VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

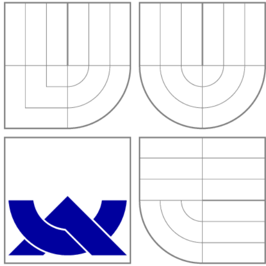
C LANGUAGE COMPILER BACK-END FOR PICOBLAZE-6

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

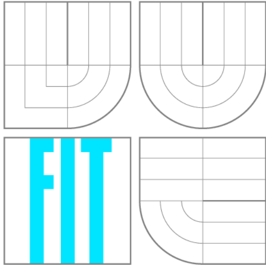
AUTOR PRÁCE
AUTHOR

MARTIN BŘÍZA

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZADNÍ ČÁST PŘEKLADAČE JAZYKA C PRO PICOBLAZE-6

C LANGUAGE COMPILER BACK-END

FOR PICOBLAZE-6

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN BŘÍZA
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZBYNĚK KŘIVKA, Ph.D.
SUPERVISOR

BRNO 2014

Abstrakt
Tato práce řeší konstrukci zadní části kompilátoru jazyka C pro soft-core procesor PicoBlaze-6 od firmy Xilinx. K řešení tohoto problému bylo zvoleno užití projektu Small Device C Compiler coby přední části překladače. Vytvořené řešení poskytuje podporu volání ukazatelů na funkce a užití struktur. Hlavním přínosem této práce je přenesení pokročilých konstrukcí jazyka C na procesor PicoBlaze.

Abstract
The goal of this thesis is to construct a C compiler back-end for the soft-core processor PicoBlaze-6 by Xilinx, Inc. The construction itself was done by use of the Small Device C Compiler as the front-end. The resulting application offers the ability to compile function pointer calling and structure usage. The main benefit of this thesis is bringing some of advanced C language constructs to the PicoBlaze processor.

Klíčová slova
vhdl, c, kompilátor, sdcc, picoblaze, procesor, fpga

Keywords
vhdl, c, compiler, sdcc, picoblaze, processor, fpga

Citace
Martin Bříza: C Language Compiler Back-End for PicoBlaze-6, bakalářská práce, Brno, FIT VUT v Brně, 2014

Rozšířený abstrakt

Tato práce se zabývá konstrukcí zadní části překladače jazyka C pro procesor PicoBlaze-6 od firmy Xilinx.

PicoBlaze je soft-core procesor (šířený jako design ve VHDL a Verilogu) určený pro vložení do FPGA čipů řady Spartan a Virtex. Jeho nově vydaná verze, 6, přidává několik nových instrukcí, upravuje chování řady stávajících a především rozšiřuje velikost programové i operační paměti spouštěných programů.

Pro vytvoření kompletního překladače je použit open-source framework Small Device C Compiler, který tak slouží jako přední část. V práci popsaná zadní část pak generuje jazyk symbolických instrukcí v notaci určené pro pBlazASM od firmy Mediatronix.

Výsledná práce se pak věnuje především implementaci pokročilých konstrukcí jazyka, které nebyly dostupné v předchozích kompilátorech.

Jmenovitě, tou nejpodstatnější vlastností jsou ukazatele na funkce, následuje podpora komplexních datových typů (například struktur.

Z ostatních vlastností kompilátor umožňuje využívat vstupní i výstupní porty procesoru pro komunikaci s periferiemi. Implementované je i volání funkcí, včetně rekurze. Podpora aritmetických operací je základní, tedy shodná s jejich podporou na straně procesoru - podporovány jsou všechny operace kromě násobení a dělení. Nicméně, kód pro násobení a dělení na tomto procesoru je možné převzít z předchozí diplomové práce na toto téma.

Vytvořený program byl otestovaný sadou krátkých netriviálních příkladů - programů, které demonstrují požadovanou funkcionalitu.

C Language Compiler Back-End for PicoBlaze-6

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval sám pod vedením Ing. Zbyňka Křivky, Ph.D.

. . . .
Martin Bříza
May 20, 2014

Poděkování

Rád bych poděkoval svému vedoucímu za odborné vedení, motivaci a cenné rady při řešení této práce a také svým blízkým za podporu v tomto období.

