FPGA-based RISC Microprocessor and Compiler (Rev. 2.10)

PRCO304 - Final Stage Computing Project

Ben Lancaster 10424877

March 29, 2018

Revision History

Table 1: Document revisions.

Date	Version	Changes
22/03/2018	2.10	Add section 4.3.2 AST Generation.
15/03/2018	2.00	Add section 4.3 Text Grammar.
11/03/2018	1.00	Initial section outline.

Abstract

ben

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern computing and electronics equipment, like function generators, oscilloscopes, and spectrum analysers, use FPGAs to implement their compute intensive logic. These FPGAs are often accompanied by a small, low-cost, microprocessor to supervise and provide interfaces to external peripherals.

The aim of this project is to implement this side-microprocessor into the FPGA to save on BOM costs, PCB space, and power costs, which contribute to higher development and product costs. While savings can be made by the lack of side microprocessor, the product may need a larger FPGA to accommodate the embedded microprocessor. The project will produce a small, soft-core, CPU design and compiler. Although there is no direct client in this project, I believe this project will produce an attractive product for FPGA-based product designers wishing to employ an embedded processor solution.

1.1 Background

1.1.1 Current Implementations

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

1.2 Project Overview

This project aims to provide an efficient and cost-saving alternative for board and hardware product designers utilising side-microprocessors by designing, implementing, and demonstrating, a small, portable, FPGA processor core design to be used in-place of the side-microprocessor.

The processor core will implement it's own processor and instruction set architecture and so a compiler and assembler will also be provided so that software code can easily be executed on the processor.

1.2.1 Core Deliverables

These core (C) deliverables are the base requirement for the project to be released in a functional and worthwhile state.

C1. To improve my knowledge and experience of FPGA development, processor architecture, compilers, and embedded systems engineering.

C2. To build a working and operational soft-core processor core capable of performing simple tasks.

C3. Implementation of the soft-core processor design on real hardware.

C4. To provide product designers with an affordable alternative to a side-microprocessor in their FPGA-based products.

C5. To provide a technical processor reference guide and specification for the embedded core.

1.2.2 Extended Deliverables

These extended (E) deliverables may not be achievable in the time frame specific in section 2.1 as they may require extra time to design and implement, require more experience or skill, or require resources currently unattainable.

E1. To provide embedded products a convenient solution to in-field updating.

E2. To provide easy interfacing between the FPGA design and the embedded core.

E3. GCC/LLVM/8CC compiler backend for C programming.

E4. Wishbone interface for easier modularity and inter-module communication.

E5. Multi-core design with Wishbone (2).

E6. Configurable build options (register/bus widths, optimisations/pipelining, user/privileged mode to support modern operating systems).

E7. Memory management modules to provide protected and virtual memory lookup tables.

1.3 Legal and Ethical Considerations

1.3.1 Privacy

The PRCO304 processor will be able to read and write to all data passing through it and control all connected peripherals (such as UARTs, SDRAMs, and SD Cards). The processor does not track or store usage behaviour, instructions and their frequency, memory contents, or timing statistics, or any other usage metric.

1.3.2 Fit for Purpose

The PRCO304 processor is not designed to run general purpose operating systems, such as Linux or embedded RTOS systems. All memory devices attached to the FPGA are fully accessible to the processor core and instructions/programs running through it, meaning that operating systems or secure applications storing private and sensitive information is not protected by modern processor features such as privilege modes and virtual memory sections. The processor lacks common components required to run modern operating systems, such as a memory management unit (MMU) and privilege modes, and so should not be run on the processor.

The PRCO304 processor is not designed to run in high-reliability or safety-critical environments that require established safety standards, such as the UK Defence Standard 00-56 (**?**) and IEC 61508 (**?**).

The PRCO304 processor, by design, should be used as a replacement for a simple micro-controller accompanying a main processing module.

1.3.3 Third-party Libraries

This project uses only 1 external library for the processor core's universal asynchronous receiver-transmitter (UART) module that does not depend on any other libraries. This allows me to guarantee that: the project rights are secure; and application behaviour is well-defined and predictable (no exploits introduced/injected from external libraries). The UART module does feature a large first-in-first-out (FIFO) buffer for temporary storage of in- and out- going messages. This FIFO is internal to the FPGA design and so is protected from external viewing/modification by probing the board in which the core is running on.

The compiler sub-project does not use any external library dependencies, does not record telemetry or usage statistics, and does not require an internet connection to run.

