Imperial College London

Telecom Paristech

Hardware-based Buffer Overflow Protection

Submitted for part completion of MEng Electronic and Information Engineering

Author: Jake Humphrey
Supervisor: Guillaume Duc

Abstract

Implementing security features in hardware can free the software programmer from worrying about them, leading in turn to simplified code. This can be especially useful when programming in low-level languages such as C, which provide minimal built-in protection.

This project report details the work undertaken modifying the Memory Management Unit (MMU) of an existing processor to implement protection against buffer overflow attacks, then produces some assembly code showing how firmware or an operating system kernel would make use of the protection. Finally, it analyses the effect of the added protection on hardware area and clock frequency.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Existing Solutions . . . 2
1.3 Project Aims . . . 2
1.4 Report Outline . . . 2

2 Research **3**
2.1 Comparison of Soft Processors . . . 3
2.1.1 OpenRISC . . . 3
2.1.2 LEON2/3/4 . . . 4
2.1.3 Lm32 . . . 5
2.1.4 Microblaze . . . 6
2.1.5 Nios II . . . 7
2.1.6 Cortex-M1 . . . 8
2.2 Choice and Justification . . . 8

3 Implementation **9**
3.1 SoCKit . . . 9
3.1.1 Building . . . 9
3.1.2 Running . . . 10
3.2 DE2 . . . 11
3.2.1 Building . . . 12
3.2.2 Running . . . 12
3.3 Hardware . . . 12
3.4 Software . . . 14
3.4.1 Before Hardware Modification . . . 14
3.4.2 After Hardware Modification . . . 15

4 Results **19**
4.1 Summary of Functionality Added . . . 19
4.2 Hardware Impact . . . 19
4.3 Closing Remarks . . . 20

Bibliography **23**

Chapter 1

Introduction

Computers today have access to large amounts of our personal data. It can often have disastrous effects on our personal and professional lives if the data is compromised. The problem with leaving security issues to software developers is that often multiple processes will be running on the same system, with a shared memory area, and a vulnerability in any one of the processes can potentially leak sensitive data, regardless of how secure the others are.

It is sometimes preferable to place security measures at the lower levels of execution, i.e. the hardware, kernel or operating system (OS). This restricts system security to a single point-of-failure, and can result in easier, faster, and more secure code.

1.1 Motivation

A **Buffer Overflow Attack** makes use of out-of-range array accesses to read or write memory locations intended to be inaccessible. They are typically associated with C and C++, which provide no built-in protection against array accesses falling outside the bounds of the array. Using a Buffer Overflow Attack, an attacker could, for example, corrupt the return address inside a function to point to and thus run arbitrary code.

The Heartbleed bug in OpenSSL[1] was an example of a vulnerability to this type of attack. An attacker was able to send a server a payload text string along with its length and have the server reply with the exact same payload. However, if the length supplied was longer than the actual length of the payload, the returned message would be appended with whatever followed the text string in memory.[1] This potentially allowed an attacker access to server private keys, user passwords, or really anything stored on the server's memory.

[1] `http://xkcd.com/1354/`

1.2 Existing Solutions

A potential solution would be to use other programming languages or the Standard Template Library container classes in C++, but for those who cannot afford the performance cut, there exist other protection strategies. For example, the use of **canary values** involves placing a known value after the buffer. Since buffer overflows are often carried out with such C functions as `strcpy`, they affect contiguous memory areas. A write would therefore alter the canary value, and thus the program can infer that a buffer overflow has occurred and can, for example, invalidate the memory area as a countermeasure.

The above solution works for writes, which corrupt the canary value, but would offer no protection against out-of-bounds reads, such as those making use of the Heartbleed vulnerability. Fortunately, the same idea can be extended to protect against both types of attack, though at significant cost, as will be explained.

1.3 Project Aims

One could imagine a system making use of the Memory Management Unit (MMU), whereby a buffer would be placed immediately before a page boundary, and the following page marked as invalid in the MMU. A buffer overflow, whether a read or a write, would then trigger a page fault, which could potentially allow the OS to resolve the situation.

The biggest downside to this method is that it requires an entire virtual memory page be invalidated (and thus unusable). On x86 processors, for example, this is 4KiB. It would be nice to have a smaller invalidatable section of memory in the MMU to use for this purpose.

This project investigates a method to allow the invalidation of smaller memory sections within the MMU, specifically by adding hardware registers to allow for memory invalidation at the cache block level; in x86 the cache block size is 64B. This allows use of the above protection system without excess wasted memory, however it adds extra silicon in the form of the hardware registers. This tradeoff is also examined during the project.

In addition, a small amount of code is written to both test the new hardware and demonstrate how firmware or software might make use of it.

1.4 Report Outline

This report is made up of 4 chapters:

Chapter 2 summarises the various processors that could be chosen on which to base the project, followed by the choice made and justifications.

Chapter 3 gives an account of the work done during the project, starting with setting up the processor to obtain a development environment, and going on to describe the hardware changes made and code written.

Chapter 4 describes the testing done and obtains figures for the area and frequency impact of the new hardware, followed by an analysis of what these figures mean in terms of the consequences of the new hardware on the design as a whole.

Chapter 2

Research

To carry out the project aims, a **Soft Processor**, a processor which is to some extent configurable, is needed. Specifically we need to be able to directly modify the MMU code. This chapter provides a comparison of the various Soft Processors available.