© Martin Bříza, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Bachelor’s Thesis Assignment

Unofficial Translation

Student

Martin Bříza

Specialisation

Information Technology

Topic

C Compiler Back-End for PicoBlaze-6

Category

Compilers

Instructions

1. Study the standard of C programming language and the specification of the version 6 of the 8-bit softcore processor PicoBlaze by Xilinx.
2. Examine the existing compilers for 8-bit processors (e.g. PCCOMP, PBCCv2) and identify their fundamental flaws and problems.
3. According to your supervisor’s instructions select an appropriate subset of C language and choose an existing C compiler front-end (e.g. SDCC or LLVM) for the implementation. Then, design the back-end for the chosen front-end to translate the subset of C language to the assembly language of PicoBlaze-6 target platform.
4. Implement the design.
5. Test the resulting compiler using at least 10 non-trivial examples and evaluate its pros and cons compared to the other examined alternatives. In the conclusion, propose some improvements of the compiler.

Supervisor

Křivka Zbyněk, Ing., Ph.D., UIFS FIT VUT

Contents

1 Introduction 5
1.1 The C Programming Language 5
1.2 The Previous Project 5
1.3 Structure of the Document 5

2 The PicoBlaze Processor 6
2.1 History 6
2.1.1 PicoBlaze 6
2.1.2 PicoBlaze-2 6
2.1.3 PicoBlaze-3 7
2.1.4 PicoBlaze-6 7
2.2 Main Features 7
2.3 Practical usage 7
2.4 New features and properties of PicoBlaze-6 8
2.5 PacoBlaze 9
2.6 PicoBlaze Assemblers 9
2.6.1 pBlazASM 9
2.7 PicoBlaze Simulators 9
2.7.1 pBlazSIM 10
2.7.2 PBSim 10

3 Programming language compiler 11
3.1 Compiler Structure 11
3.2 Preprocessor 12
3.3 Front-end 12
3.3.1 Lexical analysis 12
3.3.2 Symbol Table 13
3.3.3 Parser 13
3.3.4 Code Generator 14
3.4 Intermediate Code 14
3.4.1 The Intermediate Code Optimizer 15
3.5 Back-end 15

4 Compiler Front-End Choice 16
4.1 GNU Compiler Collection 16
4.2 Low Level Virtual Machine 16
4.2.1 Architecture 17
4.3 Small Device C Compiler 17

5 Existing Solutions 18
5.1 PCComp . . . 18
5.1.1 Features . . . 18
5.1.2 Limitations . . . 18
5.2 PBCC by Bohumil Nováček . . . 18
5.2.1 Features . . . 19
5.2.2 Limitations . . . 19
5.3 PBCC by Jakub Horník . . . 19
5.3.1 Features . . . 19
5.3.2 Limitations . . . 20

6 Implementation Design 21
6.1 New Port Addition . . . 21
6.1.1 Automation of new port addition . . . 21
6.2 Compilation . . . 22
6.3 SDCC Internals Wrapper . . . 22
6.3.1 Approach . . . 22
6.4 Utilities . . . 22
6.4.1 Emitter . . . 22
6.4.2 Function . . . 23
6.5 Register and Memory management . . . 23
6.5.1 Type support . . . 23
6.5.2 Register Allocation . . . 23
6.5.3 Stack Management . . . 24
6.5.4 Static Memory . . . 24
6.6 Code Generator . . . 24
6.6.1 Calling Conventions . . . 24
6.6.2 Function Pointers . . . 25
6.6.3 Register Bank Utilization . . . 25
6.6.4 Assignment Generation . . . 26
6.6.5 Stack and Variable Storage . . . 26
6.6.6 Arithmetic Operations . . . 26
6.7 Compiled Assembly Properties . . . 27
6.7.1 Comments in the Code . . . 27

7 Port Features 29
7.1 Code Clarity . . . 29
7.2 PicoBlaze-6 Support . . . 29
7.3 Built-in Functions . . . 29

8 Testing and Evaluation 30
8.1 Testing . . . 30
8.1.1 Tools and the Testing Process . . . 30
8.1.2 The Framework . . . 31
8.1.3 Test Cases . . . 32
8.2 Evaluation . . . 32

9 Conclusion 34
9.1 Future Development . . . 34

A CD Contents **37**

B Installation and Usage **38**

B.1 Usage 38

C PicoBlaze-6 Instruction Set **39**

Chapter 1

Introduction

Subject of this thesis project is constructing a C programming language compiler back-end for the soft-core processor PicoBlaze-6. The 8-bit processor is quite simple but after the recent update, it provides an interesting set of features to use in C programs.

However, Currently, there is no other C compiler designed especially for the PicoBlaze-6 processor, therefore a new compiler utilizing the new features is a welcome addition to the existing toolchain.

