stimulus regardless of the model.

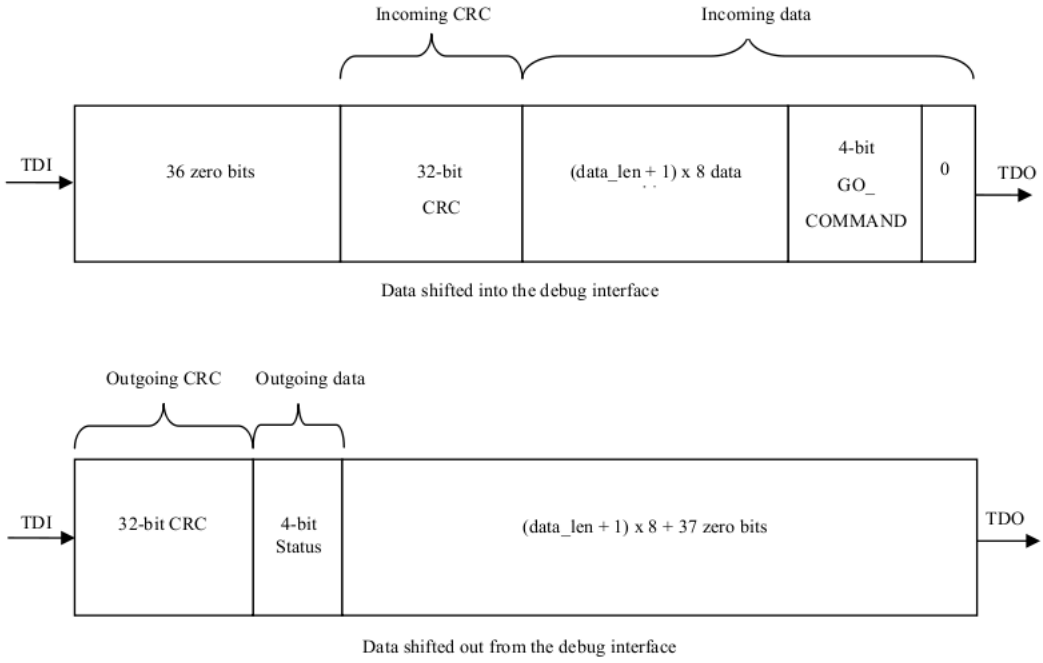


Figure 5.3. Debugging interface write command over JTAG (71)

5.6.2 ORPSoC

The OpenRISC Reference Platform System-on-Chip is used as the primary testbench of the OR1200, and as a project providing implementations of OpenRISC-based SoC designs.

It was initially an implementation targeted at a single board, but now contains several synthesizable, push-button builds and a reference implementation intended for processor core development. The project also contains software tests for various peripherals and can serve as a useful peripheral testbench, but this is not one of its primary purposes.

ORPSoC has been re-implemented over the past two years. It serves as the reference implementation of the OR1200. It is largely a suite aimed at development of OpenRISC-based SoCs, with a project structure allowing a full development flow including backend processing to generate FPGA configuration files or ASIC mask sets.

Command line control

ORPSoC's scripts make use of common GNU tools used for development, such as *bash* and *make*. The use of command line interfaces for launching simulations

and synthesis makes the project simpler to automate which, arguably, increases the productivity of the user once they are familiar with the system. It is often the case that GUIs consume more time than command line interfaces, and provide less-than-direct means of customisations. From a development point of view, GUIs, relying on graphical libraries, are never as portable or widely compatible as command line driven systems and incur greater development and maintenance overhead. ORPSoC does not aim to provide attractive graphical menus, but instead a fully functional, quick and powerful system of scripts.

There is widespread use of environment variables in the scripts, to help configure things quickly and easily at compile time. It's possible to configure things such as specific FPGA device, synthesis directives, and which modules to synthesize, from the command line. This level of configurability, from such a high level of abstraction, is rarely found in GUI-based tools. Having said that, it requires the user be aware of the capabilities of the system and perhaps, due to the lack of detailed documentation about these features at this time, explore Makefiles to determine what options are available. However, in most part, there has been an effort to document all of the available options. Not only are there many useful features available from the command line, all of the scripts, generally Makefiles, are are readily modified to provide any custom function as required.

Manual

The original ORPSoC project had very little, if any, documentation. The new project has attempted to document as much as possible in terms of the user interface, but little exists regarding the internal technical workings of ORPSoC. However documentation is considered important in ORPSoC, not just in the code, but in a central source, which is provided and uses the portable and open GNU Texinfo documentation system.

At present this manual is a forty page guide to simulating and building the various designs included in ORPSoC. It can be found in the OpenRISC project repository's ORPSoC path.

Structure

The layout of the project is based around the reference design, that is the RTL, testbench, and software and scripts available from the root path of the project. On top of this are various OpenRISC SoCs targeting specific boards. The board builds are sorted by targeted FPGA technology, and then by board name. This layout aims to facilitate the development of both the processor and tests for it, as well as encourage as much design re-use as possible and make porting for different boards as easy as possible.

It has been suggested that the easier a platform is to access, that is, to get "hands on" with, the higher the likely-hood it will be used. As ORPSoC functions as both the official test bench for RTL of OpenRISC processors and a project for

pre-built board packages, there is bound to be duplication, but the layout of the project aims to minimise this, while maximising ease of access to the platform.

RTL

The root RTL paths contain all of the modules required for the reference design, and some commonly used peripherals not present in the reference design, such as an Ethernet MAC, an I^2C core, SPI controllers, the debug interface, USB cores, and, of course, the OR1200. These are modules most commonly used in systems.

Each board may not necessarily need the exact same configuration of the OR1200 or the Ethernet MAC, for instance. So they each have their own RTL subdirectories, and must provide the Verilog files which are `` `included `` by any instantiated modules allowing a board-specific configuration of each module if desired. In this way the core RTL may remain in a common place, reducing duplication.

A board port may, however, choose to have a copy of an entire core, or its entire design, within its own RTL path, if configuring it via an include file is not sufficient to allow it to function on the board. However, there is an emphasis on technology independent, easily configurable cores to be included in the common, top level RTL path. This has many benefits, primarily being that if bugs are fixed in the core, they are they immediately picked up by all of the designs using the core, and don't have to be propagated through each version throughout the project.

Core Components

As with any processor-based SoC, there is a set of standard modules required to implement the basic system. It is not mandatory, but each board port so far has utilised a set of modules which are customised to suit the board's needs. These common components are the bus arbiters and clock and reset generation module.

In each system there are typically 3 Wishbone bus arbiters, and a clock generation module to support the basic processor system consisting of the CPU core, its debugging attachments, and a ROM for boot.

Figure 5.4 shows what could be considered the bare-bones of an ORPSoC build. Shown in the top left is the `clkgen` module, which usually must be customised to generate the necessary clocks and resets using the particular FPGA target's technology. The three arbiters are in the center of the diagram, and indicate the connections made between the processor and the memory subsystem. The instruction bus typically has a simple arbiter that either accesses the `bootrom` module, or the main memory. There are two data bus arbiters, one for data widths of up to 32-bits that is capable of performing Wishbone burst accesses, and one for peripherals with byte-wide data buses.

This organisation of the modules is common between the ORPSoC builds, and should hopefully provide an example that is simple and easy enough to modify and customise to suit a different board. In this way, too, top level files should require

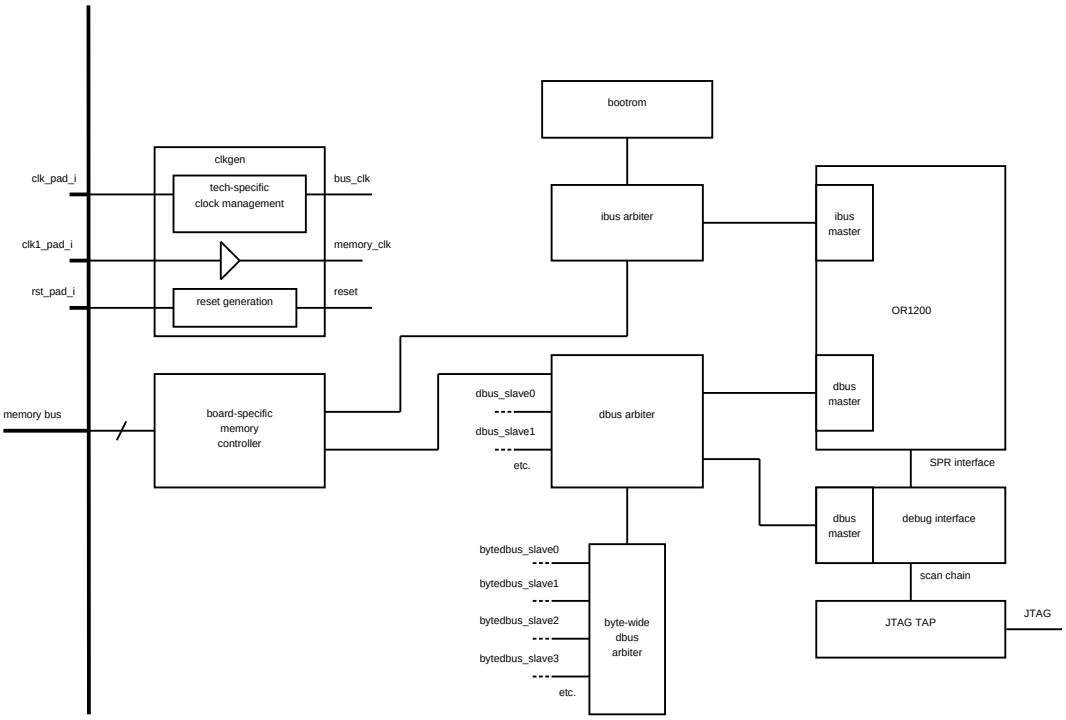


Figure 5.4. ORPSoC block-level basics

minimal work to apply to another board, with the majority of the work being in the addition of the board-specific peripherals.

Simulation

There is an emphasis on providing a simulation environment for each system that can make use of the common set of software tests and drivers. This encourages designers to ensure their system is demonstrably functional, and allows other users an easy way of checking functionality for themselves. With a relatively thorough testbench and stimulus available, it's easier to ensure any modifications to the system have not caused a feature to malfunction by checking all of the tests still pass.

Verilator

The scripts included with ORPSoC now provide the capability of generating a cycle-accurate model from the Verilog HDL RTL description. This feature makes use of the Verilator tool to generate a cycle-accurate C++ model, which is then compiled with a SystemC wrapper and peripheral models to create a simulator executable. The cycle-accurate modeling method reduces the resolution of the simulation to the system clock's edges resulting in greatly increased performance. The generated

models simulate at clock rates in the range of hundreds of kilohertz on modern desktop machines. This enables high-speed simulation of RTL models, and reduce the amount of time required to test large suites of software tests such as those in the GNU GCC regression tests.

Gatelevel with RTL

Some of the board builds' simulation scripts are capable of simulating against the gatelevel netlist of the entire design, or just individual models. Checking the function of the design in its post-synthesis form helps ensure the design's description is correct for synthesis, and can help pinpoint any issues arising between RTL simulation and testing on target.

Although whole-system gatelevel simulations are often tricky to run in that they are difficult to initialise correctly and to stop `X` values propagating from uninitialised blocks, the ability to synthesise and run individual modules' gatelevel netlists in simulation, instead of their RTL description, helps pinpoint any problems arising from synthesis. When problems are encountered in the design on target, and if the entire design's gatelevel netlist is not running properly in simulation, checking the suspect module in its post-synthesis form, while the remainder of the design is still the RTL description, can be very useful.