2.1 Comparison of Soft Processors

The following metrics will be taken into account:

- Bus Used
- Architecture, including Instruction Set Architecture (ISA)
- Cache
- Memory Management Unit (MMU)
- Protection/User Levels
- License
- Documentation
- Toolchain
- Maturity

Basic information was taken from the Wikipedia page on soft processors.[2] More detailed information unless otherwise cited was taken from the relevant company's website.[3][4][5][6][7][8]

2.1.1 OpenRISC

Developer
OpenCores

Bus Used
OpenRISC uses Wishbone, an open source specification by OpenCores.

Architecture
32/64-bit MIPS-like.

Cache
1–64kB for Instructions and Data each.

MMU
Virtual Memory support. Translation Lookaside Buffer (TLB) with page size 8kB and 16–256 entries.

User Levels
User/Supervisor Mode.

License
There are two main open source implementations available. A Verilog implementation, OpenRISC 1200, is released under LGPL. Another Verilog implementation, mor1kx, is available under a weak copy-left licence created specifically for it, the OHDL.

Documentation
There exists a document detailing the architecture specification.[9] Documentation for both implementations are also available at their respective Github pages.[10][11]

Toolchain
A Compiler (based on gcc) and Debugger (based on gdb) are available.[12]

Maturity
The OpenRISC 1000 specification has been under development since 2001. Version 1.0 was released in 2012, and Version 1.1 in 2014. The OpenRISC 1200 implementation is stable but not in active development, and is the widely used version.[13] Mor1kx is still in active development, but is stated to be more sophisticated.

2.1.2 LEON2/3/4

Developer
Aeroflex Gaisler

Bus Used
AMBA2

Architecture
32-bit. ISA based on SPARC-V8

Cache
LEON2: Instruction and Data caches. Set-associative with 1–4 sets and 1–64kB/set. LRR or LRU replacement. LEON3/4: 1–4 sets, 1–256 kB/set. Random, LRR or LRU replacement. LEON4 includes an optional L2 cache. 256-bit internal, 1-4 sets, 16kB–8MB.

MMU
SPARC Reference MMU (SRMMU) with configurable TLB

User Levels
The User and Supervisor modes of SPARC.

License
Gaisler state on their website that LEON2 is available from Atmel (AT697 and AT7913)[14], but the VHDL source code appears to be on Github.[15] LEON3 is available under GPL as part of the GRLIB library.[16] LEON4 is only available under a commercial license from Gaisler.

Documentation
LEON3[17] and LEON4[18] have documentation.

Toolchain
Being SPARC V8 conformant, compilers and kernels for SPARC V8 can be used with LEON processors. Cross compiler BCC.[19] gdb can be used for debugging.

Maturity
No information found.

2.1.3 Lm32

Developer
Lattice

Bus Used
Wishbone

Architecture
32-bit. ISA documentation is available.[20]

Cache
Configurable between no cache, 8kB I-cache, and 8kB I-cache and 8kB D-cache.

MMU
None.

User Levels
No separate modes.

License
LatticeMico32 is licensed under a free (IP) core license.

Documentation
Documentation is available from the LatticeSemi website, but registration is required.[21]

Toolchain
GCC supports the LM32 since version 4.5.0, and Binutils since 2.19. gdb and Newlib are also supported.

Maturity
Introduced in 2006.

2.1.4 Microblaze

Developer
Xilinx

Bus Used
Customisable between CoreConnect PLB, OPB, FSL, LMB, AXI4.

Architecture
In terms of its instruction set architecture, MicroBlaze is very similar to the RISC-based DLX architecture, which in turn is based upon the MIPS architecture.

Cache
2kB–64kB direct mapped write-through or write-back.

MMU
Optional. 4GB virtual memory, supports page sizes of 1kB, 4kB, 16kB, 64kB, 256kB, 1MB, 4MB, and 16MB. Page-replacement strategy controlled by software.

User Levels
User and privileged modes.

License
Proprietary. Available from Xilinx along with the EDK (Embedded Development Kit). A smaller, less functional version with limited target devices is available for free, but the full version costs $3000–$6000, depending on the license type. The license also restricts the use of Microblaze to Xilinx FPGAs. LGPL clones also exist in Verilog, VHDL, or MyHDL.[22]

Documentation
Rather comprehensive reference manual available.[23]

Toolchain
Xilinx provides the Vivado design suite, including an IDE, SDK, Simulator, Analyser, and Synthesiser. Support for Microblaze also exists in gcc as of version 4.6.

Maturity
Currently in version 8.50b, under development since 2002.

2.1.5 Nios II

Developer
Altera

Bus Used
Altera's own Avalon bus.

Architecture
32-bit RISC architecture, with support for up to 256 custom instructions. 3 different configurations are available: fast (f), standard (s), and economy (e).

Cache
Nios II/f has separate instruction and data caches of 512B to 64kB, while the Nios II/s has only an instruction cache, and the Nios II/e has none.

MMU
There is an optional MMU and MPU[1] for the Nios II/f. Default is 16-way set-associative cache, the default size depends on the target FPGA but is either 128 or 256 entries or 4 or 8kB. Replacement strategy is set by the software.

User Levels
The Nios II supports user and supervisor modes only when the design includes an MMU or MPU.

License
The License for the Nios II IP core is around $500. Currently the Altera website has a redirect loop and I am unable to find information on toolchain licenses.

Documentation
Both Hardware and Software development documentation are available.[24]

Toolchain
Qsys and Quartus for hardware, Altera's Embedded Design Suite for software, based on Eclipse and the GNU toolchain.