1.3.4 Generated Code

The code generated by the compiler is **not guaranteed** to:

- **Produce code for secure environments**. The compiler will not randomise, obfuscate, or split-up and spread, output code. Output machine code will be in a predictable format (global variables in low-memory, instruction memory in middle-memory, and stack memory in high-memory) making the binary easily subject to reverse-engineering and modification.
- **Produce constant time executable code for expressions**. For example, the compiler output for an *if* statement may implicitly vary depending on it's condition expression, which may have been optimised out, constant-folded, or without-optimisation. This also applies for user code aiming to create reliable and accurate time delay loops; although the processor does not perform optimisations such as instruction caching or branch prediction, access to memory and ALU operations may vary in time, resulting in unreliable instruction times.

Chapter 2

Project Management

2.1 Time Management

2.2 Version Control

Version control will be utilised to improve work-flow, reference and review code changes, and protect the project from data loss and corruption. GitHub, a git hosting provider, will be utilised to host all project files, including documentation and design files.

The repository can be found here: https://github.com/bendl/prco304.

2.3 Method of Approach

Development of the **core** and **compiler** will be done in separate stages of the project (see section 2.1). The two deliverables will be split into 2 sub-projects. Both sub-projects will employ the **Agile development process**, using Agile's sprints to split up tasks into sub-tasks and Agile's scrums to discuss progress, features, and changes. This technique allows revisiting of tasks to tweak and iterate over their implementation which will be key when for incrementally adding features to both sub-projects, for example, adding to the core's ALU module to add conditional branching, or adding new instructions to the core's decoder module.

2.4 Requirements

2.5 Resources and Dependencies

For the first half of the development cycle, the core can be developed and verified using the Verilog simulator and test suite, **Verilator**, and VHDL and Verilog simulator, **iSim**.

The second half of development will require deploying and debugging on real hardware. This will require an FPGA development kit. To better emulate customer products, the development kit should feature common components such as LEDs, GPIO, USB interface, flash-based storage and memory, and optionally an analogue audio output port. The low-middle range of FPGA devices the project is targeting is the popular and affordable yet feature rich Spartan-6 and Artix-7 FPGAs. From my placement, I have gained experience in Xilinx FPGAs and so will be targeting them for this project to reduce risk and development time.

The following FPGA development kits are suitable for this project:

1. MiniSpartan6+ - Scarab Hardware - *$*79 (already owned) (**?**). The MiniSpartan6+ features a Spartan-6 XC6SLX9 FPGA, 8 LEDs, 2 digital and analogue headers, FT2232 FTDI USB to JTAG, 64Mb SPI flash memory, 32MB SDRAM, an audio output jack, and a MicroSD socket.

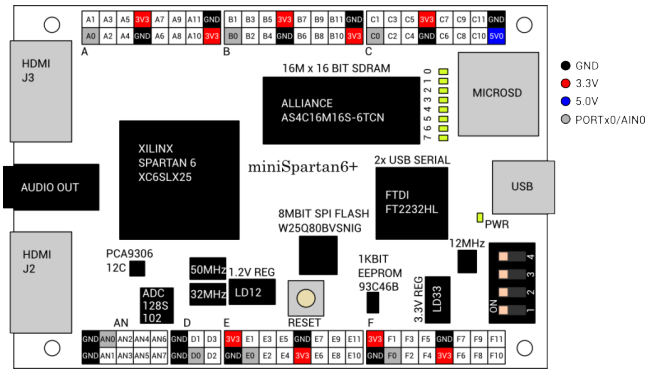

Figure 2.1: Scarab Hardware MiniSpartan6+ board layout.

2. Arty Artix-7 FPGA Development Board - Digilent - *$*100 (**?**). The Arty development board features a larger Artix-35T FPGA with over 20x the number of logic cells and block memory compared to the LX9 in the MiniSpartan6+. The board components include 256MB DDR3 RAM, 16MBx4 SPI flash memory, USB-JTAG, 8 LEDs (4 of which are RGB), 4 switches, 4 buttons, and multiple Pmod connectors.

   The greater number of IO options and larger FPGA make the Arty board better suited to emulating real customer products.

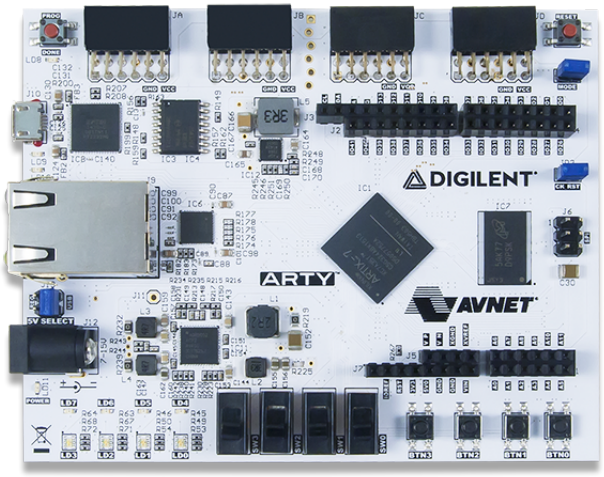

Figure 2.2: Digilent Arty Artix-7 board.