Testbench

Each board port has their own testbench top level with applicable peripheral models instantiated. It is a relatively straight forward configuration. Additional useful modules developed recently have been the UART support modules, VPI JTAG debug interface stimulus module allowing access from GDB, and ability to have environment variables, passed at compile time, to control various features of the simulation, such as logging, VCD creation, and use of gatelevel netlists of individual modules, or the entire design, if they exist.

The OR1200 processor monitor, implemented in Verilog and used during RTL simulation, has also been updated to provide better logging, ensure cache coherency and correct MMU behavior, and a new experimental feature, ensuring execution of every instruction has the desired effect on the system, has been added. Stimulus modules and protocol checkers for other simple protocols, including Ethernet, have also been added.

Software

A set of software is included with ORPSoC that implements a small support C library, basic drivers and module tests. This is, again, done in such a way that permits board builds to implement their own version of drivers and tests that will override any versions available in the root software library.

Each board has configuration C header file, which specifies parameters such as addresses of peripherals and clock frequency. These are used by the drivers

and the tests at compile time. There is also a system of converting ORPSoC's main Verilog HDL `‘include` file, `orpsoc-defines.v`, and the OR1200's defines file `or1200-defines.v`, into C-compatible include files. In this way, sections of test software can check if certain features of the OR1200, or whole peripherals, are enabled in the design and thus enable tests for them.

Alternate OR1K Cores

Although the RTL implementations don't yet exist, ORPSoC has the facility for running software appropriate for a different OR1K-compatible CPU core. It can provide support for a CPU-specific set of start-up and utility functions to assist during a different OpenRISC processor's development and testing. As the OR1200 has received the criticisms of being resource-heavy when compared with it's technology specific counterparts, perhaps alternate implementations for an OR1K processor with less features, say a small, MMU-less, compact variant specifically for RTOSes. This facility of ORPSoC, to allow the quick switching of processor software drivers, could be used to facilitate alternate OR1K core development.

Tests

The software tests for the OpenRISC processor included in ORPSoC have been expanded to test each new feature added during this work, and the existing tests have been reviewed and updated. The original ORPSoC library contained about 3,500 lines of test code for the processor. This code has been updated and new tests added, and now the library consists of 8,500 lines of code to test the processor. These tests are intended to confirm correct behavior of the architectural features as they are controlled by the programming interface. This library is still not exhaustive, and does not test every possible permutation of instruction and control sequence. However they do ensure proper functionality for a known set of corner cases, and reasonable set of use cases.

Chapter 6

Critical Analysis of OpenRISC

This chapter will critically examine the OpenRISC project, and OR1K architecture. The following chapter will contain suggested ways of addressing the issues identified in this section.

The first aspect discussed here will be the architecture definition, followed by the two main implementations, or1ksim and the OR1200. The quality and level of testing of these implementations will be discussed. Following this the issues with the toolchain and software will be identified. Last, the open source community development model, and how it has been applied to the OpenRISC project, will be explored.

6.1 Architecture

The OpenRISC 1000 architecture is closely related to the DLX architecture, which is similar to the first MIPS (MIPS-I) architecture. No significant changes were made during specification, resulting in an architecture that is very much from the era of RISC processors in the late 1980s. As such, OR1K shares the same issues as those RISC CPU architectures of that era.

Code Density

From an embedded application developer's perspective, the code density of 32-bit RISC architectures is about as bad as it gets. Code density relates to the amount of memory taken up by the instructions required to perform a certain function. This varies between instruction sets, and is typically better (less instructions per function) on CISC machines than RISC. This was remedied by later RISC architecture developments by including 16-bit instructions for the most common instructions. Vendors such as ARM and MIPS developed such instruction set variants. OR1K has none of these features, and thus is limited to 32-bit instructions.

Synchronisation Primitives

The provision of features allowing safe inter-process communication is another lacking feature of OR1K. Although MIPS had introduced such a mechanism with its load-linked and store-conditional instructions, OR1K did not adopt a similar feature. Although there is an indication the early designers did have some consideration of multi-processor implementation, such as the cache system, the instruction set lacks the basic atomic operations required for safe inter-process communication.

Power Management

As power budgets have become increasingly constrained, microprocessor architectures have added features to allow more fine-grain control of operating frequency and core voltage. Providing a software interface for these features allows the ability to have operating systems schedulers make decisions about what to run, and with what power footprint. The OpenRISC, despite having most of these features already, lacks an extensible interface, and actual implementation of such features. The capabilities are mainly suited only to ASIC technologies and it is not clear how an FPGA implementation might employ these features.

A trend seen in nominally RISC architectures, over time, has been the inclusion of non-RISC-like instructions that will otherwise improve the architecture. Examples of these are the AVR32 architecture which consists of duplicates of instructions, containing either a reduced number of operands or immediate length, load/store instructions with pointer arithmetic included (post- or pre-increment by constant or immediate) and additional addressing modes. The ARM instruction set has the feature of making every instruction's execution conditional, and contains the ability to perform shifts and rotates on any register manipulation instruction.

Floating Point

The OR1K floating point and vector instructions utilise the same register file for its operands and results. An increasingly common approach is to provide a register file specifically for each calculation unit, where the register-to-register instructions operate only on an arithmetic unit-specific register file, and specific load and store instructions for those register files are also provided. The advantages are primarily to do with implementation, as a single computation unit's resources can be more easily shared among multiple processors, saving area and reducing complexity within the CPU. It can also potentially allow for data widths in these computation units to be wider than the general purpose register file, say 64 or 80-bits for a floating point register file, compared to thirty-two for the integer register file.

With regard to the floating point instructions, the lack of specification regarding the rounding mode of floating point to integer conversion instructions has lead to a sub-optimal compiler implementation. There is also no instruction for converting between floating point precisions in hardware.

Arithmetic Flags

Despite the instructions for integer addition being nominally signed, both overflow and carry flags are set. This raises problems with detecting overflow in systems that chose to trap on overflow, as additions intended to be unsigned must use the signed addition instruction, and if they overflow into the sign bit, will cause an unnecessary overflow exception.

Another curiosity is that the integer divide instruction somehow sets the overflow bit, but divide by zero sets the carry bit. This is strange in that an integer divide can probably never cause an overflow, except in one corner case of signed divide of the largest negative value being divide by negative one, which would cause the value to be reduced to the largest possible signed integer, which in absolute terms is one less than the largest negative signed value. But this case aside, there is the problem that a divide by zero cannot generate an exception if desired, as it only is supposed to set the carry flag, and not the overflow flag.

Specification Document

The architecture specification document contains a few points of confusion, mainly regarding what is considered a class I and class II instruction. For example, even though the integer multiply instructions were listed as class I in the specification, in the OR1200 and in the GCC port they are optionally enabled. All instruction which set the flag containing immediates are non-mandatory, but it is inconceivable that these instructions be omitted from the compiler implementation. There is also no way to tell, from an SPR or similar, which class II instructions are present or not, meaning complex methods of testing and catching illegal instructions at runtime must be performed if it is to be determined.

The software application binary interface defined in the OR1K architecture specification was left unfinished. It lacked definitions for multi-threading and dynamic library loading functionality.

6.2 Implementations

This section will look at issues relating to the current implementations of the OpenRISC 1000 architecture; or1ksim and the OR1200. The reference platform project, ORPSoC, will also be evaluated.

6.2.1 or1ksim

The OpenRISC 1000 architectural simulator, or1ksim, project has been crucial to the OpenRISC platform as it has provided a fast model to allow early software evaluation, and provides a fast model to run large software regression suites on. The implementation, however, is far from neat and organised, and despite efforts

by contributors in the last couple of years, some issues with the simulator still remain.

The source code, as left by the original developers, had many half-implemented features. These include a dynamic execution model, 16-bit OpenRISC instruction set experimentation platform, debug interface, interactive prompt, and support for other architectures (a DLX CPU model remains.) The progress of the development of these features was never recorded.

The model cannot provide accurate estimates of the number of executed cycles as there is no accurate pipeline or bus model. Although each peripheral, memory and CPU unit (caches/MMUs) can have a cycle count for each transaction configured at runtime, the simulator is let down by the inability to correctly model the pipeline of the OR1200, or any other potential implementation, resulting in inaccurate overall cycle counts.

The built-in debug interface, or interactive console, provided by the simulator is largely ineffective and provides no useful access to the simulator's internals at runtime. If this were more fully featured then performing simple debugging directly on the simulator would be a great deal easier.

6.2.2 OR1200

This section will give an overview of the issues relating to the OR1200 implementation and testing.

Implementation

The main OR1K RTL implementation at present is the OR1200. Its design is based on a description of a microprocessor contained in the Hennessy and Patterson book *Computer Organization and Design*. This design is rather monolithic in that its integer pipeline unit is essentially a single, non-partitioned module. Although not unusual for a single-issue RISC, this makes attempts to replace or modify sections of the pipeline description difficult. Attempts to add an additional stage to the pipeline, in an effort to increase the OR1200's operating frequency, have so far been unsuccessful due to the heavily interconnected nature of the design. The main issues have been the lack of documentation regarding the behavior of internal signals. Attempts to change the behavior in one module, maybe resulting in delaying a chosen signal by a cycle, will break dependencies on that signal, or a derivative of it, in other modules. As there is no documentation on the RTL design of the pipeline, or what signals need to be where and when, any alterations to the pipeline logic ultimately cause the CPU to exhibit subtle bugs. Despite this, modifications to the processor's pipeline description have been made successfully during recent development, although the pipeline has not received an optional additional stage to break the critical path, typically in the fetch and decode stage between caches and MMUs. Outside of the core CPU the modularity of the design is reasonable,

and allows for the optional inclusion of memory management units, caches, memory access buffers and bus controllers.

The debug unit, too, is entirely optional and is easily optioned off at synthesis time. However, the debug unit remains in a state of partial implementation. Hardware watchpoints and breakpoints are not fully functional, and thus not fully integrated with the OR1K debug system as provided by GDB. The primary method of implementing breakpoints is via the use of the trap instruction, with the desired breakpoint address' instruction replaced with a `l.trap` opcode. This is a reasonable method of controlling the program flow while debugging, but has the limitations of not being able to insert breakpoints to read-only code (when, say, executing from flash memory) and cannot be used to implement data-related watchpoints. For these features, hardware watchpoints are required.

Other OR1200 modules which have not been utilised by the various implementations during recent development are the power management, performance counters, *quickmem* and the store buffer. Although the OR1200 does have these modules in varying states of functionality, they remain largely untested by the main development team.

Several bottlenecks to performance for certain applications have been identified. One being the context switch overhead experienced during exception handling for operating systems, and the other with floating point arithmetic.

As the CPU is seeing increasing use running complex multi-tasking operating systems, the OR1K's shadow register file and fast context switch feature might provide worthwhile performance improvements as it would eliminate context switch overhead. Even, perhaps if just two or four shadow register files (out of a possible sixteen) were implemented, this would improve performance in an operating system such as Linux or any RTOS.

Testing

One issue, that there has never been a satisfactory solution to since the beginning the project, is the testing of the core. The original idea of the reference platform implementation, ORPSoC, was to provide a testbench for the processor. A more comprehensive approach to testing is required, and the re-write of the ORPSoC test suite has attempted to go some way to achieving this. However, as was identified earlier, the degree of testing of the OR1200 still is not at a satisfactory level for ASIC developers, although with a lack of proved open source verification tools, this may not be a reality any time soon.