Maturity
In development since 2004, currently in version 13.1, released in 2013.

[1] Memory Protection Unit

2.1.6 Cortex-M1

Developer
ARM

Bus Used
ARM Corelink Interconnect

Architecture
Three-stage 32-bit RISC, ARMv6M ISA.

Cache
No cache, TCM only, up to 1MB each.

MMU
None.

User Levels
None.

License
Proprietary, must contact sales team for cost.

Documentation
No reference manual specifically for the M1. See the "resources" tab on the Cortex M1 home page for more information.[8]

Toolchain
An extension of Altera's Quartus II. Supported by the ARM microprocessor development kit for software development.

Maturity
No information found.

2.2 Choice and Justification

As well as having to contain an MMU, another requirement for the project is that the processor have open code for it. Just having a customisable size is not enough, since the aims include adding entirely new registers and logic to the MMU.

These restrictions eliminate all the above save OpenRISC and the LEON2/3 processors. The OpenRISC was chosen for the project because its architecture is closer to the MIPS and ARM architectures commonly in use today, while the SPARC architecture of the LEON processors would be more difficult to learn to program for.

Chapter 3

Implementation

This chapter gives an account of the work done during the project. As described in Chapter 2, the OpenRISC (**or1k**) was chosen as the target processor architecture. To allow for repeated instantiations and modifications of this hardware design, a Field Programmable Gate Array (FPGA) development board was used. Initially the Altera SoCKit[25] was chosen as the target device. However, even after extensive modifications, a working build was not obtained. The details of the attempt will nonetheless be detailed both as a record of work done and to advise future attempts.

Following this attempt, an Altera DE2 board[26] was used. This target device proved successful, and the hardware modifications and code written that are detailed in the following sections were run on this board (although in theory, the code is device-agnostic).

3.1 SoCKit

The initial project aim was to use the Altera SoCKit to instantiate and test the hardware. This section details the (ultimately unsuccessful) attempt to get it working.

3.1.1 Building

The first step was to build the OpenRISC processor and program it onto the board. For this a tool called **FuseSoC**[27] (formerly **OrpSoCv3**) was used. FuseSoC's readme describes it as a "package manager and a set of build tools for HDL [Hardware Description Language] code." Using this, it is simple[1] to build **mor1kx** (the OpenRISC implementation in question) for the SoCKit with the command `fusesoc build sockit`. This command invokes the relevant Quartus executables to build a netlist to be programmed.

However, the build did not work at first. A lot of searching revealed the problem to be a bug in FuseSoC's libraries. As shown below, the port names between two files did not agree. Patching this error allowed the build to continue.[2]

[1] In theory.

[2] Unfortunately for me, by the time I thought to submit a bug report or a pull request upstream, the bug had already been found and patched by the developers.

```
module avalon_to_wb_bridge \#(
        parameter DW = 32,      // Data width
        parameter AW = 32       // Address width
)(
        input              wb_clk_i,
        input              wb_rst_i,
        // Avalon Slave input
        input [AW-1:0]     s_av_address_i,
        input [DW/8-1:0]   s_av_byteenable_i,
        input              s_av_read_i,
        output [DW-1:0]    s_av_readdata_o,
        input [7:0]        s_av_burstcount_i,
        input              s_av_write_i,
        input [DW-1:0]     s_av_writedata_i,
        output             s_av_waitrequest_o,
        output             s_av_readdatavalid_o,
        // Wishbone Master Output
        output [AW-1:0]    wbm_adr_o,
        output [DW-1:0]    wbm_dat_o,
        output [DW/8-1:0]  wbm_sel_o,
        output             wbm_we_o,
        output             wbm_cyc_o,
        output             wbm_stb_o,
        output [2:0]       wbm_cti_o,
        output [1:0]       wbm_bte_o,
        input [DW-1:0]     wbm_dat_i,
        input              wbm_ack_i,
        input              wbm_err_i,
        input              wbm_rty_i
);
```

Figure 3.1: The declaration of the avalon_to_wb_bridge module[28]

Figure 3.1 gives the declaration of the `avalon_to_wb_bridge` module. If this is compared to an instantiation of that module, such as the one given in Figure 3.2, it can be seen that several port names are incorrect. This is likely due to the former file being changed without proper care being taken to update port names throughout the code.

Once the build succeeded, it could be programmed onto the board using the command `fusesoc pgm sockit`.

3.1.2 Running

After the processor was built and the board programmed, the next step was to load an object file onto the processor. For this a Linux image was used, compiled using the `or1k-elf-gcc` toolchain as detailed on the OpenCores website[30]. The resulting object file was simulated under the toolchain's simulator and found to run as expected.