The project will require a computer or laptop to develop the core and compiler on and continuous integration systems to perform testing on the incremental builds. For the project demo, an oscilloscope (already owned) or digital logic analyser may be required to demonstrate some of the core's features.

Chapter 3

PRCO304 Processor Design

3.1 Introduction

The PRCO304 Processor Design is the first of two deliverable sub-projects required for this project. The processor is designed to be a small, instantiated, Verilog module that can be easily inserted into existing FPGA-based Verilog projects.

The processor core is not designed for physical implementation in silicon but rather for FPGA devices.

3.2 High Level Design

The PRCO304 processor is a modularised processor with modular logic blocks for the ALU, Registers, RAM, and it's peripherals.

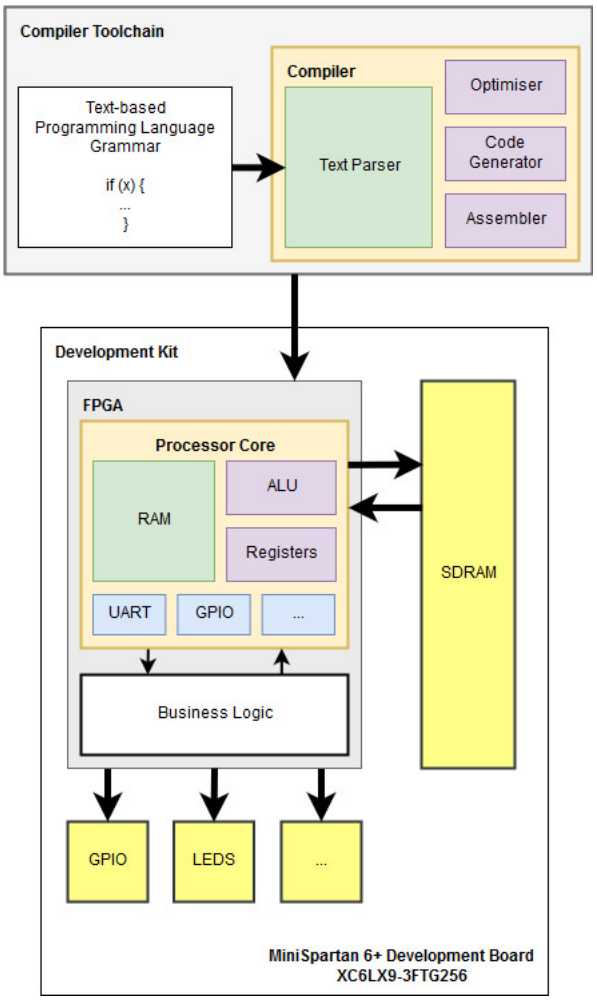

Figure 3.1: test

3.3 Registers

PRCO304 has a total of 6 addressable, read and write, registers. These registers are identified by letters A through H.

3.3.1 General Purpose Registers

Registers A through E are designed for general purpose use and are safe to store user values over the run-time of the processor.

Table 3.1: General purpose registers.

Registers	Bits	Description
A through E	15:0	5 General purpose registers

Instructions that require a destination register, such as CMP, can reference any register (even special registers if that is your requirement). For the CMP instruction as an example, the processor will put the result of the comparison instruction in the destination register, overwriting any value present in that register.

3.3.2 Special Registers

Registers F through H are special registers within the processor. The processor cannot guarantee that a value written or read in these registers will persist over the run-time of the processor. Erroneously writing to these registers may severely affect program and processor behaviour.

Even though all registers can be used at the will of the programmer, it is recommended to isolate a few registers to provide special features, such as RAM stack management, interrupts, and IO multiplexing.

Table 3.2: Special registers.

Registers	Bits	Description
F	15:0	Status Register
G	15:0	RAM Base pointer
H	15:0	RAM Stack pointer

Status Register

The Status Register is a dedicated register used by the ALU to provide additional information on results of instructions. Using the Status Register is essential for programs wanting to perform conditional branching or operate on dynamic data.

Table 3.3: Status Register breakdown.

Bit	Name	Description
0	SR_Z	Set if the result of a CMP instruction is 0.
1	SR_E	Set if the two operands of a CMP instruction are equal.
2	SR_S	Set if operand B is greater than operand A.