The existing tests, and Verilog HDL testbench for the OR1200, do go some way to ensuring every feature of the processor is tested in some way. Examples of this are the assembly source code tests, exercising every OR1K instruction supported by the OR1200, and more complex tests written in C to exercise the MMU, cache and exception systems. The Verilog HDL processor monitor, used during event-driven simulations, can check cache coherency, and log each instruction for comparison with execution in or1ksim, which is considered to be the golden reference. In addition

to this, the recent work to enable the cycle accurate model (compiled from the OR1200 RTL description) to be used by DejaGNU when running the GCC regression test suite for the OR1K compiler port goes a long way in demonstrating that the processor behaves correctly for a large set of C code compiled by the GCC port.

The support modules used to test the RTL model generally cannot be used to test a synthesised netlist of the design. At present there has been no work done enabling the conversion of a technology-dependent Verilog HDL netlist into a cycle accurate model with the Verilator tool. Nor can the RTL test monitor modules be used on these post-synthesis netlists to confirm the functionality of the design.

The degree to which the processor and its auxiliary modules are tested, at both RTL and gatelevel, needs to be improved. A discussion of ways to move toward a better solution for the OR1K platform in general will be presented in the next chapter.

6.2.3 ORPSoC

The OpenRISC reference platform SoC (ORPSoC) improvements have increased the amount of testing the processor is subject to, and improved access to the platform by providing several push-button synthesis builds for various FPGA vendor parts and platforms. ORPSoC, however, is still not as simple to use as some FPGA vendors' proprietary SoC configuration tools. It lacks detailed documentation of the internals of the components making up the example systems, and cores either support Wishbone classic or B3 bursting, with none supporting the latest Wishbone B4 pipelined burst architecture. Although several board build examples are provided with the project, none are fully featured in that they provide support for all board's peripherals.

There currently exists no example ASIC flows, for use in industry or academia, to help rapidly characterize the core's area, speed and power use during development.

No automated system configuration tool exists and this would help in the creation of new board ports. Despite the author's aversion to graphical user interfaces (GUIs) it would perhaps make it easier for relative beginners to configure and build OpenRISC-based systems.

The use of the Subversion revision control system by OpenCores.org makes it cumbersome for users to manage and develop their own additions to ORPSoC. It has been recommended that a distributed revision control system, such as provided by *git*, would make this easier for contributors to develop their additions, and allow the maintainers to check and merge their work with the mainline project.

On the testing front, the reference design lacks a way to automatically run the GNU GCC regression test suite against the cycle accurate model generated from the OR1200 RTL code.

6.3 Toolchain

The toolchain, consisting of the GNU binutils, GCC and GDB ports implementing support for the OR1K architecture, have received a great deal of improvement during the last 12 months. At the time of writing, the C++ compiler using the uClibc library was the last of the core toolchain and software libraries to be receiving work to ensure it passed its entire regression test suite. The GDB port still required some work to ensure its compatibility with recent GCC port improvements. All GNU tool ports have been updated to the latest versions as of early 2011.

One major feature noticeably lacking from the OR1K toolchain is support for dynamic library compilation and loading.

The upstream GNU binutils repository contains files that implement support for OpenRISC 1000, however these files were first committed in the early stages of the project, and have not continued to be maintained by the contributor. The toolchain port implementation being used by the development community on OpenCores is entirely separate from the versions existing upstream. For many reasons it would be advantageous to have the OR1K toolchain port upstream in the GNU binutils, GCC and GDB repositories. This would maximise exposure of the architecture, and with a newly reinvigorated port, would hopefully be easier to keep up to date with the developments at the head of the repository.

The availability and ease of use of the toolchain has always been a problematic area for the OpenRISC project. In an effort to make the tools easier to obtain, a toolchain release program was initiated by the contributors to the project on OpenCores. However, the releases of the toolchain have been continually delayed as the small development team worked through the various methods of distribution and installation on the user machine.

A convenient feature of almost all GNU/Linux operating systems has been their package management systems, used to ease the distribution and installation of software. Examples of these are the Debian Aptitude system, or the Red Hat Yellow dog Update (Yum) system. OpenRISC project contributors have been working on getting the toolchain distributed via these channels, too, but due to the intricate nature of a cross compiling toolchain supporting two C libraries, and the limited time of those contributing to the project, despite best efforts this work has not yet been completed. The use of these channels to distribute the toolchain would provide the toolchain greater exposure, and hopefully, increased uptake.

The testing of the toolchain has been the greatest area of improvement recently, and has helped prove the GCC port is mature and shown that the simulators behave correctly. However there is still no mechanism in the DejaGNU testing suite allowing tests that require execution, to run on physical targets. This would allow rapid validation of all generated code on real systems and help provide a more complete testing platform. This would also allow designers to be confident their processor implementation on the hardware functions as expected.

6.4 Software

This software section will focus on three main areas; libraries, operating systems and applications.

6.4.1 Libraries

The two main C libraries which have received work recently are the newlib and uClibc C libraries. The newlib library is being used to provide the capability of compiling simple C programs to run on the "bare metal", or without an operating system, and the uClibc library is providing the capability to compile code to run on the Linux operating system.

There is no port for the GNU C library, or glibc, planned or expected to be needed for the OpenRISC platform any time soon. As OpenRISC implementations are largely targeted at embedded uses, the small but effective nature of uClibc and newlib are the right choice of libraries to port.

What is lacking is low-level functionality in the newlib library's board support packages, implemented in libgloss. At present the only target "board" for bare metal applications is the or1ksim simulator. Although additional board support packages are in the works, a minimal library and API allowing access to the architecture's basic features should be implemented to extend the capability of these bare-metal programs.

However any additional functionality should be implemented at the operating system level, and as will be discussed next, additional RTOS ports for OpenRISC should be a goal, as these are ultimately more appropriate for the kind of performance and implementation options the platform provides. The newlib library can provide support for the RTEMS RTOS, of which there is already an OpenRISC port of in the project's OpenCores repository. Without the newlib operating system interface to RTEMS, however, or a guide on how to customise and build a targeted compiler with RTEMS support, this capability will go unused.

6.4.2 Operating Systems

At present the only well-tested operating system ported for OpenRISC available from the public project repositories is the Linux kernel. The OR1200 implementation can be used to run this port. However as this implementation largely targets FPGAs for the time being, it makes sense for more embedded-oriented operating systems to be ported for use on the platform.

The Linux kernel typically requires at least 16MB of main memory to run, takes up at least a megabyte or two itself, but has almost endless capabilities in terms of the features it can implement. Many of these features, however, will not be necessary for an embedded system, and those that are are desirable are largely supported by other, more light-weight operating systems. Examples of such features useful for embedded platforms are networking, file system support and the ability

to interface to various hardware devices (via drivers and an abstraction layer.) There are a plethora of operating systems with reduced memory demands than the Linux kernel in terms of compiled kernel size and memory footprint, and that may provide equivalent functionality for a certain task. Examples of operating systems that would be suitable to port to OpenRISC will be discussed in the final section.

Some examples of these alternate operating systems for embedded use already exist in the OpenRISC repository. Ports of eCos and RTEMS are available, however they only support one board each, were ported several years ago, and appear to not have had extensive testing.

6.4.3 Applications

The OpenRISC platform has almost no software application packages in its public repository. It's likely that the reason for this has been the lack of a thorough operating system implementation on which to develop applications.

The standout bare-metal application developed for the project is the ORPmon monitor and bootloader program which assists with board bring-up by providing diagnostic and benchmark capabilities, and can load other programs from flash or over network using the TFTP protocol.

The recent improvements to the Linux kernel provide a better platform for application development, however the obstacle remaining in the way of being able to implement the vast amount of software available for GNU/Linux platforms is the lack of support for shared objects in the linker and loader. Until this feature of the toolchain is implemented, applications must be linked statically, drastically increasing the size of executables, and rendering a lot of the applications unbuildable due to their reliance upon shared objects.

6.5 Open Source

The open sourcing of the IP developed for the OpenRISC project, and others on OpenCores, has been both a hindrance and a help, in that it lays bare the development status, but is helpful in that it allows anyone to participate in the continued development of cores. This section will discuss the pros and cons of the open source approach.

6.5.1 Unverified and Unpopular

Despite the widespread use and acceptance of software that is developed under open source licenses, open source RTL projects have not received the same sort of interest or involvement from larger IP companies. The following points address the question of why this is the case.

One issue is that for people developing IP destined for implementation in ASIC, the lack of a formal verification suite in most of the open source IP, and thus inability to quickly verify a core's functionality, is unacceptable given the extreme cost of

fixing bugs. One might suggest the mandated inclusion of a thorough verification testbench with a core, however the industry standard verification tools are expensive, proprietary, and can vary from vendor to vendor. It is somewhat surprising no industry-standard open source verification tools exist yet, but this poses a barrier to entry into the serious IP developer market as the cost of a suite of EDA tools capable of performing verification is significant. The lack of open source variants of verification tools will continue to limit the applications of these cores to lower-risk implementations, such as in FPGA where RTL changes can usually be made cheaply.

6.5.2 Who Has The Hardware?

Another barrier to entry for open source hardware development, not faced by those participating in software development, is the requirement of a specialist prototyping platform on which to implement designs. These platforms, typically FPGA-based boards containing multiple peripheral ICs, need to be purchased, as well as debugging and programming hardware. Add this to the usually cumbersome FPGA vendor programming tools, and the relatively steep learning curve hardware development imposes on beginners, and it's not too surprising that technically minded individuals wishing to contribute to an open source project would choose a software project over a hardware project almost always. It is also true that the utility of any hardware design that could be implemented on FPGA is limited by the fact that it is done at a very low level of abstraction, and to achieve any "useful" outcome for a tinkerer or hobbyist, it usually requires a lot of work throughout many levels of abstraction to achieve something that is easily usable from an interface on a modern PC. An example might be the development of a core to perform non-standard I/O transactions with a sensor or other boutique IC, to provide information to a program assisting with home automation, or controlling a scale model and the like. This would require the development and testing of the hardware model and implementation in FPGA. Assuming there was a microprocessor running on that FPGA providing network services via an RTOS, this new custom module would then require its software layer developed, which means a driver, and satisfying various OS-level hooks into the application running on the FPGA's microprocessor, to provide the data over the network link. Only then would this sensor's data then be available to the higher-level application. This is just one example where, quite probably the designer might have chosen a solution that uses a standard bus, however there's often cases for custom controller or interface cores in FPGAs to provide access to legacy, or very-new or esoteric bus standards, and highlights the extra work required beyond writing RTL to provide the physical interface. Considering the amount of development and testing required to typically implement these solutions, it would be easy to become overwhelmed by the amount of work required to complete such a seemingly trivial task.

Compare this work with that involved in beginning work on an open source software project, which would usually consist of downloading a development source

tree and building (within minutes) the project with development tools already included, or easily obtained, on the operating system. The application can then be run on the host system to check functionality, and the development cycle largely ends there. The differences are the inherent access to the development platform (the host machine), the simpler development tools (gcc, make on the host system) and the shorter and easier development and testing cycle (running on the host machine via a shell.)

As more open source hardware projects are developed, and more streamlined systems of development are put in place, it can be expected that these barriers to entry will become diminished. The early days of open source software development would have seemed equally challenging and labor intensive. Over time, however, improvements in the way projects are organised, and the tools used to develop them, have occurred. The body of available open source software has grown and continues to do so at an increasing rate. It can be expected that with time and increased participation, open source hardware will achieve similar success.

6.5.3 Licenses

One issue that remains to be resolved is that of licensing for open source hardware designs. The OpenRISC project uses the GNU project's public licenses. These specifically relate to software, and it's not known how well these apply to hardware. The GNU project's website contains a frequently-asked-questions (FAQ) section that addresses this query. It states the following.