To debug the running processor, including loading object files, the **Open On-chip Debugger** (**OpenOCD**) was used as detailed, once again, on the OpenCores website[31].

```verilog
wb_to_avalon_bridge #(
        .DW                         (32),
        .AW                         (32),
        .BURST_SUPPORT              (1)
) hps_ddr3_wb2avl_bridge (
        .wb_clk_i                   (wb_clk),
        .wb_rst_i                   (wb_rst),
        // Wishbone Slave Input
        .wb_adr_i                   (wb_m2s_hps_ddr3_adr),
        .wb_dat_i                   (wb_m2s_hps_ddr3_dat),
        .wb_sel_i                   (wb_m2s_hps_ddr3_sel),
        .wb_we_i                    (wb_m2s_hps_ddr3_we),
        .wb_cyc_i                   (wb_m2s_hps_ddr3_cyc),
        .wb_stb_i                   (wb_m2s_hps_ddr3_stb),
        .wb_cti_i                   (wb_m2s_hps_ddr3_cti),
        .wb_bte_i                   (wb_m2s_hps_ddr3_bte),
        .wb_dat_o                   (wb_s2m_hps_ddr3_dat),
        .wb_ack_o                   (wb_s2m_hps_ddr3_ack),
        .wb_err_o                   (wb_s2m_hps_ddr3_err),
        .wb_rty_o                   (wb_s2m_hps_ddr3_rty),
        // Avalon Master Output
        .m_av_address_o             (avm_hps_ddr3_address),
        .m_av_byteenable_o          (hps_0_f2h_sdram0_data_byteenable),
        .m_av_read_o                (hps_0_f2h_sdram0_data_read),
        .m_av_readdata_i            (hps_0_f2h_sdram0_data_readdata),
        .m_av_burstcount_o          (hps_0_f2h_sdram0_data_burstcount),
        .m_av_write_o               (hps_0_f2h_sdram0_data_write),
        .m_av_writedata_o           (hps_0_f2h_sdram0_data_writedata),
        .m_av_waitrequest_i         (hps_0_f2h_sdram0_data_waitrequest),
        .m_av_readdatavalid_i       (hps_0_f2h_sdram0_data_readdatavalid)
);
```

Figure 3.2: An example attempt to instantiate the avalon_to_wb_bridge module[29]

However, running OpenOCD for the SoCKit gave an error regarding mismatched clock speeds. Some searching revealed a discussion board[32] where the error was mentioned, which pointed to a patch for the issue[33].

Compiling a version of OpenOCD with this patch applied seemed to get around the issue, but unfortunately it would still not run.[3] After a meeting between the Author and Supervisor, it was decided to scrap the SoCKit route and work with the DE2 board, which had been used successfully in a previous project.

3.2 DE2

This section details the work done in setting up a development environment for the DE2 board. These steps were largely similar to those taken in Section 3.1.

[3]At time of writing, I've long since forgotten the exact issue, if indeed I ever worked out what it was. Let this be a lesson to write this kind of thing down somewhere.

3.2.1 Building

Once again, Fusesoc was used to build the processor. Appropriate configuration files were provided from a previous project by the project supervisor.

Support for the Cyclone II FPGA, as is used in the DE2, has been discontinued in later versions of Quartus, so version 13.0 needed to be installed for the build to work.

After moving the DE2 files to the appropriate Fusesoc directory, `fusesoc build de2` and `fusesoc pgm de2` worked as expected.

3.2.2 Running

Running OpenOCD proved far more successful this time, though in place of a Linux image, a Hello World program provided from the previous DE2 project was used.

3.3 Hardware

With a development environment finally obtained, the actual aim of the project could be implemented: the addition of hardware registers and logic to allow for invalidation of virtual memory of cache block size within the Data MMU (DMMU).

The first step was to add the new registers. The processor implementation being used was **mor1kx**[11]. This implementation was configured to have a cache block size of 32B, and a DMMU page size of 8kB. This results in $\frac{8kB}{32B} = 256$ cache lines (and therefore invalidation bits required) per page.

In addition, the DMMU has 1 way and 64 sets. With 64 potential pages in the DMMU, this results in a grand total of $64 \times 256 = 16384$ extra bits. Since the mor1kx word size is 32 bits, this could be achieved with the addition of $\frac{16384}{32} = 512$ extra 32-bit Special Purpose Registers (SPRs).

SPRs are a feature of the or1k architecture which are generally used to hold configurations and data specific to a unit within the processor. For example, the Translation Lookaside Buffer (TLB) within the DMMU is implemented as a series of “match” and “translate” SPRs. SPRs are accessible to software via the supervisor-mode instructions `mtspr` and `mfspr` (move to/from SPR).

Each unit has a “group” of 2048 SPR addresses allocated to it, and fortunately the last 512 addresses in the DMMU group were unused, allowing their allocation for the purposes of this project.

Figure 3.3 shows the instantiation of the new registers. Although the address width was given in terms of the set width[4], increasing the number of sets or ways in the DMMU now would cause the addresses of the new registers to overflow their allocated address space. Therefore, the DMMU size should henceforth be considered unconfigurable.

The next step was to add logic allowing the registers to be accessed. There are two ways this might occur:

[4] 6 bits for 64 sets, +8 for 256 bits per page, -5 for the 32 bits in each SPR

```
generate
   // DTLB invalidation registers
   mor1kx_simple_dpram_sclk
     #(
       .ADDR_WIDTH(OPTION_DMMU_SET_WIDTH+3),
       .DATA_WIDTH(OPTION_OPERAND_WIDTH)
       )
   dtlb_invalidation_regs
     (
      // Outputs
      .dout                            (dtlb_inval_dout),
      // Inputs
      .clk                             (clk),
      .raddr                           (dtlb_inval_addr),
      .re                              (1'b1),
      .waddr                           (dtlb_inval_addr),
      .we                              (dtlb_inval_we),
      .din                             (dtlb_inval_din)
      );
endgenerate
```

Figure 3.3: The code instantiating the new registers in the DMMU

- The `mtspr` and `mfspr` instructions allow the CPU to write to or read from the registers directly.
- Any address translation should first check to see if the appropriate invalidation bit is set, and if so trigger a page fault.