By default, the JMP instruction will read the Status Register to compare against the instruction's conditional jump parameter.

Base Pointer

The PRCO304 processor assumes that the compiler will employ a stack management scheme similar to that of x86 machines. By doing so, the compiler assumes the last 2 registers are dedicated to stack management. The Base Pointer register is used in a similar way to the x86 Base Pointer register.

Compilers and code generators should utilise this register for storing the address of the current stack frame. By utilising the register this way, features such as local and passed variables become available as they are addressable by offsetting the Base Pointer by a constant value.

Stack Pointer

The Stack Pointer is similar to the x86 Stack Pointer in that it stores the address of the top of the stack. This register is used primarily for PUSH and POP operations (see section 4.3.4 PUSH and POP for example usage).

3.4 Pipeline Architecture

The PRCO304 processor employs a *feed-forward* pipeline strategy. This pipeline supports:

- Time-varying processes: Multi-clock cycle decoding; Memory access; ALU ops.
- Module re-ordering: Instruction dependencies; Module skipping; Output redirection.
- Interruption (see section **??**: **??**).

As the pipeline is feed-forward, no information is sent back to previous modules to tell them of their status. This means that if a module is stalled (due to mutli-cycle processes or future modules are stalled), and the previous module is ready, the previous module will signal the next module that information is ready and it should take it, but the current module is unable to as it is busy. The pipeline resolves this issue by it's cyclic nature. This means that only 1 module at any time is processing data. Of-course, the downside to this approach is that instruction parallelism is reduced.

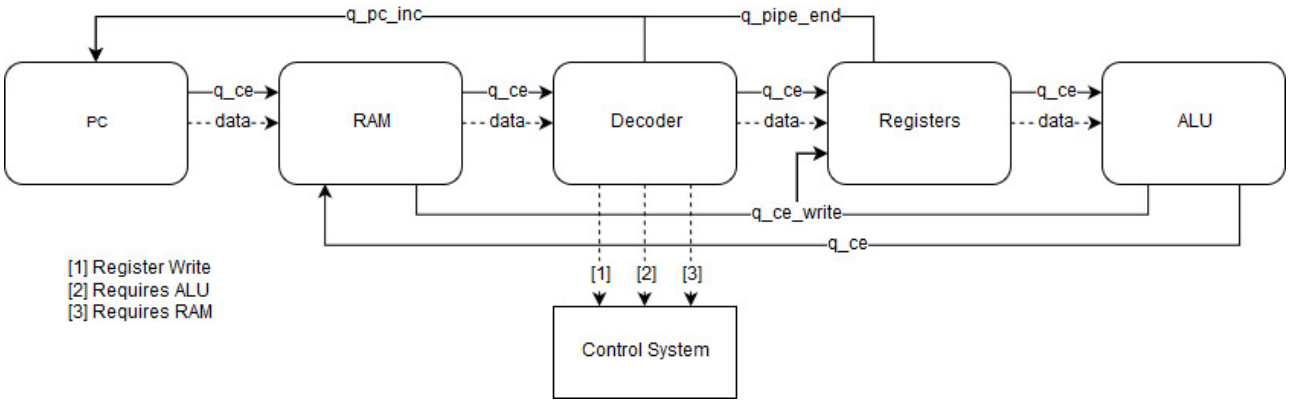

Figure 3.2: The feed-forward pipeline interconnect diagram used by the PRCO304 processor.

The pipeline structure is described in figure 3.2 (above). The general order of the modules is shown from left to right, but this can change due to the pipelines re-ordering functionality.

The Decoder module will decode instruction words from memory and will output appropriate signals containing the requirements of the instruction, such as requiring register write access, any ALU operation, and whether the instructions requires access to internal/external memory.

To improve instruction performance, the decoder can also choose what modules are required and when they are called. For example, for the **??** (move immediate) instruction the Decoder will assign the following modules in the following order: ALU and Register write, resulting in a total of 5 stages (including PC, Fetch, and Decode). The last module in this pipeline, the Register write, will raise the *q_pipe_end* signal indicating that the pipeline has finished and to start fetching the next instruction.

For the **??** instruction, the decoder identifies that the instruction requires no dependencies and will hence signal the *q_pc_inc* signal resulting in only 3 pipeline stages.

For instructions that require RAM access, a typical pipeline order might look like: PC, Fetch, Decoder, Register Read, ALU, RAM, resulting in 6 stages being used.