> Any material that can be copyrighted can be licensed under the GPL. GPLv3 can also be used to license materials covered by other copyright-like laws, such as semiconductor masks. So, as an example, you can release a drawing of a hardware design under the GPL. However, if someone used that information to create physical hardware, they would have no license obligations when distributing or selling that device: it falls outside the scope of copyright and thus the GPL itself.

This is not so clear about designs targeted for FPGA, or even RTL code, as it may end up as a set of masks, or it may end up as a binary bitstream for FPGA configuration.

One indication of how nascent the open source hardware development idea is comes from the relatively recent (February 2011) publication of a set of principles for open source hardware community participants. The following is the Open Source Hardware (OSHW) Statement of Principles 1.0 from FreedomDefined.org.

> Open source hardware is hardware whose design is made publicly available so that anyone can study, modify, distribute, make, and sell the design or hardware based on that design. The hardware's source, the design from which it is made, is available in the preferred format for making modifications to it. Ideally, open source hardware uses readily-available

> components and materials, standard processes, open infrastructure, unrestricted content, and open-source design tools to maximize the ability of individuals to make and use hardware.(65)

These guidelines are provided as a benchmark against which the licenses of designs using the "open source hardware" label should be measured. What follows the principles on FreedomDefined.org is a list of guidelines dealing with the specific topics of source and documentation, derived works, and the limitations of the licenses. It is expected, according to these principles, that all source and documentation material for the design be made available. Any derived or modified works should be permissible. There is also the acknowledgment that any Open Hardware license can probably be used to restrict (or, in this case, intentionally *un*-restrict) the *plans* of a design, but not the use of the manufactured device. These are all concepts found often in open source software licenses, but again, it is not so clear in every use case of open source hardware designs how these licenses for software would apply.

What remains, however, is for actual licenses to be written, applied to works and their validity tested. For now, the first major open source hardware license to be released is the Tuscon Amateur Packet Radio Open Hardware License (TAPR OHL.) The TAPR OHL authors identified the problem with existing software licenses as being that while copyright protects documentation from unauthorized copying, modification, and distribution, it has little to do with your right to make, distribute, or use a product based on that documentation(66). Their license identifies patents as an issue, but claims those who benefit from the OHL cannot then bring a lawsuit claiming that the design then infringes their patents or other IP. How open source hardware licenses and patent law will be compatible with regards to handling infringement is yet to be seen. Regardless, the TAPR OHL has been adopted by a handful of hobbyist and commercial interests. It has received criticism from the Open Source Institute (OSI) for adopting a different meaning of the word "distribution" than is typically used in their licenses, and thus does not have widespread support among established open source promoters(67). However, it is likely alternate open source hardware licenses will emerge to suit most needs.

For the OpenRISC project there is a balance to strike between adopting a license that is either too liberal, and thus less likely to result in contribution back to the development community, and a license that encourages more open source development but is then deemed too restrictive with regard to the use of open source IP with proprietary IP. On the one hand, there is a desire to increase the participation in open source hardware development in general, and in the OpenRISC project specifically, and to increase the body of available work, which a viral license along the lines of the GNU GPL (where synthesis is considered equivalent to static linking) can achieve. On the other hand the use of the work in largely-proprietary designs by ASIC houses is desirable as it helps prove the IP's worth, and so a license permitting the use of open source IP along side proprietary IP is desirable. For the OpenRISC's RTL implementation, the OR1200, the non-viral GNU Lesser

(L)GPL license has been used and, although the instantiation of an RTL IP block in a "hardware" design is not dealt with specifically in the (L)GPL, it is the latter (more liberal) licensing approach that has been taken for the OpenRISC project thus far.

However, perhaps this factor has, too, contributed to the relatively low level of community participation thus far in the OpenRISC project. Comparatively, the early stages of the open source software movement saw a lot of code released under the GPL, which ensured all other code used with it came under a similarly "restrictive", viral license, and ensured a large body of code was released into the public domain. However, not all open source software was released under these viral licenses, with the BSD and MIT licenses being less restrictive with enforcing the freedom of the user.

6.5.4 OpenRISC

As stated already in the discussion on open source technology, it is typically not the case that open source development models are adopted for cutting edge, innovative work. The bulk of open source projects aim to implement relatively well known solutions in a way that permits openness and removes restrictions found in other proprietary implementations. This is the certainly the case for the OpenRISC project. It is largely taking ideas that are already well known and commoditized and creating a version with more freedom for the end user. There was very little, if anything, innovative in the OR1K architectural specification. This does not mean the results are valueless. Nor does it necessarily preclude any future OpenRISC architectures or implementations from aiming to innovate.

Making this clear helps answer questions regarding the motivations for most open source development, and that of microprocessor architectures, at least, in the early twenty first century. The resulting work can either be a source of pleasure, or a source of income made by developing and supporting such designs - maybe, even, both.

The motivations for pursuing open source development might be largely subjective. Despite varying motivations, though, the goals of these open source projects are usually aligned. Perhaps one might want to see the ceasing of the continued reinvention of the technological wheel. For the most ambitious projects, they might aim to supplant the proprietary alternative. Either way, the goal is to provide something useful, productive, and open.

For example, consider the success enjoyed by the Linux kernel project, with thousands of contributors and dozens of commerical enterprises regularly participating in development. The Linux kernel is clearly competing with the proprietary variants of UNIX and Microsoft's server products, and more recently, Microsoft and Apple's desktop PC operating systems, and in all cases is experiencing tremendous success.

For a project like OpenRISC, a goal may be widespread use among vendors who currently use processors by the industry leader, ARM, for their microprocessor-

based systems. For the reasons already outlined concerning the verification of open source IP and the risks involved, this kind of uptake is unlikely to happen in its current state, however that is not to say it never could. At the time of the OpenRISC's inception, observers indicated that open source IP could become a major player in the industry, if done right(43). It could be the case that for designs which are largely re-implementations of well-known technology, the fundamentals of which are 25-years old, open source development is a realistic approach. It could potentially lead to lower-cost ASICs, and thus lower-cost consumer electronics, as royalties would be eliminated, and teams of engineers tasked with implementing in-house controllers or microprocessors, rather than working to implement and support a a largely superfluous design, could either be tasked with honing specific parts of the open source processor to achieve greater efficiency for their application, or be retasked entirely to design more innovative IP.

It is very probably the case, though, that it is a scenario of "build it and they will come". It's unlikely a concerted effort among proprietary IP developers would spontaneously occur. Even if it were to appear from the likes of universities and government funded research houses, the likely-hood of developing microprocessor designs attracting the attention of ASIC houses, then spurring on collaborative development, is low. Factors already discussed, such as difficulty of verifying such designs and high barriers to entry due to cost and expertise are there, but so too restrictions on the kind of information required to hone ASIC implementations - technology library specifics from the fabrication plants themselves, which are closely guarded secrets and unlikely to be released for open source projects to prepare designs for. There is probably a distinction to draw here, too, between ASICs deriving their competitive advantage from their microprocessor or from other custom IP. Potentially those relying on a less-cutting edge processor might be happy to help contribute to an open source development project for a microprocessor, assisting by contributing engineering hours to verification and perhaps even adding features, in order to avoid royalty payments which could lower their per-unit cost. Developers targeting FPGA implementation, where barriers to entry are lower and implementation technology specifics non-confidential, might be a good fit for such a collaborative development approach.

6.5.5 Summary

The nascent state of the open source hardware community, and thus the small amount of IP that is currently open source, on top of the relatively few number of contributors, means that at present what is being achieved by open source hardware projects pales in comparison to the achievements of the proprietary IP industry and, as well, to the achievements of the open source software community. This can give the impression of a largely ineffectual community if one is not aware of the constraining factors.

Chapter 7

Future Directions

This chapter will further discuss and suggest solutions to the issues identified during the analysis of the OpenRISC project. More straight-forward issues, such as not-yet-implemented features, will be briefly discussed, and more interesting issues will be discussed in greater detail. This section will serve as both a checklist of tasks to be completed on the OpenRISC project, and discussion of solutions to broader issues.

7.1 RTL Testing

This section will discuss the lack of verification of the OR1200 core itself and look at the broader issue of verification of open source IP in general.

7.1.1 OR1200

A formal verification suite of the OR1200 should be developed. Work to this end has been implemented (68) and has been submitted to the project's public repository. However, this is yet to be made usable by any open source tool and, ideally, should these tools become available, this testing suite should be evaluated, and perhaps updated to test both instruction and exception behavior.

A big issue for those wishing to work with, and modify, the OR1200 is the lack of an internal reference specification detailing how each sub-block functions and what assumptions this function is based on. Creating some basic documentation of these blocks which could be used to create testbenches for each (approximately ten modules within the CPU itself) should probably occur. With this, newcomers to the processor may find they can get up to speed with the design quicker, and may find themselves able to make contributions quicker.

It is unlikely a block-level testbench of the `or1200_cpu` module will ever be developed. However the testing performed by the GNU GCC regression suite against the RTL model can give a good indication that at least a large amount of compiler-generated machine code will execute correctly on the processor. These tests, in

conjunction with the tests of exception handling and additional architecture modules in ORPSoC should help indicate the processor's functionality.

An issue of being able to check the functionality of a gatelevel netlist still remains. However overall the current testing provides a relatively good set for checking for regressions during development and provides fairly solid assurances about the execution of compiler-generated code.

7.1.2 Open Source IP Verification

The issue of the lack of verification of open source IP is something that is holding back open source IP from becoming more widely accepted within the semiconductor industry. It is not clear how this will be solved, either. It is clear that some method of verification of the cores should become standard and supported by freely available tools. However, while the EDA tool vendors have forged on with their respective verification techniques and tools, very little has occurred in the open source EDA arena to pursue equally well-implemented verification solutions. Open source IP developers, currently, would need to purchase licenses for proprietary verification tools to provide the ability to run a verification on their core to the end user. This is prohibitive to most who might wish to develop a core.

Verification and testing is typically considered less interesting work than the actual IP core design implementation. It is usually the case that designers will spend less time on testing than development if targeting FPGA, where testing can be performed relatively quickly. This results in an approach where the core is made to work in one application and then published with incomplete or inaccurate documentation, and no thorough testbench. This unwillingness to implement comprehensive testbenches or verification suites is something that will make any IP unattractive, proprietary or not. It is often more time consuming for a designer who downloads a free IP core to understand how the core works without thorough documentation and testing than it is for them to implement it from scratch. This is especially true when bugs in a larger system are tracked back to a poorly-tested and poorly-documented IP. If a thorough testbench and set of documentation has not been written the core may as well not exist. This is a major issue that must be understood and addressed by the community raising the standards of open source IP core development.

What would help is an effort towards an open source set of tools to provide the ability to thoroughly test IP. The Verilator tool, capable of generating cycle-accurate C models from Verilog HDL, helps in that it opens up the possibility of writing testing suites in either C++ or SystemC.

Ultimately industry-grade verification tools will bee needed, which is probably just a fully-featured open source System Verilog simulator. This could be used to run verification suites based on the new Universal Verification Methodology libraries which aims to standardize the verification methods and programming interfaces. Communities such as OpenCores could look at making a big push toward increasing the standard of verification of cores by adopting UVM as their recommended verification platform. However the lack of simulators capable of running the test-

benches is currently holding back any concerted effort at ensuring open source IP is tested to levels nearing industry standard.

7.2 OR1200 Implementation

This section will look at the OR1200 implementation and discuss the issues concerning it.