The `mtspr` and `mfspr` instructions take as argument a 16-bit SPR address; the first 5 bits encode the group number, and the last 11 bits the register number within that group. The DMMU SPRs belong to group 1, and we are looking for the last 512 registers[5]. Therefore, we are looking for an address of the format `0b0000 111x xxxx xxxx`.

Within the DMMU, all signals relating to these instructions are provided via the SPR bus, and it is via this bus that pertinent data must be sent back to the other sections of the CPU.

Figure 3.4 shows the preexisting code for the match and translate registers, alongside the new code added for the new registers. The `_spr_cs` signals indicate whether an incoming SPR address falls into one of the named SPRs.[6] If this signal is asserted, then the register address within the RAM block is taken from the lower 9 bits of the SPR address.

Since the software is going to be in charge of invalidating memory, the registers will only ever be written by an `mtspr` instruction. Therefore the RAM's input data bus reads directly from the data-in field of the SPR bus, and the write enable signal is asserted on an `mtspr` instruction.

[5] i.e. register numbers 1536 to 2047

[6] i.e. `dtlb_match_spr_cs` is asserted if the SPR address points to a match register in the DTLB.

Finally, the data from the RAM is returned to the SPR bus. The `_spr_cs_r` signals are the `_spr_cs` delayed by one cycle to align with the read cycle. When these are asserted, the appropriate RAM block is selected to read the output from.

Next comes the logic to read the appropriate bit during an address translation. When a virtual address comes in, the logic needs to extract the word and bit address to return the correct bit in the RAM. The virtual address should be decoded as follows:

The lower 5 bits `[4:0]` refer to the word inside the cache block and should be ignored. Bits `[12:5]` give the 8-bit index into the 256 bits allocated to each page. Bits `[18:13]` give the 6-bit set index.

In other words, the 14 bits `[18:5]` give the bit index within the RAM. Of these, the 9 upper bits `[18:10]` index one of the 512 registers, while the lower 5 `[9:5]` index the bit within the register.

Figure 3.5 shows the lines edited. A new signal was created to indicate when a load or store operation attempts to access invalid memory, then this signal is used alongside the existing access code violations as a condition for the page fault signal.

3.4 Software

On the software side, two programs were needed, one to test the MMU of the base implementation, and one to test the new added features of Section 3.3.

3.4.1 Before Hardware Modification

For the first program, the first step was to write the exception handlers, specifically the reset handler that gets called whenever the processor is reset or turned on, and the Data TLB (DTLB) miss exception handler that occurs whenever a Page Table Entry (PTE) needs to be loaded into the DTLB.

The exception handlers are shown in Figure 3.6. The reset handler initialises register 0 to a contain a value of 0, as is required by the or1k specification, then sets up registers 1 and 2, the stack and frame pointers respectively. It then jumps to the start of the program.

The DTLB miss exception handler is responsible for loading PTEs into the DTLB. In normal operation this would involve doing a page walk over the Page Table (PT). However, for this project it was not necessary to emulate this behaviour. A PT was not used, and instead the handler simply creates a PTE corresponding to the identity transformation[7] on the fly.

To set up the relevant entries in the DTLB, the exception handler must fill the match register in the appropriate set with the Virtual Page Number (VPN), and fill the corresponding translate register with the Physical Page Number (PPN). In addition, the match register requires bit 0 to be set to be valid, and the translate register has several protection bits saying whether the corresponding page can be read or written in supervisor or user mode. For the purposes of this test program, these can all be enabled.

[7] i.e. Virtual Address = Physical Address

In Figure 3.6, the exception handler first obtains the Effective Address[8] (EA) whose translation triggered the exception. The page number of this address is made up of the top 19 bits, so the bottom 13 bits are cleared. The set number is found by taking bits 18 to 13 of the EA. The result is then used as an offset for the `mtspr` instruction, which takes a register value and ORs it with an immediate to obtain the SPR address.

After the exception handlers, the main program was written. All this had to do was store some data in memory, then activate the DMMU and attempt to retrieve the data. Figure 3.7 shows the code written to do this. The DMMU is enabled by setting bit 5 of an SPR called the Status Register (SR). The test value is loaded into register 15, and by checking the contents of this register with OpenOCD, it can be confirmed that the program worked as expected.

3.4.2 After Hardware Modification

The second program was written after the hardware modifications detailed in Section 3.3. In addition to the old exception handlers, which remain unchanged, the Page Fault exception handler now writes a distinctive value to a register and loops indefinitely. In this way it is obvious when a page fault occurs.

The main program stores a test value to memory as before, then sets the appropriate bit in one the new registers to mark the whole cache line as invalid. It then activates the DMMU and attempts to access the memory location, which should trigger a page fault.

The code is given in Figure 3.8. As detailed in Section 3.3, the register index is given by bits 18 to 10 of the EA, and the bit index by bits 9 to 5.