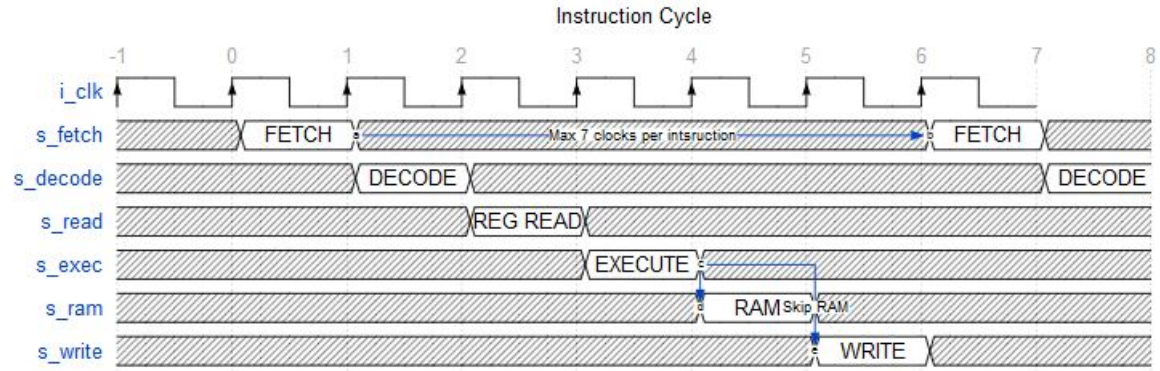

Figure 3.3: PRCO304 processor instruction cycle time diagram.

3.5 Testing and Verification

Chapter 4

PRCO304 Compiler

4.1 Introduction

The PRCO304 compiler is the second of two sub-project deliverables for this project.

The PRCO304 compiler is a command line based software tool used to convert a high-level text grammar (a programming language) into executable machine for the PRCO304 processor.

The compiler is invoked with parameters for the input code file and optional parameters specifying optimisation level, target architecture, verbosity, output file name, and include directory paths. The full command line parameter list can be found in CLI Arguments.

4.2 Implementation

The compiler is implemented fully in the ANSI C programming language due to my familiarity and experience in the language. The compiler is self-contained and requires no dependencies other than the standard C library and CMake to build the project. The project strictly follows the ANSI C89 style guide to make the code more readable and is compiled with `-Wall -Wextra -Wno-comment`.

Building the Compiler

To build the compiler, run the following commands:

```
cd prco304
mkdir build && cd build
cmake ..
cmake --build .
```

If you wish to build the compiler's own standard library run the following command as root/administrator to install the sources and header files:

```
cmake --build . --target install
```

4.3 Text Grammar

The input to the compiler is a generic programming language similar to C.

```
def main() {
    int a = 0;
}
```

The grammar is defined below in Backus-Naur Form:

```
<word>     ::= [a-zA-Z]+[0-9]*
<string>   ::= """ <word> """
<number>   ::= [0-9]+

<top>      ::= <func_def>|<decl>|<extern>

<func_def> ::= <proto><body>
<proto>    ::= "def" <word> "(" <args> ")"
<body>     ::= "{" <primary> "}"

<primary>  ::= <decl>|<control>|<assign>
<decl>     ::= <word> "=" <expr>

<control>  ::= <if>|<for>|<while>
<if>       ::= "if" "(" <expr> ")" <body>
<for>      ::= "for" "(" <expr> <expr> <expr> ")" <body>

<expr>     ::= <assign>|<binop>|<number>|<string>|"("|")"

<assign>   ::= <word> "=" <expr>
<binop>    ::= "+"|"-"|"*"|"/" <expr>
```

Figure 4.1: BNF definition for the input programming language.

It should be noted that the grammar and compiler do not have any terminals for defining datatypes, such as ”short” and ”int”. This is because there is only one datatype supported by both compiler and processor. This is due to the complexity required to support different sized datatypes, for example, calculating how many 16-bit words to allocate on the stack for local parameters and accessing them through offsets is difficult and out of scope.

4.3.1 Text Parser

The compiler implements it’s own recursive descent parser for the grammar described in 4.3. The parser is able to recognise all context free grammars and therefore would be capable of parsing more complete programming languages such as C and Python.

The text parser is inspired by Jack Crenshaw’s ”Let’s Build a Compiler” book, (**?**).

While parser generators already exist, such as Bison and Java’s ANTLR, it was decided to implement the parser by hand using recursive descent principles as a matter of learning rather than ease of use. Although parsing a more complex grammar would easily be more achievable using a parse generator, the overhead of generating compliant assembly for that complex grammar would be too time consuming and is hence out of scope (see extended deliverable E3.).