7.2.1 Features

As previously mentioned, the core CPU implementation is largely monolithic in that it is barely modular and is heavily interconnected. This design does not make it easy to have its constituent components modified or interchanged. There is no time effective way to alter this and will most likely remain the way it is.

Debug Unit

The debug unit appears to be not entirely complete, and discussions with early designers indicate that the debug capabilities of the OR1200 were very much implemented very late in the project. This may indicate why things such as hardware breakpoints and watchpoints don't appear to be fully functioning. The GDB implementation left by the original OpenRISC developers never made use of hardware breakpoints or watchpoints, and the lack of implementation in the OR1200 may be an indication why.

Bus Interfaces

The release of the Wishbone Bus specification revision B4 in 2010 saw the addition of the ability to stall accesses and pipelining of accesses. The OR1200's line burst cache accesses would be helped by adding support for B4 on the OR1200's Wishbone bus bridges (interfaces.) At the time of writing very little IP on OpenCores supported the B4 revision pipelining feature, however if the OR1200 supported it, it would assist bus and peripheral designers as they would have a master to help test with.

There have been requests for to add support for additional bus standards, such as ARM's AMBA. This could easily be done and is a good example of a small project that could increase the versatility of the processor.

QMEM

Micro-architecture features such as the store buffer and quick-memory (QMEM) systems have not been fully tested, nor have their performance gains been measured. To increase determinism of the processor's exception handling ability, handler code might be placed in a ROM accessible via QMEM, and would ensure the code is available in a similar manner to being permanently cached. These features need to be checked to see that they are working and in the case of QMEM, must have their

mechanism for accessing a ROM implemented in a versatile way. The comments in the QMEM top level source file header indicate that optimisation is still required to reduce the store latency from two clock cycles to one.

Multiple Way Caches

For designs that may be able to make use of large amounts of on-chip memory, having multi-way caches implemented in OR1200 would enable it take advantage of the larger amounts. Currently the largest amount of cache the OR1200 can be configured for is 32KB. This is a line-length of 8 words, and 1024 lines. Instruction and data cache can be up to 32-way, therefore providing a maximum of 1MB of total cache for each instruction and data memory.

Multi-way MMUs could potentially provide a worth-while performance benefit to the performance of the Linux kernel port on OpenRISC. It has been observed that the MMU TLB cache miss rate is high on a busy system. At present there are just 64 cached TLB entries in the OR1200, but this could be increased to 128 per way, and 4 ways for a total of 512 TLB entries, or 8 times more than at present. One would expect the TLB miss rate to be reduced by even adding just two-ways.

One outstanding issue to do with the invalidation of multi-way caches is how the invalidation is controlled. For the instruction and data caches it's not clear how the invalidate register (ICBIR/DCBIR) should work. The uncertainty lies in whether only a specific way should be invalidated (determined by match of line tag with address written to the D/ICBIR) or should all ways be invalidated? Similarly for the MMU invalidate register. However, in the MMU's case, with direct access to each match register (containing a valid bit) it could be easy enough to simply detect which of the ways has the desired entry to invalidate directly via their SPR interface. However, issues such as this must be resolved before any implementation can occur as it is unclear how software would initialize the system. There are no instances of multi-way implementations or driver code.

Co-processor interface

One additional feature that could be of use to those designing coprocessors for the OR1200 is the implementation of an interface to attach coprocessors. This interface would probably need to provide access to the register file, and stall the processor for multi-cycle instruction sequences. The four OR1K custom instructions could be used to control the coprocessor, or the immediate value of the NOP instruction.

FPU

Although the CPU now has a single-precision floating point unit, floating point arithmetic in C code may rely heavily upon the software libraries to perform operations for double, or greater, precision calculations when used. Double precision is often used for floating point operations as the GNU GCC compiler has a tendency to "upgrade" the precision in use from single to double. Although this compiler

behavior can be controlled to some extent, a lot of software uses the `double` type explicitly for any floating point operations, and these instances cannot, and should not, be reduced to single precision. The benefit of having the floating point unit is then largely lost as software routines are used to perform the double precision floating point arithmetic. The compiler, however, places 64-bit values in adjacent registers, and testing has shown that this could be the basis for implementing a 64-bit datapath unit such as a vector processing unit or double precision floating point support. All that would be needed is 2 cycles to get the operands out, and two cycles to write back the result.

The single precision FPU, as it is now, is quite resource intensive. There is likely to be duplication of sections due to the piecemeal approach of implementation. For example, the pre- and post-normalisation stages in FPU 100 appears to be repeating a similar operation for each arithmetic block. Perhaps the synthesis tool is already aware of this and optimising it, however the module is very large, and it is likely this normalisation logic is being repeated.

The serial multiplier and divider within the FPU are very similar to the integer multiplier and divider that is already in the CPU design. Perhaps these could be implemented more efficiently to save area.

7.2.2 Synthesis

The issues relating to synthesis of the processor are largely to do with performance, and specifically, the critical path that usually emerges when caching and MMUs are enabled.

When the OR1200 is synthesized targeting Xilinx Virtex 5 technology the critical path for the data emerges from the data cache tag RAM, through the cache's address hit logic, through the data TLB miss signal, and then into the CPU to trigger a DTLB-miss exception, which ultimately controls the enables on the register file's RAM, which is where the path ends. The path is 16ns long, and therefore constrains the design to a maximum frequency of 62.5Mhz. Having the data cache hit/miss logic unregistered is adding 5ns of logic propagation delay and 1ns of routing delay onto a 16ns path. Without this, this path would have potentially an extra cycle of latency but could be clocked at 100MHz. Another approach could be to allow the data address to arrive earlier, potentially increasing a non-critical path elsewhere and allowing the shortening of this one.

The critical path in the instruction fetch and decode stage is similar (just over 16ns) and originates from the instruction cache's tag RAM, goes to the cache hit logic, back to the `or1200_genpc` module which then appears to be combinatorially calculating the next address, which is put out through the IMMU, and a TLB hit comparison, the result of which triggers an exception in the case of a miss, and flows through various pipeline control logic until its endpoint as an enable signal on the register file RAM. The solution to this is less obvious, as the path could be registered at several places. More investigation, into what is actually causing this path to result from synthesis, is required to adequately solve this problem.

7.2.3 Documentation

There is no internal documentation about the how the OR1200 operates. Despite the design being very similar to the Hennessy and Patterson DLX machine, its other interfaces and assumptions about signal timing, are not explicitly set out.

The OR1200 specification has received some work, however it still requires a better description of the debug system.

7.3 Test Software

At present, both or1ksim and ORPSoC have their own, separate suites of test software. Ideally a unified repository of all OR1K test software should be created, and should be accessible by all OR1K designs in a way that allows them to automatically test the software against their designs. The or1ksim testsuite does this in the most optimal way using the DejaGNU test framework, and ORPSoC less so using its custom testing scripts. A unified, DejaGNU-capable OR1K test software library would be ideal if it could be run against every model with ease. At present, OR1K test software development occurs in both or1ksim and ORPSoC as features are added or fixed. The task of bringing that newly implemented test software to another project seems like unnecessary duplication, and as more implementations are created, increasing fragmentation of a great set of test software will occur.

The major work in this amalgamation of libraries will be standardising the runtime code they rely on, such as initialisation routines and exception or interrupt handlers. Both have their own custom implementation, and although some work towards standardising them occured, the lions share remains to be done.

7.3.1 Using libgloss

Part of the solution may be in recent work on the OpenRISC portions of the libgloss library which has implemented a method of very simply changing the targeted board of software at compile. Using the `-mboard=` option, an appropriate set of parameters is linked into executable at compile time, and ensures the start-up routines are correctly customised for the board.

In this way, a library of test software could be be compiled with one target in mind, say or1ksim, and recompiled with just the target redefined to be capable of running correctly on the ORPSoC reference design.

As each existing test software library was written for their specific targets, they contain certain assumptions that may not be correct about another model. An example is ORPSoC's OR1K tick timer test, which assumes a certain behavior of the pipeline, and relies on a specific value of the timer after a certain instruction flow. This exact same value is not the same under or1ksim, which makes no effort to model the pipeline at all, and thus this section of the test is not appropriate for all targets. A question of whether to allow defines, based on the targeted board, or

whether such a test should be removed and replaced by something more portable. Sorting out the tests, and facing situations such as this, will be an arduous task.

A unified test library, however, will benefit every current model by increasing the available test cases, increase the usefulness of test writing in the future, and be of great use to the development of any future OR1K processors, or OR1K-based systems.

7.4 Platform Access

Given the learning curve faced when dealing with a microprocessor architecture at the register transfer description level, it is normally desirable to use it first at a higher level, or deal with it as a system-level block. This allows the processor to be implemented, and allows first use via higher-level control of it through software programming in languages such as C.. Once users are more familiar with it at a higher level they then feel more comfortable when delving down into the inner workings. This makes providing easy access to the platform important to increase the number of users, and therefore, the number of potential participants in the open source development.

The motivation for developing or1ksim and ORPSoC has been along these lines. They both provide relatively easy access to the platform; or1ksim by allowing users to instantly simulate software with a rich set of peripherals, and ORPSoC by providing push-button synthesis builds which, when combined with the ORPmon bootloader, provide easy access to the platform in a very tangible form.

Through continued development of ORPSoC's board support builds, and improving or1ksim's ease of use and set of peripherals, the platform can become easier to use by lowering barriers to entry of lack of access to developing on, and using, the OpenRISC platform.

7.4.1 or1ksim

or1ksim's existing set of peripherals may not match the needs of those who wish to prototype an OpenRISC-based system. It would be largely trivial to implement new peripherals, however a better tutorial or documentation demonstrating how to implement new peripherals could be created to better explain this process. At present very little official documentation or examples exist.

The interactive prompt of or1ksim lacks a way of enabling or disabling peripherals, and has no simple "run" command that will run until completion or breakpoint. Although, perhaps the interactive prompt is superfluous when or1ksim can have GDB attached to it via RSP.

or1ksim has been left in a state by previous developers which sees many part-implemented features left in the code. There was obviously an effort to increase the execution rate of the model by using dynamic execution, or dynamic binary translation where the OR1K instructions are translated more directly than an interpreter would model them at runtime. There is also models of the DLX CPU and an exper-

imental 16-bit OR1K CPU. This, and other dormant code, are now of questionable value, and the project could do with a cleanup. For reasons explored later in the discussion on QEMU, the execution speedup work may be proved unnecessary.

7.4.2 ORPSoC

Increased Automation

This section will discuss the motivations for increasing the automation and ease of use of ORPSoC.

It is common for FPGA vendors to provide graphical FPGA-targeted system configuration tools. These are provided by Xilinx and Altera for their Microblaze and Nios II platforms, respectively, and appear to lower the level of understanding required before launching into custom system generation. Currently in ORPSoC most of the work of implementing a new system would require a non-trivial amount of hand-coding and connecting the modules seen in figure 5.4, such as the clock and reset generation module, the top level modules and bus arbiter, and for newly added peripherals, any I/O and glue-logic required.

For more experienced RTL designers, this work would be simple and the ORPSoC project encourages people to develop ports for their board based on existing examples and to then submit them for others to make use of. This library of boards should provide push-button implementations that greatly improve the access to the OpenRISC platform.

However, it's a realistic goal to think of implementing a largely automated board porting system, which would create the bare-bones RTL of a board support package, with customisable bus arbiter and peripheral system, where existing peripherals can be chosen to be included, and stubs for any user-specified peripherals created and left for the designer to populate with their board-specific modules.