[8] For all intents and purposes, this is identical to a virtual address, but the or1k specification distinguishes between them

```
always @(*) begin
   // Snip block initialisations
   dtlb_inval_we = dtlb_inval_spr_cs & spr_bus_we_i;
   // Snip other write-enable signals
end

assign dtlb_match_spr_cs = spr_bus_stb_i
                         & (spr_bus_addr_i[15:11] == 5'd1)
                         & ^spr_bus_addr_i[10:9] & !spr_bus_addr_i[7];
assign dtlb_trans_spr_cs = spr_bus_stb_i
                         & (spr_bus_addr_i[15:11] == 5'd1)
                         & ^spr_bus_addr_i[10:9] & spr_bus_addr_i[7];
assign dtlb_inval_spr_cs = spr_bus_stb_i
                           & (spr_bus_addr_i[15:11] == 5'd1)
                           & &spr_bus_addr_i[10:9];

assign dtlb_match_addr = dtlb_match_spr_cs ?
                         spr_bus_addr_i[OPTION_DMMU_SET_WIDTH-1:0] :
                         virt_addr_i[13+(OPTION_DMMU_SET_WIDTH-1):13];
assign dtlb_trans_addr = dtlb_trans_spr_cs ?
                         spr_bus_addr_i[OPTION_DMMU_SET_WIDTH-1:0] :
                         virt_addr_i[13+(OPTION_DMMU_SET_WIDTH-1):13];
assign dtlb_inval_addr = dtlb_inval_spr_cs ?
                           spr_bus_addr_i[OPTION_DMMU_SET_WIDTH+3-1:0] :
                           virt_addr_i[18:10];

assign dtlb_match_din = dtlb_match_reload_we ? dtlb_match_reload_din :
                        spr_bus_dat_i;
assign dtlb_trans_din = dtlb_trans_reload_we ? dtlb_trans_reload_din :
                        spr_bus_dat_i;
assign dtlb_inval_din = spr_bus_dat_i;

// Snip area translate buffer signals

assign spr_bus_dat_o =
      dtlb_inval_spr_cs_r ? dtlb_inval_dout :
      dtlb_match_spr_cs_r ? dtlb_match_dout[spr_way_idx_r] :
      dtlb_trans_spr_cs_r ? dtlb_trans_dout[spr_way_idx_r] :
      dmmucr_spr_cs_r ? dmmucr : 0;
```

Figure 3.4: New logic signals to present the new registers to the SPR bus.

```
   assign dtlb_inval_access = (op_store_i || op_load_i)
                              && dtlb_inval_dout[virt_addr_i[9:5]];

   assign pagefault_o = ((supervisor_mode_i ?
                        !swe & op_store_i || !sre & op_load_i :
                        !uwe & op_store_i || !ure & op_load_i)
                        || dtlb_inval_access) &
                        !tlb_reload_busy_o;
```

Figure 3.5: New logic signals to check for invalidated memory during address translation.

```
        .global _start
        .org 0x100
reset:
        l.andi  r0, r0, 0
        l.movhi r1, hi(_stack)
        l.ori   r1, r1, lo(_start)
        l.or    r2, r0, r1

        l.j     _start
        l.nop

        .org 0x900
dtlbms:
# virt = phys
        l.mfspr r23, r0, 0x30           # r23 = EA
        l.movhi r25, 0xffff
        l.ori   r25, r25, 0xe000        # r25 = 0xfffe000
        l.and   r25, r25, r23           # r25 = EA[31:13] = VPN/PPN

        l.movhi r27, 0x0007
        l.ori   r27, r27, 0xe000        # r27 = 0x0007e000
        l.and   r27, r27, r23
        l.srli  r27, r27, 13            # r27 = EA[18:13] = set no.

        l.ori   r29, r25, 0x0001        # set valid bit
        l.mtspr r27, r29, 0x0a00        # ->match[set no.]
        l.ori   r29, r25, 0x03c0        # set all permissions
        l.mtspr r27, r29, 0x0a80        # ->trans[set no.]

        l.rfe                           # return from exception
```

Figure 3.6: The Reset and DTLB Miss exception handlers

```
# store test value in memory
l.movhi r13, 0xc0d1              #
l.ori   r13, r13, 0xf1ed         # r13 = 0xcod1f1ed
l.ori   r15, r0, 0x4444          # r15 = 0x4444

l.sw    0(r15), r13

# activate mmu
l.mfspr r13, r0, 0x0011          # r13 <- sr
l.ori   r13, r13, 0x20           # enable bit 5 (dmmu)
l.mtspr r0, r13, 0x0011          # r13 -> sr

# try to load memory location
l.lwz   r15, 0(r15)
```

Figure 3.7: Main function of first test program

```
# store test value in memory
l.movhi r13, 0xc0d1              #
l.ori   r13, r13, 0xf1ed         # r13 = 0xcod1f1ed
l.ori   r15, r0, 0x4444          # r15 = 0x4444

l.sw    0(r15), r13

# mark cache line as invalid in dmmu
l.ori   r19, r0, 1
l.andi  r17, r15, 0x1f           # r17 = bit no = EA[9:5]
l.sll   r19, r19, r17            # r19 = one-hot(r17)

l.movhi r17, 0x0007
l.ori   r17, r17, 0xfb00
l.and   r17, r15, r17
l.srli  r17, r17, 10             # r17 = reg no = EA[18:10]

l.mfspr r21, r17, 0x0e00
l.or    r21, r19, r21
l.sw    4(r15), r21
l.mtspr r17, r21, 0x0e00         # set prot reg

# activate mmu
l.mfspr r13, r0, 0x0011          # r13 <- sr
l.ori   r13, r13, 0x20           # enable bit 5 (dmmu)
l.mtspr r0, r13, 0x0011          # r13 -> sr

# try to load memory location
l.lwz   r15, 0(r15)
```

Figure 3.8: Main function of second test program

Chapter 4

Results

This chapter gives a summary of what was achieved during the project and an analysis of the new hardware, including its impact on area and frequency, followed by some closing remarks.