4.3.2 AST Generation

The recursive descent parser stores all terminals in the grammar as structures in *ast.h* containing relocatable information about the parsed text and it’s future implementation. This AST result of the text parser is the initial immediate representation used by the compiler.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis

```
struct ast_func {
    struct ast_proto *proto;
    struct ast_item  *body;
    struct ast_item  *exit;
    struct list_item *locals;
    struct ast_func  *next;
    int    num_local_vars;
};
```

Figure 4.2: An AST structure representing a parsed function. It contains sub-structures pointing to it's prototype, body, exit statement, and a list of local variables. (*ast.h:63*)

ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.3.3 Optimisation

The PRCO304 compiler can optionally perform simple optimisations, such as unreachable code elimination and constant folding. The optimisations can be controlled by specifying the `-On` parameter to the CLI, where `n` is the level of optimisation.

The techniques used by the optimiser to perform these optimisations are primitive; the optimiser is not given AST information in SSA (static single assignment) form; and because of this limitation, only basic optimisations can be identified.

Constant Folding

Constant folding is performed by the optimiser to reduce (fold) expressions that can be identified as constant. This allows the optimiser to replace AST tree structures containing constant values and no dependencies with shorter and simpler AST items. This optimisation can drastically improve the performance of the output code by reducing the number of instructions emitted.

For example, the following expression in Figure 4.3 can be identified as constant and can be reduced to a single AST node as shown in Figure 4.9. As the optimiser is not passed AST information in SSA form, the optimiser cannot follow or track variable references and modifications throughout the life-cycle of the program. Although the parser does a contain a primitive symbol table, the symbol table does not map variables to values, and so the code segment in Figure 4.4 cannot be identified as constant by the optimiser.

```
int a = 1 + (2 + 3) * 4;
```

Figure 4.3: Example of an expression suitable for constant folding.

```
int a = 1;
int b = 2;
int c = a + b;
```

Figure 4.4: Example of an expression the optimiser cannot identify as constant.

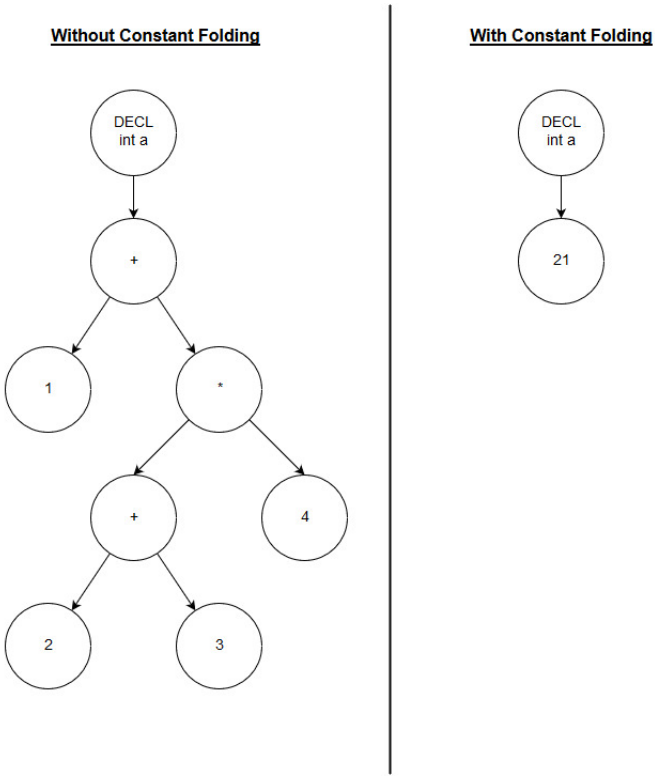

Figure 4.5: AST transformation performed by Constant Folding.

Unreachable Code Elimination

Unreachable code elimination is the removal of code that will never be run on the processor. This can be in the form of uncalled function, unused variables, and control statements that operate on constant values.

The PRCO304 compiler can identify some unreachable code segments, such as control statements that operate on constant values, by utilising it's constant folding optimisation discussed previously. By first running the constant folding optimisation on the body of functions, the optimiser looks at the conditions of *if* statements. If it's condition has been constant folded to a constant and is *true* (i.e. not 0) then the AST tree can be replaced with the items in it's body, effectively removing the condition check if it's always true, or the whole structure if it's false.

4.3.4 Code Generation

The compiler Code Generation stage converts the optimised AST into an intermediatary list of `struct prco_op_struct`. It does this by iterating over each `struct ast_item` in the AST and checks whether the item requires code generation. For example, an `struct ast_item` with type `AST_FUNC` is one which requires code generation. The AST is then passed to the `void cg_func_prco(...)` function where the code generation takes please. For this type, the stack frame header is generated first, before the body of the function. At the end of the function's body, the stack frame end code generation routing is run.