Further to this idea of automating the generation of systems is the idea of implementing a standardised way of configuring such SoC designs. A configuration build system, similar to that used by the Linux kernel, which generates listing of features to be included in the kernel at compile time, could be used for a board's SoC configuration. A generated configuration file, included at simulation and synthesis time to control which modules, and with what configuration, are used. This could be done via generating a list of compatible Verilog HDL `` `define `` and parameter values for inclusion at synthesis time. This would require common infrastructure, such as buses and clock management, are suitably parameterisable, and that each conforms to the agreed standard of configuration. A configuration tool might also need a method of determining the options available in a design for a user to configure. Lastly would be a method of generating, or selecting, which constraints files for timing and pin-placement based on the configuration of the design. Most backend tools are unforgiving with the specification of constraints. A method of passing selected portions of pre-written backend scripts might be the best option, and is is already done this way in some ORPSoC boards already. Despite the adoption of

TCL, and some common constraints formats such as SDC among EDA tool vendors as a way of automating commands, each vendor still largely has their own custom way of specifying constraints to the backend flow, and a one-size-fits-all solution for this part of a project is bound to be tricky to implement.

At a hardware level, a useful feature would be to define a configuration information standard that all of the board ports are compliant with. This would probably involve specifying a location of some registers that can be read to indicate which cores are in the system, where, and if cannot be determined from the core itself, what their capabilities were configured to at synthesis time. This would help with any automated system assisting with custom board build creation. Standards such as this do exist already, however most are subject to proprietary licenses and require membership and fees to be able to use. The development of an open standard in this regard, if done correctly, might make the job of supporting legacy designs, and software drivers, much easier.

The script system of ORPSoC needs to be improved to reduce duplication. With relatively few board builds, it is simple to propagate new simulation script features to the remainder of the files. Instead, a central set of Makefile fragments should be employed to increase the reuse of common rules.

Boards in libgloss

Work has commenced to provide support for board ports in the OpenRISC GNU toolchain port's newlib libgloss library. Once a board port is created in ORPSoC, a single file and configuration option can be added to the libgloss library allowing users to compile bare-bones applications for their board by simply passing an option such as `-mboard=myboardport` during compilation with GCC. This would ensure the board's bring-up code for the bare-metal application is appropriately configured for that board and would provide quicker and easier programming access to the board.

7.4.3 QEMU

Although or1ksim is quite fast (up to several MHz on a recent desktop PC) and its peripheral library has a lot of OpenCores staple modules, there is a popular and high performance simulator, capable of supporting multiple architectures, that could have OpenRISC support added. This simulator project is named QEMU and achieves very high speeds of execution of the simulated architecture as it dynamically translates blocks of simulated-architecture instructions to, ultimately, the host's instructions. The speed increases due to this would make, arguably, a multi-gigahertz desktop PC "run" OpenRISC software faster than any existing hardware.

The simulator can do full system simulations, and has an extensive set of peripherals, with networking capability via the network interfaces, already implemented.

There is also the ability to add support for running OpenRISC Linux userspace programs, known as "User mode emulation". It takes the userspace program's instructions and executes them in the manner described above, and provides the

kernel services via an endianness translation and a 32/64-bit conversion layer that ultimately lets the host machine's kernel act as the kernel layer.

Implementation would require a maintainer to first implement the system level port, and then the user space mode port. It should be relatively straight forward with the only issue being handling the highly-specific aspects of the architecture such as CPU peripheral modules (caches, MMUs, etc.) and exception behavior.

QEMU has great prospects to help OpenRISC development. The GNU toolchain regression suite could be run in less time both for bare-metal testing or Linux userspace level. It could also help evaluate the performance of OpenRISC hardware implementations that do not yet exist. An OpenRISC port in the upstream would also increase visibility of, interest in and hopefully use of, the platform.

7.4.4 Moving Upstream

The use of the term "upstream" in software development is related to the development community and repositories used by the authors of a project. These upstream repositories are typically the main source from which people obtain their copy of the project source code. The idea of submitting work upstream, or moving upstream, is to make your work on the project available from the primary source - the upstream servers.

In the case of the OpenRISC GNU toolchain ports, this would mean submitting ports for the binutils, GCC and GDB suites. The other ports to be submitted would be the newlib and uClibc ports. Last, but not least, would be the Linux kernel port for the OpenRISC architecture.

This is much easier said than done. The upstream development community must first check over the submitted architecture, and in each case it is a big deal to be adding support for a new architecture. Many things must be considered by those upstream, and every part of the code is reviewed. Large submissions such as a new architecture port for GCC and the Linux kernel typically require several submission attempts and also there must be a nominated maintainer.

The advantages of being upstream would be tremendous for the OpenRISC project. From the beginning of the project, toolchain issues have plagued the OpenRISC project, and many consider an architecture, if there isn't a reliable toolchain, to be a non-starter when considering it for use. Have the GNU projects upstream would increase the visibility and be a sign that the port has achieved the level of quality that is required to be accepted upstream. This would also increase the visibility of the project, and hopefully, the use of the OpenRISC platform. This would be one of the largest missing pieces in place of the truly open source and free to use microprocessor architecture project.

7.5 Toolchain

In large part, the OpenRISC 1000 GNU toolchain has been thoroughly upgraded and practically fully passes the regression test suite included with it. Some major

features still remain to be implemented.

7.5.1 Shared Object Support

Now that the Linux kernel port for OpenRISC has been largely reimplemented, there is the ability to make use of shared object libraries and dynamic linking.

At present none of this capability of the linker and loader is implemented. This, on the outset, appears to be a significant amount of work but would enable large amounts of software for Linux, which rely heavily on shared object support, to become compilable for the OpenRISC platform.

This work would require the GNU toolchain's binutils linker has the ability to link against shared objects implemented, and so too the loader in the C library providing user-space support for Linux, uClibc. Additionally, some software ABI choices have to be made regarding a register to use for a global offset table pointer register. Some discussion to this effect has been had by the development community but no solution has been specified or worked on as of yet. This would be an interesting set of work for an undergraduate thesis or post-graduate project in the field of operating systems.

7.5.2 Packaging and Releasing

The toolchain release is still yet to occur as of February 2011. It is anticipated this will happen with the not-too-distant future. Although the comments regarding moving upstream indicate that the developers aim to make the ports available from upstream repositories, an automated method of compilation and installation, or fully-compiled binary versions, will need to be made available for the most convenience. Additionally, as mentioned in the analysis section, the package management systems in all modern Linux distributions needs to be supported.

At present the toolchain release schedule is expected to begin with the first release, and hopefully occur each six months thereafter. The package-management distribution methods may not be implemented immediately, as this is typically a not-so straight forward process to prepare for, so precompiled binary distribution will, initially, be provided for download.

It is hoped that, by releasing every six months, fixes and upgrades will flow regularly and problems or contributed features can be delivered having been tested by developers in a realistic time frame. It should also indicate that the toolchain, despite its significantly improved implementation, still receives attention and has not been abandoned.

7.6 Software

This section will discuss potential areas for development of software for the OpenRISC 1000.

7.6.1 Operating Systems

The analysis section indicated the Linux kernel port has received a great deal of attention, however what was lacking, and what is arguably more appropriate for the available processor implementation, are real time operating systems (RTOS). Ports of operating systems such as Contiki and FreeRTOS, and an update of the existing RTEMS and eCos ports, would provide small embedded systems developers with the choice of adopting the OpenRISC platform.

A good choice would be the Contiki operating system, developed at the Swedish Institute of Computer Science (SICS). It is extremely lightweight and provides a solid networking capability. Alternatively, RTEMS with its BSD TCP/IP stack, which is already somewhat ported, could also be a good choice. Basic ports of these RTOSes, if made available, would provide developers with the ability to evaluate the OpenRISC platform for their embedded applications with greater ease.

7.6.2 Libraries

uClibc

The uClibc port, solely used for providing userspace support for Linux applications, has its architecture-specific portions of floating point support mostly completed but lacking testing.

Mentioned earlier in the toolchain section was the lack of shared object support. The dynamic object loader is required to be implemented in the operating system support library, in this case it is in the OpenRISC-specific parts of the uClibc port. This will be required in the event that shared object support is implemented in the linker and compiler.

EGLIBC

The standard C library, *glibc*, is typically not suited to use on embedded systems, as it compiles together all components non-optionally, and thus usually includes swathes of code that is of no interest which takes up precious memory. uClibc was forked off glibc to attempt to remedy this by making parts optional. It is not binary compatible with glibc - that is, programs must be recompiled against glibc to run on a system providing glibc. EGLIBC aims to be binary compatible with glibc, in so far as EGLIBC actually includes the required functions. The EGLIBC project also aims to closely track, or base its source on, the glibc project, providing new features or fixes as soon as they are included into glibc. It, however, also aims to include a configuration system for glibc, making it more suitable for embedded platforms.

EGLIBC has been chosen over standard glibc for use in the Debian Linux distribution, proving its maturity. The OpenRISC community could consider adding OR1K Linux userspace support to EGLIBC, instead of continuing to use uClibc. EGLIBC is still fairly new, but its widespread use shows it is already considered to be worthwhile.

newlib

As mentioned in the ORPSoC section, board specific builds in ORPSoC should have the compiler create board-specific executables made for them by simply adding some system parameters into newlib's libgloss. This would allow users of those board ports to generate simple bare-metal programs to help with board bring-up and development. A method of passing board-specific parameters to the start-up code has been discussed by the developers and it is expected a solution for this could be implemented shortly.

7.6.3 Applications

The analysis section made mention of the fact that there were few, if any, software applications, beyond library tests and bootloaders, available the OpenRISC repositories.

Hopefully this will change now the OpenRISC Linux kernel port has been improved, but the lack of shared library support will hinder many who will attempt to compile common programs for embedded systems that rely on basic shared libraries. Embedded Linux distributions, such as OpenWrt, cannot be compiled without shared library support. No sensible distribution deals with static executables as they would be too large if they were to include every library they needed. This renders any reasonably-featured Linux installation impractical.

It remains to be seen what interesting applications could be developed for OpenRISC, be it to run on the bare metal, an RTOS or the Linux kernel. Even an attempt at an exhaustive list would occupy too much space here. However, all of this work aims to make the platform usable for people to do exactly this - experiment with the platform and share their work.

7.7 Consideration of Successor Architecture

The issues with the OR1200 implementation raised earlier in this section caused the development community to consider a new OR1K implementation. In implementation terms, it would aim to be more modular and allow for alternate pipeline implementations, and have better synthesis characteristics, better testing and more comprehensive documentation. Considering such a development brought into focus the OR1K architectural issues outlined above. Suggestions of slight changes to the architecture specification, or extension of the instruction set, were put forward.

Alterations to the OR1K specification, no matter how minimal, will cause a breakage of all software for any existing implementations. Considering this, it's perhaps best to move on and base a new, non-binary compatible, architecture on OR1K and hope to add to it all of the best improvements embedded microprocessor architectures have enjoyed over the last twenty years, and perhaps some emerging features.

7.8 OpenRISC 2000

The OpenRISC 2000 architecture will be the successor architecture to OpenRISC 1000. It, too, will be an open source microprocessor architecture with at least one RTL implementations licensed under a non-viral open source license. Although the OpenRISC 1000 platform is now very solid and should still prove a viable option for use in development where a single-issue 32-bit RISC is appropriate. The OpenRISC 2000 project aims to define an updated architecture taking into consideration trends such as increased emphasis on parallel programming and execution, and the suitability of such implementations on ever growing FPGAs. With the platform largely targeting embedded use, it is intended to be able to play a role in systems requiring a stripped-down implementation, so to this end, and similar to OR1K, large portions of the OR2K architecture will be optional.