4.1 Summary of Functionality Added

The work done in this project modified the Data Memory Management Unit (DMMU) of the OpenRISC soft processor, specifically the `mor1kx` implementation. 512 extra 32-bit registers were added to allow for invalidation of individual 32B cache blocks in the virtual memory. This is an improvement over the original system, which only allows for invalidation of entire 8kB pages.

The software running on the processor is then able to invalidate a cache block in virtual memory by setting one of the bits in these registers, the index of which is based on the set number of the page in and the cache block number within the page. Thereafter, when an invalid cache block is accessed, the hardware automatically signals a page fault, dropping the processor into the exception handler. From here, control can potentially be given to the operating system to avoid a buffer overflow, or terminate the offending process.

4.2 Hardware Impact

Figures 4.1 and 4.3 depict the output from Quartus's Fitter stage. It is interesting to note that the total number of Logic Elements (LEs) used has decreased slightly from 11,313 to 10,855, which corresponds to about 4%. In addition, the total number of registers has decreased from 6,248 to 5,472, a 12% drop. This is in spite of the fact that only extra signals have been added, and nothing has been removed.

It's possible this drop is indicative of an error introduced in the new design: extra hardware could unintentionally short-circuit existing logic. However, the logic added was rather simple; a more likely explanation is this: The DMMU is often on the critical path of the CPU, meaning added logic could violate timing constraints. In addition, fitting is hard; Quartus's fitting algorithm is lazy, meaning it stops optimising once it passes the timing requirements. The added logic to the DMMU could therefore have caused the Fitter to redouble its optimisation efforts, resulting in a smaller design overall.

The other notable change is that the total memory bits used has increased from 46,080 to 64,264, an increase of 16,384, or nearly 40%. This is exactly as expected, since it corresponds to the 512 extra 32-bit registers added.

Figures 4.2 and 4.4 depict the output from Quartus's Static Timing Analysis (STA) stage. `clk[0]` is the SDRAM clock and `clk[1]` is the Wishbone bus clock, both based off the input clock signal from the board.

The STA shows that the SDRAM has sped up 4% and the Wishbone bus has slowed down by 6%. As before, these results could be attributed to extra optimisation, so it is difficult to pinpoint the exact effect of the added logic on frequency, but at the very least it is not excessive.

4.3 Closing Remarks

In the end the aims of the project were implemented. It is unfortunate that the more modern SoCKit with its Cyclone V FPGA could not be used, and that attempts to do so detracted from the final acheivements of the project. To sum up, the works created by this project are:

- A modified DMMU to be used in the mor1kx processor,
- An assembly program that uses the new hardware.

This program also served to test for the hardware. Ideally, though, a test suite would be created to ensure that functionality is otherwise unchanged from before.

There is much more work that could follow from this project. First of all more testing is required as stated. In addition more work is needed to ascertain the benefits of the hardware-based approach over traditional ones.

Further related work could also include a modification to the Linux kernel that would allow a distribution running on the mor1kx processor to use the added hardware for memory management.

Fitter Summary	
Fitter Status	Successful - Thu Jun 18 16:18:53 2015
Quartus II 32-bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	de2
Top-level Entity Name	orpsoc_top
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	11,313 / 33,216 (34 %)
Total combinational functions	9,401 / 33,216 (28 %)
Dedicated logic registers	6,248 / 33,216 (19 %)
Total registers	6248
Total pins	60 / 475 (13 %)
Total virtual pins	0
Total memory bits	46,080 / 483,840 (10 %)
Embedded Multiplier 9-bit elements	6 / 70 (9 %)
Total PLLs	1 / 4 (25 %)

Figure 4.1: Fitter Summary for the base mor1kx provided by OpenRISC

Slow Model Fmax Summary

Fmax	Restricted Fmax	Clock Name	Note
56.54 MHz	56.54 MHz	clkgen0|pll0|altpll_component|pll|clk[1]	
119.65 MHz	119.65 MHz	clkgen0|pll0|altpll_component|pll|clk[0]	

Figure 4.2: Static Timing Analysis for the base mor1kx provided by OpenRISC

```
+--------------------------------------------------------------------------------------+
; Fitter Summary                                                                       ;
+------------------------------------+-------------------------------------------------+
; Fitter Status                      ; Successful - Thu Jun 18 16:43:41 2015           ;
; Quartus II 32-bit Version          ; 13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition ;
; Revision Name                      ; de2                                             ;
; Top-level Entity Name              ; orpsoc_top                                      ;
; Family                             ; Cyclone II                                      ;
; Device                             ; EP2C35F672C6                                    ;
; Timing Models                      ; Final                                           ;
; Total logic elements               ; 10,855 / 33,216 ( 33 % )                        ;
;     Total combinational functions  ; 9,440 / 33,216 ( 28 % )                         ;
;     Dedicated logic registers      ; 5,472 / 33,216 ( 16 % )                         ;
; Total registers                    ; 5472                                            ;
; Total pins                         ; 60 / 475 ( 13 % )                               ;
; Total virtual pins                 ; 0                                               ;
; Total memory bits                  ; 62,464 / 483,840 ( 13 % )                       ;
; Embedded Multiplier 9-bit elements ; 6 / 70 ( 9 % )                                  ;
; Total PLLs                         ; 1 / 4 ( 25 % )                                  ;
+------------------------------------+-------------------------------------------------+
```