1.1 The C Programming Language

The C programming language was created in the 1970s by Dennis Ritchie and Ken Thompson. In this project, I focused mainly on implementing its two latest standards, ISO/IEC 9899:1999 [2] and partially ISO/IEC 9899:2011 [11].

1.2 The Previous Project

Similar thesis topic was elaborated by Jakub Horník as his Master thesis in 2011. This project is discussed in Section 5.3.

While he was writing the thesis, a new version of the target processor was released (see Chapter 2). The resulting application is discussed in Section 5.3.

1.3 Structure of the Document

After the introduction to PicoBlaze in Chapter 2 and explaining the basic principles of compiler design in Chapter 3, the available existing and open source C language compiler front-ends are enumerated in Chapter 4.

Next, the previous projects implementing a compiler for the older releases of the PicoBlaze processor, are listed in Chapter 5.

And finally, Chapter 6 explains the principles that the final application builds upon and its functionality is tested and evaluated in Chapter 8. Chapter 7 contains the summary of the compiler features and their description.

Chapter 2

The PicoBlaze Processor

PicoBlaze is a 8-bit processor created by Xilinx Inc. for their Spartan and Virtex FPGA[1] series as an embeddable circuit to implement sequential programming in the parallel FPGA architecture. This means the processor is not meant to be physically manufactured, yet it would be possible [24].

2.1 History

Historically, the name of the design was `KCPSM`, for Ken Chapman's Programmable State Machine and then Constant Coded Programmable State Machine. This term is now still used in the FPGA design - the *VHDL*[2] and Verilog components are still caled `kcpsmX` where `X` stands for the version of the processor [18].

2.1.1 PicoBlaze

The first version of PicoBlaze was released in May 1999.

The initial PicoBlaze was very simple, especially compared to the current one. There was only space for 256 instructions in the program memory and there was no RAM. Only 16 8-bit registers were available to be used for storage. However, it is possible to connect an external memory through the 256 I/O pins.

The multi-byte arithmetic instructions (`ADDCY` and `SUBCY`) were included since the very beginning.

There was no dedicated instruction for value comparison but both the zero and carry flags were already present [3].

2.1.2 PicoBlaze-2

The second version of the processor was initially released in December 2002.

The program could consist of 1024 instructions at most and there was no scratchpad memory for runtime variables. The instructions were able to operate only with the 32 8-bit registers and constant values.

The PicoBlaze instruction set remained the same in this revision.[4].

[1] Field-Programmable Gate Array

[2] *VHSIC* (Very High Speed Integrated Circuit) Hardware Description Language

2.1.3 PicoBlaze-3

First release of the previous version, 3, of PicoBlaze was released in May 2003. Its last revision, was released in August 2004.

The register count was reduced to 16 while their size has been kept the same. The main addition in this version is the 64 byte large scratchpad memory, that eliminates the need of an external memory connected through the I/O ports.

Instruction set changes in this version include the addition of comparison, testing and parity counting instructions [6].

2.1.4 PicoBlaze-6

The PicoBlaze-6 was initially released on 28th October 2010. Since then, a total of 7 newer revisions of this versions were released.

The most recent one, PicoBlaze-6 release 8, was released on 31th March 2014 [5].

The differences and new features compared to PicoBlaze-3 are discussed in Section 2.4. The main features, regardless of history, are listed in Section 2.2.

2.2 Main Features

PicoBlaze is a *RISC*[3], Harvard architecture[4] processor. Every instruction is executed in 2 clock cycles [5].

Program memory	Up to 4096 instructions
Scratchpad RAM	Up to 256 bytes
2 Register banks	Containing 16 8-bit registers each
Input/Output	256 8-bit ports
8-bit Arithmetic-Logic Unit	Supports shifting, adding and subtracting
	Provides AND, OR and XOR operations
	Compares and tests
	Implements carry and zero flags
FPGA design	Can be included directly in the hardware code
	No other equipment or code needed

Table 2.1: PicoBlaze-6 features [24]

2.3 Practical usage

A sequential processor, in comparison to parallel hardware design, is much more feasible for implementing state machines and computational cycles.

It is suitable to used to control simple devices and to communicate over serial interfaces. The use-cases presented by the manufacturer are LCD drivers, SPI communication, controlling devices such as A/D controller.

Time-based operations as pulse width modulation are possible with the processor, as well as display of real time clock or frequency measuring [27].

[3] Reduced Instruction Set Computing

[4] The program code and the data memory are stored in separate locations

2.4 New features and properties of PicoBlaze-6

Compared to the previous version, there were major changes in PicoBlaze design after the upgrade to the sixth version, which is still backwards-compatible.