For true scalability, a single, well-written RTL implementation will need to have multiple pipeline implementations - one for low area, low speed, and one for high area, high speed uses. The focus on improved implementation should result in a better tested set of source that has better modularity. A focus on modularity and better defined interfaces between sections of the processor should also provide designers with improved accessibility and opportunity to customise sections to suit a different set of constraints. It is hoped that this approach to the implementation will make the project more attractive to designers, increasing its uptake and hopefully the participation in, and contribution to the project.

At present, the OpenRISC 2000 architecture specification is a work in progress in wiki format on the OpenCores website. Open discussion is occurring, and the developers are bringing it to the attention of those interested in computer architectures to comment on the developments so far.

7.8.1 Target

The platform's implementations will be, initially, FPGA-targeted. As FPGAs are growing in size, the architecture will aim to make multi-processor implementations as easy as possible. Although FPGA clock frequencies are staying relatively constrained, increases in performance can be achieved through the parallelisation of processing between CPUs. With this in mind, a suitable set of features of the architecture will need to be implemented.

7.8.2 Proposed Features

Although this is not a definite list of features that will feature in OR2K, it is what is up for discussion at present.

Atomic Transaction Primitives

Any multi-processor architecture will no doubt require synchronisation primitives in software. These are most reliably implemented when instruction-level atomic

transactions are possible. This is common in most architectures today, and the load-linked store-conditional pair of instructions are being proposed for OR2K.

Code Density with 16-bit instructions

Complementing the 32-bit instructions proposed for OR2K might be a set of 16-bit instructions implementing a lot of common functions available in 32-bit format, but with either limited register operands or immediate values. In order to take the most advantage of this, very common instructions, and likely streams of instructions should aim to have 16-bit variants. If chosen correctly, this could potentially make a lot of instructions map to 16-bit variants, greatly reducing, up to a maximum of 50%, the code size for an equivalent version compiled without 16-bit formats.

Whether to make this mandatory, or optional, between implementations is currently under discussion. Those arguing for making it optional point out that it will probably bloat the instruction fetch and decode stages, with extra logic required to handle non-32-bit word aligned instructions, and a whole additional set of opcodes, and therefor make the smallest-possible implementation large. On the other hand, potentially, with 16-bit support always enabled, perhaps the implementation overhead in transistors is less than those required to store the additional data for larger instructions.

No Delay Slot

OR1K used a delay slot to maximise use of fetched instructions in a pipeline after a branch. For ease of more complex implementation, perhaps at the expense of performance in some designs, the delay slot will no longer be used on OR2K. Although this is probably something that doesn't depend so much on architecture specification, and more on RTL and compiler implementation, it is a de facto architecture-specific feature.

Branch against immediate

Code analysis of OR1K indicated that a lot of branches and compares rarely used the entire immediate space to indicate the target, and as such, it could be possible to perhaps combine both in such a way as both instructions could usefully be combined into a single one.

Virtualisation

An easy way to design for virtualisation is to map all I/O to memory, including controls for CPU, interrupt and cache controls. The memory mapping hardware can then be used to manage both privileges and virtualisation.

Shared Co-processors, separate register files

Instead of requiring each co-processor unit, such as floating point units, to reside within, and deal with each CPU's register file directly, they should, instead, have their own register file, and the ISA should provide instructions from moving operands between the general purpose registers and the co-processor's. In this way, multiprocessor implementations can share the same co-processor hardware.

Core Configuration Register

Although this exists on OR1K, it lacks the ability to determine the precise implementation of the core, as there's no way to detect which class II instruction is supported apart from attempting to execute it and catching the trap. There should be a cleaner way of detecting the precise configuration of the core.

Caching

Multiprocessor implementations require careful cache synchronisation mechanisms, and must ensure cache snooping is implemented. Additionally, multi-level caching control interfaces should be included.

7.8.3 Testing and Development

There should be an emphasis on block-level testbenches, where practical. The development process will no doubt be collaborative, and should focus first on a clearly defined specification as well as a simulator, maybe QEMU, and compiler port. Perhaps these stages could occur first, and the final decision on which instructions should have 16-bit versions, or if any novel instructions can improve code-density without making implementation overly complex.

RTL implementation

The language of the model should remain as Verilog HDL. This is largely due to the greater support for the language by open source tools. In particular, the Verilator tool, which can create a fast cycle-accurate model from the Verilog RTL, can be very useful as it would make possible developing SystemC-based verification testbenches.

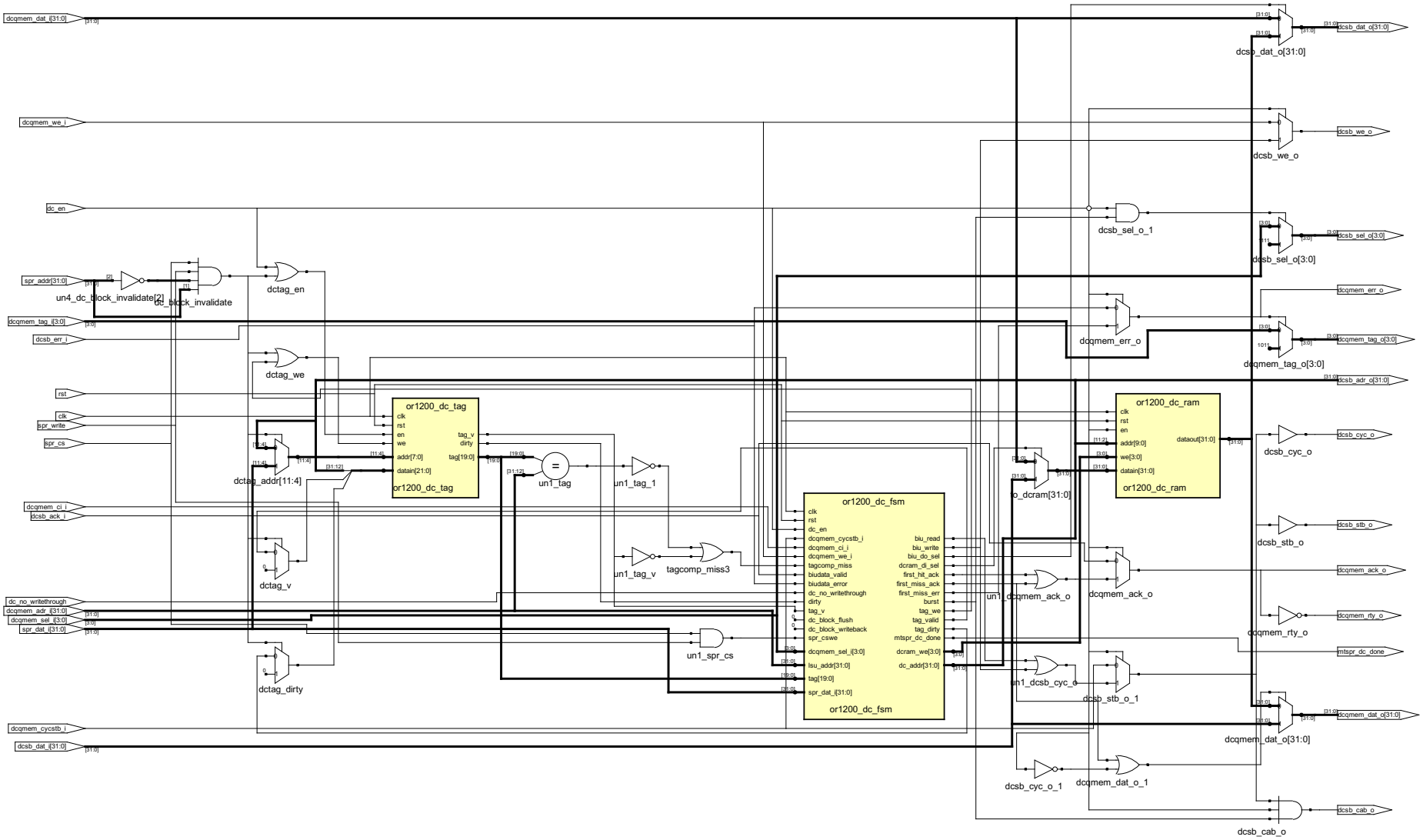
7.8.4 Summary

Although the OR2K project is only in its infancy, the architectural and implementation choices already on offer indicate that it will most likely be a considerable step on from OR1K. As work towards finalising the architecture specification continues, and the simulator and compiler are implemented, the architecture's final state will take form. It is hoped that by taking an open source approach to this project it will allow those with experience to participate and guide what could turn out to be a highly popular and useful platform.

Appendix A

Data Cache Synthesis Schematic

The following diagram was produced from the OR1200's RTL using the Synplify Pro Version E-2010.09A-1 targeted at Actel ProASIC 3 technology. The view is the RTL view of the `or1200_dc_top` module with its connections from the processor and to the bus bridge. It shows the structure of the data cache and its partitioning into the tag RAM, holding the physically tagged set addresses, the FSM block coordinating everything, and the actual data RAM.


dcqmem_dat_i[31:0]
dcqmem_we_i
dc_en
spr_addr[31:0]
un4_dc_block_invalidate[2]
dc_block_invalidate
dctag_en
dcqmem_tag_i[3:0]
dcsb_err_i
dctag_we
rst
clk
spr_write
spr_cs
dctag_addr[11:4]
or1200_dc_tag
un1_tag
un1_tag_1
dcqmem_ci_i
dcsb_ack_i
dctag_v
un1_tag_v
tagcomp_miss3
dc_no_writethrough
dcqmem_adr_i[31:0]
dcqmem_sel_i[3:0]
spr_dat_i[31:0]
un1_spr_cs
dctag_dirty
dcqmem_cycstb_i
dcsb_dat_i[31:0]
or1200_dc_fsm
dcsb_dat_o[31:0]
dcsb_we_o
dcsb_sel_o_1
dcsb_sel_o[3:0]
dcqmem_err_o
dcqmem_tag_o[3:0]
dcsb_adr_o[31:0]
or1200_dc_ram
dcsb_cyc_o
dcsb_stb_o
dcqmem_ack_o
dcqmem_rty_o
mtspr_dc_done
dcqmem_dat_o[31:0]
dcsb_cab_o
un1_dcqmem_ack_o
un1_dcsb_cyc_o
dcsb_stb_o_1
dcsb_cyc_o_1
dcqmem_dat_o_1

Bibliography

[1] Robert N. Noyce and Marcian E. Hoff. Jr., *A History of Microprocessor Development at Intel*. Micro, IEEE, Volume 1 Issue 1, 1981.