This code generation stage is named intermediatary because absolute addresses of `JMP` instructions have not been calculated. The calculation of these addresses is performed in the following Assembling stage. In addition, the location (and offset's) of functions may need to be rearranged.

PUSH and POP

Due to limitations of the PRCO304 processor's instruction set, high-level instructions such as PUSH and POP cannot be performed in a single instruction. Instead, the compiler is able to replicate the

behaviour of these high level instructions by emitting multiple primitive instructions. Figure 4.8 below details how the compiler emulates these high-level instructions.

```
void cg_push_prco(enum prco_reg rd)
{
    asm_push(opcode_add_ri(Sp, -1));
    asm_push(opcode_sw(rd, Sp, 0));
    asm_comment("PUSH");
}
```

Figure 4.6: PUSH emulation. The Stack Pointer is subtracted the amount to store on the stack (1 word), followed by storing the destination register (*rd*) at the new Stack Pointer.

```
void cg_pop_prco(enum prco_reg rd)
{
    asm_push(opcode_lw(rd, Sp, 0));
    asm_comment("POP");
    asm_push(opcode_add_ri(Sp, 1));
}
```

Figure 4.7: POP emulation. The value pointed to by the Stack Pointer is loaded in the destination register (*rd*), followed by incrementing the Stack Pointer the size of the data type (1 word).

Figure 4.8: PUSH and POP emulation functions used by the PRCO304 compiler (*arch/prco_impl.c:255*). Example of use: `cg_push_prco(Ax)` to push register Ax to the stack; `cg_pop_prco(Ax)` to pop stack into Ax.

4.3.5 Assembling

The final stage of the compiler is the assembling stage. This stage takes the list of `struct prco_op_struct` and outputs a list of machine code instructions. The assembler accomplishes this by calculating offsets and addresses of functions, branching instructions, and global variable addresses. It may also rearrange function locations so that the main function is the first instruction to be emitted.

Assembling code is found in `assembler_labels()` at `arch/template_impl.c:38`.

Executable Layout

Another role of the assembler in the PRCO304 processor is to output the machine code in a format that allows the widest range of programs to be run by the processor.

This format is not enforced by the processor core and it's up to the compiler to lay out the processor's memory contents. The only feature that the processor states is that it will start reading instructions from address `0x00`. The compiler uses this information to structure the output program. The first two words of memory (`0x00` and `0x01`) contain `MOVI` and `JMP` instructions to jump the processor to the address of the `main()` function.

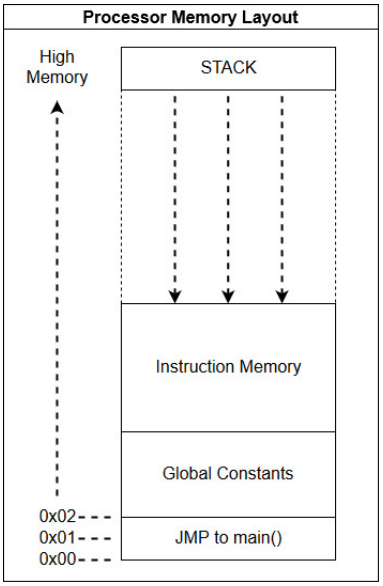

Figure 4.9: PRCO304 memory layout.

Limitations

Due to time constraints, the assembler introduces many constraints to the output program that are not explicitly identified in the high-level code.

The most prominent limitation is that the assembler can only address 255 words of memory. This is because the assembler only builds up instruction addresses using a single MOVI instruction, which is limited to an 8-bit immediate. This is easily fixable as the assembler could insert additional instructions to build up 16-bit addresses to use. For example, to build a 16-bit address, *0xFECA*, the following instructions could be used:

```
MOVI  $0xFE,   %Ax
LSHF  %Ax,     $8
ORI   %Ax,     $0xCA
JMP   %Ax,     JE_UC (unconditional)
```

4.4 Testing and Verification

Verifying the output assembly is a bit more involved as there are multiple layers of tests required. The output code generation must be tested for:

- (A) Correct machine code output for different code generation routines.
- (B) Correct and complete flow of the output program.

For (A), a code generation routine refers to the code generation function used to produce machine code for a specific structure, for example a function or assignment expression. When machine code instructions are emitted from the code generation routines, they are pushed to a list of `struct prco_op_struct` containing information about the emitted instruction. Using this information, the final output machine code word (e.g. `0x2020`) is rebuilt into an equivalent `struct prco_op_struct` structure and compared against the original. If they are the same, the encoded machine code word is considered correct. This check happens every time an instruction is emitted from the code generation routines.