Figure 4.3: Fitter Summary for the modified mor1kx with added registers and logic

```
+--------------------------------------------------------------------------------+
; Slow Model Fmax Summary                                                        ;
+------------+-----------------+--------------------------------------------+------+
; Fmax       ; Restricted Fmax ; Clock Name                                 ; Note ;
+------------+-----------------+--------------------------------------------+------+
; 53.3 MHz   ; 53.3 MHz        ; clkgen0|pll0|altpll_component|pll|clk[1] ;      ;
; 124.72 MHz ; 124.72 MHz      ; clkgen0|pll0|altpll_component|pll|clk[0] ;      ;
+------------+-----------------+--------------------------------------------+------+
```

Figure 4.4: Static Timing Analysis for the modified mor1kx with added registers and logic

Bibliography

[1] Heartbleed Website
heartbleed.com
Fetched 2015-06-15.

[2] Soft Processors on Wikipedia
en.wikipedia.org/wiki/Soft_microprocessor
Fetched 2015-06-19.

[3] Home page of or1k
http://opencores.org/or1k/Main_Page
Fetched 2015-06-19.

[4] Gaisler website
gaisler.com
Fetched 2015-06-19.

[5] LatticeSemi IP home page
www.latticesemi.com/Products/DesignSoftwareAndIP.aspx
Fetched 2015-06-19.

[6] Microblaze home page
www.xilinx.com/tools/microblaze.htm
Fetched 2015-06-19.

[7] Nios II home page
www.altera.com/products/processors/overview.html
Fetched 2015-06-19.

[8] Cortex M1 home page
www.arm.com/products/processors/cortex-m/cortex-m1.php
Fetched 2015-06-19.

[9] OpenRISC 1000 Architecture Manual
github.com/openrisc/doc/blob/master/openrisc-arch-1.1-rev0.pdf?raw=
true
Fetched 2015-06-19.

[10] Official or1200 repository on Github
github.com/openrisc/or1200
Fetched 2015-06-19.

[11] Official mor1kx repository on Github
github.com/openrisc/mor1kx
Fetched 2015-06-17.

[12] OpenRISC GNU tool chain
opencores.org/or1k/OpenRISC_GNU_tool_chain#Tools
Fetched 2015-06-19.

[13] OpenRISC project overview
openrisc.github.io
Fetched 2015-06-19.

[14] Aeroflex Gaisler's list of available processors
www.gaisler.com/index.php/products/processors
Fetched 2015-06-19.

[15] LEON2 source code on Github
github.com/Galland/LEON2
Fetched 2015-06-19.

[16] GRLIB library on Gaisler website
www.gaisler.com/index.php/products/ipcores/soclibrary
Fetched 2015-06-19.

[17] LEON3 Documentation
www.gaisler.com/doc/leon3_product_sheet.pdf
Fetched 2015-06-19.

[18] LEON4 Documentation
www.gaisler.com/doc/LEON4_32-bit_processor_core.pdf
Fetched 2015-06-19.

[19] BCC SPARCV8 cross-compiler
www.gaisler.com/index.php/products?option=com_content&task=view&id=147
Fetched 2015-06-19.

[20] LatticeMico32 ISA documentation
sourceware.org/cgen/gen-doc/lm32-insn.html
Fetched 2015-06-19.

[21] LatticeMico32 documentation
www.latticesemi.com/en/Products/DesignSoftwareAndIP/
IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx#

_B63AC1B3E7604B0DA4F80789A0B69A31
Fetched 2015-06-19.

[22] List of Microblaze clones on Wikipedia
en.wikipedia.org/wiki/MicroBlaze#Clones
Fetched 2015-06-19.

[23] Microblaze documentation
xilinx.com/support/documentation/sw_manuals/xilinx11/mb_ref_guide.pdf
Fetched 2015-06-19.

[24] Nios II documentation
https://www.altera.com/support/literature/lit-nio2.html
Fetched 2015-06-19.

[25] SoCKit page on Distributor site
parts.arrow.com/item/detail/arrow-development-tools/sockit
Fetched 2015-06-16.

[26] DE2 page on Altera site
wl.altera.com/education/univ/materials/boards/de2/unv-de2-board.html
Fetched 2015-06-17.

[27] FuseSoC repository on Github
github.com/olofk/fusesoc
Fetched 2015-06-16.

[28] OrpSoC-cores repository on Github
github.com/openrisc/orpsoc-cores/blob/14fb5aa0d96aea7e2594d0bb42195c9477dd33f8/
cores/wb_avalon_bridge/verilog/avalon_to_wb_bridge.v
Fetched 2015-06-16.

[29] OrpSoC-cores repository on Github
https://github.com/openrisc/orpsoc-cores/blob/
70b775e2ec2e53d15e3f595336e05053ba81f139/systems/sockit/rtl/verilog/
orpsoc_top.v#L587
Fetched 2015-06-16.

[30] Linux info on the OpenCores website
opencores.org/or1k/Linux
Fetched 2015-06-16.

[31] Debugging info on the OpenCores website
opencores.org/or1k/Debugging_physical_targets_(FPGA/ASIC)
Fetched 2015-06-16.

[32] Discussion of bug with OpenOCD and the SoCKit
`comments.gmane.org/gmane.comp.debugging.openocd.devel/26474`
Fetched 2015-06-16.

[33] OpenOCD patch on Github
`github.com/fjullien/openOCD/commit/0c834568e020ed3b61a055086d9d142f03860e0f`
Fetched 2015-06-16.