[2] Russo P.M , *VLSI Impact on Microprocessor Evolution, Usage, and System Design*. Solid State Circuits, IEEE Journal of, Volume 15 Issue 4, 1980

[3] Schoeffler, James D. , *Microprocessor Architecture*. Industrial Electronics and Control Instrumentation, IEEE Transactions on, Volume 22 Issue 3, 1975

[4] Agrawala, A.K. and Rauscher, T.G, *Microprogramming: Perspective and Status*. Computers, IEEE Transactions on, Volume 23 Issue 8, 1974

[5] Moore, Gordon E., *Cramming more components onto integrated circuits*. Electronics Magazine, Volume 38 Number 8, 1965

[6] Watson, I.M., *Comparison of commercially available software tools for microprocessor programming*. Proceedings of the IEEE, Volume 64 Number 6, 1976

[7] Cocke, John and Markstein, V., *The evolution of RISC technology at IBM*. IBM Journal of Research and Development, Volume 34 Number 1, 1990

[8] Patterson, D.A. and Sequin, C.H., *A VLSI RISC*. Computer, IEEE, Volume 15 Number 9, 1982

[9] Hennessy, J. et al., *A pipelined 32b NMOS microprocessor*. Solid-State Circuits Conference, Digest of Technical Papers, Volume XXVII, 1984

[10] Hennessy, J. et al., *MIPS-X: a 20-MIPS peak, 32-bit microprocessor with on-chip cache*. Solid-State Circuits, IEEE Journal of Volume 22, Issue 5 1987

[11] Garner, R.B. et al., *The scalable processor architecture (SPARC)*. Compcon Spring '88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers, 1988

[12] SPARC International Inc., *SPARC International, Inc. - Processors*. http://www.sparc.org/ads/chips.html, Accessed November, 2010

[13] Thomas, A.R.P et. al., *A 2nd Generation 32b RISC Processor with 4KByte Cache*. Solid-State Circuits Conference, 1989. ESSCIRC '89. Proceedings of the 15th European, 1989

[14] Jamil, T, *RISC versus CISC*. Potentials, IEEE Volume 14, Issue 3, 1995

[15] Carter, R, et. al., *Commodity clusters: performance comparison between PCs and workstations*. High Performance Distributed Computing, 1996., Proceedings of 5th IEEE International Symposium on, 1996

[16] Marsala, A. and Kanawati, B., *PowerPC Processors*. System Theory, 1994., Proceedings of the 26th Southeastern Symposium on, 1994

[17] Kozyrakis, C.E. and Patterson, D.A, *A new direction for computer architecture research*. Computer, IEEE, Volume 31, Issue 11, 1998

[18] Auer, D. and Buer, M, *A design flow for embedding the ARM processor in an ASIC*. ASIC Conference and Exhibit, Proceedings of the Eighth Annual IEEE International, 1995

[19] Szygenda, S.A. and Thompson, E.W, *Modeling and Digital Simulation for Design Verification and Diagnosis*. Computers, IEEE Transactions on, Volume C-25, Issue 12, 1976

[20] Edwards, M.D. and Forrest, J, *Hardware/software partitioning for performance enhancement*. Partitioning in Hardware-Software Codesigns, IEE Colloquium on, 1995

[21] MIPS Technologies, Inc., *Choosing an Intellectual Property Core*. http://www.mips.com/media/files/white-papers/Choosing_an_Intellectual_Property_Core.pdf 2002 Accessed Nov. 2010

[22] EDA Consortium, *EDA Consortium Market Statistics Service*. http://www.edac.org/mss/stats_mss.jsp Accessed Nov, 2010

[23] EDACafe, *Electronics IP Industry - An August 2010 Update*. http://www10.edacafe.com/nbc/articles/view_article.php?articleid=852230&page_no=7 Accessed Nov, 2010

[24] EETimes, *The death of microprocessors*. http://www.eetimes.com/design/other/4025001/The-death-of-microprocessors Accessed Nov, 2010

[25] ARM, *ARM Company Milestones*. http://www.arm.com/about/company-profile/milestones.php Accessed Nov, 2010

[26] Jesus M. Gonzalez-Barahona, *A brief history of open source software*. http://eu.conecta.it/paper/brief_history_open_source.html Accessed Nov, 2010

[27] Richard Stallman, *The GNU Operating System and the Free Software Movement*, Open Sources : Voices from the Open Source Revolution, http://oreilly.com/catalog/opensources/book/stallman.html Accessed Nov, 2010

[28] Free Software Foundation, Inc. *Free software is a matter of liberty, not price - Free Software Foundation - working together for free software.* http://www.fsf.org/about/ Accessed Nov, 2010

[29] Free Software Foundation, Inc *GNU General Public License, version 1.* http://www.gnu.org/licenses/old-licenses/gpl-1.0.html Accessed Nov, 2010

[30] Lawrence Rosen, *GPL, Legally Speaking.* Open Magazine, http://www.nusphere.com/products/library/gpl_0401openmag.pdf April, 2001, Accessed Nov, 2010

[31] Matt Asay, *The GPL: Understanding the License that Governs Linux.* http://www.novell.com/coolsolutions/feature/1532.html Accessed Nov, 2010

[32] Linus Torvalds, *RELEASE NOTES FOR LINUX v0.12.* http://www.kernel.org/pub/linux/kernel/Historic/old-versions/RELNOTES-0.12 Accessed Nov, 2010

[33] Mark Tarver, *The Problems of Open Source.* http://www.lambdassociates.org/blog/the_problems_of_open_source.htm Accessed Nov, 2010

[34] Free Software Foundation, Inc., *Various Licenses and Comments about Them - GNU Project - Free Software Foundation(FSF).* http://www.gnu.org/licenses/license-list.html Accessed Nov, 2010

[35] Free Software Foundation, Inc., *The Free Software Definition - GNU Project - Free Software Foundation(FSF).* http://www.gnu.org/philosophy/free-sw.html Accessed Nov, 2010

[36] Open Source Initiative, *The Open Source Definition.* http://www.opensource.org/docs/osd Accessed Nov, 2010

[37] Joseph Feller, et al., *Perspectives on Free and Open Source Software.* The MIT Press, 2005

[38] Wikipedia, the free encyclopedia, *Free and open source software.* http://en.wikipedia.org/wiki/FOSS Accessed Nov 22, 2010

[39] Eric Raymond, *A Brief History of Hackerdom.* http://www.catb.org/ esr/writings/cathedral-bazaar/hacker-history/ Acessed Nov, 2010

[40] Eric Raymond, *The Cathedral and the Bazaar.* http://www.catb.org/ esr/writings/cathedral-bazaar Acessed Nov, 2010

[41] Eric Raymond, *Goodbye, "free software"; hello, "open source".* http://www.catb.org/ esr/open-source.html Accessed Nov, 2010

[42] Richard Stallman, *Open Source vs. Free Software* http://perens.com/policy/open-source/ Accessed Nov, 2010

[43] Peter Clarke, *Free 32-bit processor core hits the Internet.* http://www.eetimes.com/electronics-news/4107438/Free-32-bit-processor-core-hits-the-Internet Accessed Nov, 2010

[44] Damjan Lampret, et al. *OpenRISC 1000 Architecture Manual.* http://opencores.org/ocsvn/openrisc/openrisc/trunk/docs/openrisc_arch.pdf Accessed Nov, 2010

[45] Peter Clarke, *Open source guru to demo Flextronics' 'free' processor.* http://www.eetimes.com/electronics-news/4120023/Open-source-guru-to-demo-Flextronics–free-processor Accessed Nov, 2010

[46] Peter Clarke, *OpenCores website, brand up for sale.* http://www.eetimes.com/electronics-news/4188480/OpenCores-website-brand-up-for-sale Accessed Nov, 2010

[47] Peter Clarke, *Swedish design house agrees to maintain OpenCores.* http://www.eetimes.com/electronics-news/4190299/Swedish-design-house-agrees-to-maintain-OpenCores Accessed Nov, 2010

[48] Horowitz, M. and Stark, D. and Alon, E., *Digital Circuit Design Trends.* Solid-State Circuits, IEEE Journal of, Volume 43, Issue 4, 2008

[49] Hofstee, P, et al., *A 1 GHz single-issue 64 b PowerPC processor.* Solid-State Circuits Conference, 2000. Digest of Technical Paper ISSCC. 2000 IEEE International, 2000

[50] Deleganes, D. and Douglas, J. and Kommandur, B and Patyra, M., *Designing a 3 GHz, 130nm, Pentium 4 processor.* VLSI Circuits Digest of Technical Papers, 2002. Symposium on, 2002

[51] Consumer Electronics Association, *Consumer Electronics and Electronic Components – Factory Sales, by Product Category.* http://www.allcountries.org/uscensus/1263_consumer_electronics_and_electronic_components_f Accessed Nov, 2010

[52] Consumer Electronics Association, *CONSUMER ELECTRONICS TO GROW IN 2010, ACCORDING TO CEA FORECAST.* http://www.ce.org/Press/CurrentNews/press_release_detail.asp?id=11861, Accessed Nov, 2010

[53] Masakatsu Nakai, et al., *Dynamic Voltage and Frequency Management for a Low-Power Embedded Microprocessor*. Solid-State Circuits, IEEE Journal of, Volume 40, Number 1, 2005

[54] Young Choi, *Analysis gives first look inside Apple's A4 processor*. http://eetimes.eu/en/analysis-gives-first-look-inside-apples-a4–processor.html?cmp_id=7&news_id=222901800 Accessed Dec, 2010

[55] Low, R., *Microprocessor trends: multicore, memory, and power developments*. http://www.embedded-computing.com/pdfs/Freescale.Sep05.pdf, Accessed Dec, 2010

[56] Royannez, P., et al., *Leakage Power Reduction Techniques applied to 90-nm SoC Application Processor*. Integrated Circuit Design and Technology, 2006. ICICDT , IEEE International Conference on 2006

[57] Horowitz, M., *Scaling, Power and the Future of CMOS*. IEEE VLSI Design, 6th International Conference on Embedded Systems., 20th International Conference on, 2007

[58] Electronics.ca Research Network, *Microprocessor Market to 2020 - MultiCore Processors, Laptops and Smartphones to Drive Growth*. http://www.electronics.ca/presscenter/articles/1341/1/Microprocessor-Market-to-2020—MultiCore-Processors-Laptops-and-Smartphones-to-Drive-Growth/Page1.html, Accessed Dec, 2010

[59] Embecosm Limited, *Embecosm Application Notes*. http://www.embecosm.com/download/application_notes.html, Accessed Dec, 2010

[60] Rich D'Addio, *Meansoffreedom (aka MOF)*. http://www.meansoffreedom.com, Accessed Dec, 2010

[61] Damjan Lampret, *new architecture OpenRISC 1000*. http://gcc.gnu.org/ml/gcc/1999-11n/msg00390.html, Accessed Dec, 2010

[62] Johan Rydberg, *Add support for or32-*-* and openrisc-*-**. http://sourceware.org/ml/binutils/2001-01/msg00072.html, Accessed Dec, 2010

[63] Embecosm Limited, *Howto: Porting newlib*. http://www.embecosm.com/appnotes/ean9/ean9-howto-newlib-1.0.html, Accessed Dec, 2010

[64] Steve Bush, *ARM reveals more details of Cortex A5 processor*. http://www.electronicsweekly.com/Articles/2009/10/22/47236/arm-reveals-more-details-of-cortex-a5-processor.htm Accessed March, 2011

[65] Freedom Defined, *OSHW*. http://freedomdefined.org/OSHW, Accessed Feb, 2011

[66] TAPR, *The TAPR Open Hardware License*. http://www.tapr.org/ohl.html, Accessed Feb, 2011

[67] Ars Technica, *TAPR introduces open-source hardware license, OSI skeptical*. http://arstechnica.com/old/content/2007/02/8911.ars, Accessed Feb, 2011

[68] Waqas Ahmed, *IMPLEMENTATION AND VERIFICATION OF A CPU SUBSYSTEM FOR MULTIMODE RF TRANSCEIVERS*. A thesis submitted to the faculty of Royal Institute of Technology (KTH), 2010

[69] Jidan El-Eryani, *Floating Point Unit*. http://opencores.org/ocsvn/fpu100/fpu100/trunk/doc/FPU_ Accessed Feb, 2011

[70] Jeremy Bennett, *Processor Verification using Open Source Tools and the GCC Regression Test Suite: A Case Study by Embecosm*. http://www.embecosm.com/articles/ear6/dvc-or1k-verification.pdf, Accessed Feb, 2011

[71] Igor Mohor, *SoC Debug Interface*, http://opencores.org/ocsvn/dbg_interface/dbg_interface/trunk/ Accessed Feb, 2011