For (B), two approaches are used. The first is to emit equivalent assembly for another architecture to be compiled are run on the host computer. If the return value of this program is correct, the assembly is assumed to be correct. These tests can be launched by running the `test/run_tests.bat` file. The second approach is to run the compiler output directly on the PRCO304 processor, however, this requires rebuilding the FPGA design with the new program code which is time consuming and not always practical.

Another potential solution for verifying output code that is not employed is to create an emulator for the PRCO304 processor and have it run the output machine code. This suffers the same problem as (B) where we are relying on systems that may not exactly share the behaviour of the physical processor, be it through unknown bugs, assembly printing errors, or architecture differences.

Chapter 5

Conclusion

5.1 Project Post-mortem

Chapter 6

Appendices

6.1 Appendix A. PRCO304 Core Reference Guide

6.2 Appendix B. PRCO304 Compiler Reference Guide

6.2.1 CLI Arguments

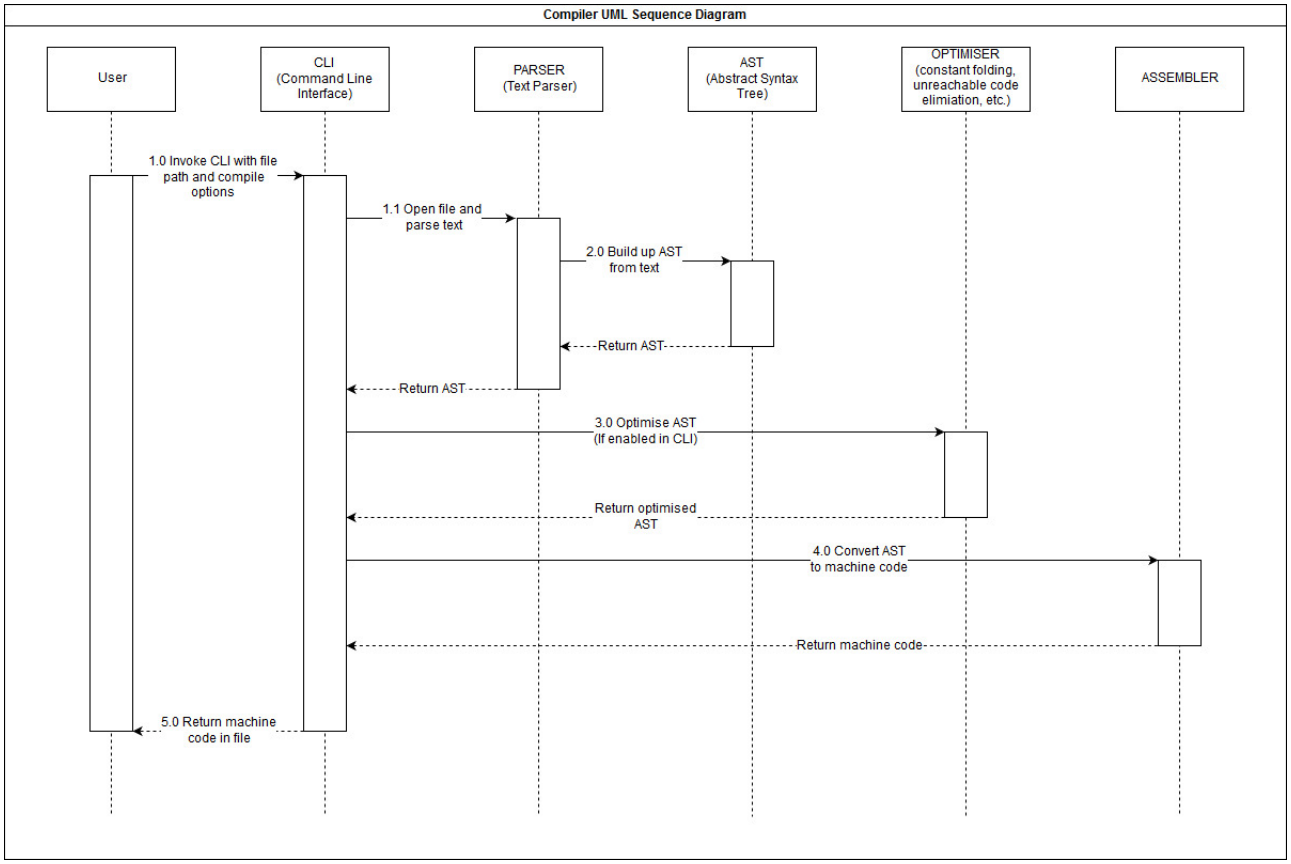

Figure 6.1: UML sequence diagram for the PRCO304 compiler.

6.3 Appendix C. Project Initiation Document