The most important ones are covered in this section in descending order according to their impact on programming in C and the compiler itself [5].

New `JUMP@` and `CALL@` Instructions

Indirect jumps and calls, provided the address of the function or label, are possible in the new version.

This improves compatibility with sophisticated C programs greatly as it allows the implementation of function pointers and their calling.

It also means there had to a new addressing mode introduced - the whole program memory cannot be covered with only an 8-bit value, so the code's section is stored in the lower four bytes of the first register and the rest of the address in the second one.

New `REGBANK` and `STAR` Instructions

The new version of the processor now provides two sets of 16 general purpose registers that are switchable using the `REGBANK` instruction - this means only one of the sets can be accessed at a time.

To store values in the inactive bank, the `STAR` instruction is provided - it is used to store a value from the active bank register in another register that is in the inactive bank.

`ADDCY` and `SUBCY` Instruction Changes

The previous version of these instructions modified the zero flag only according to their own result. Now they add the previous zero to the current one.

New `COMPARECY` and `TESTCY` Instructions

Instructions added to make comparing multi-byte types easier. They store the flags (carry and zero) and propagate them according to the progress through the single bytes.

Program Memory Changes

Due to the two newly added memory addressing pins, it is possible to address four times more program memory (now up to 4096), increasing the possible program size and complexity.

RAM Changes

The amount of RAM addressable by the program was increased to up to 256 bytes from 64, yet it depends on the target device the processor will be implemented on.

This was achieved by modifying the opcodes of instructions of the processor, not the internals.

Call stack

Only 30 levels of function call depth are now available compared to 31 of PicoBlaze-3.

New `LOAD&RETURN` Instruction and `STRING` Directive

The user (or compiler) can now specify a byte string location in the memory using the `STRING` directive in the assembly.

Then, using the `LOAD&RETURN` instruction, it is possible to load a constant value into a specified register and unconditionally return from a subroutine, making these very useful in conjunction to generate text strings to be presented to the user.

New `OUTPUTK` Instruction

This instruction allows the program to output a constant instead of loading it into a register and outputting it using the regular `OUTPUT`.

2.5 PacoBlaze

PacoBlaze is an open-source (under the BSD license) clone of the PicoBlaze processor written in Verilog.

Its main advantage is the possibility to be used with hardware not provided by Xilinx and modifiability and configurability.

Due to the higher versatility, *PacoBlaze* is not as efficient (resource-wise) as the original implementation but it enables the user to remove unneeded parts from the processor and use its smallest required subset, therefore reducing FPGA space use as the result.

The latest version is 2.2 (released in 2007) so it cannot possibly implement the new instructions of PicoBlaze-6 that this project tries to use as much as possible [13].

2.6 PicoBlaze Assemblers

Compilation to the machine code directly is not feasible because it prevents further modifications of the resulting code. It's also not absolutely necessary as there already are complete assemblers, with features, implementing which would reduce the time available for writing the C compiling part of the toolchain.

2.6.1 pBlazASM

PBlazASM is an open source assembler created by Mediatronix to be used for compiling the machine code source files directly to machine code in several formats. It can also create a representation (`.lst` files) that is then used in the pBlazSIM simulator (that is distributed in the same repository, see Section 2.7.1) [21].

However, the program itself is bug infested and crashes quite often. There were some trivial bugs that were possible to be fixed quickly, such as calling `free` on a pointer that was not returned by `malloc`.

2.7 PicoBlaze Simulators

As the debugging capabilities of the processor are limited, making debugging of the programs on real hardware too complicated, possibility of usage a simulator of the target processor is more than welcome.

2.7.1 pBlazSIM

One of the most recent ones, pBlazSIM, was created by Mediatronix, too. It is actually hosted in the same repository tree as pBlazASM (see Section 2.6.1), along with some other tools [23]. [22]

Hosted under the GPLv2 open source license, it is written in C++ and Qt, so its use is not limited only to one platform. Mediatronix distributes only the Windows binary form, though.

Running Under Linux

First attempts to run the simulator in *Wine*[5] were quite unsuccessful as it was only capable to simulate the code which was included in the distributed binary package and no other.

Attempts to compile the simulator from the source provided were also unsuccessful at first. Some changes to the distributed files were required.

The needed steps to finish the compilation successfully are fixing the `qmake` project file to include all needed source files and adding one missing icon for the GUI[6] (for example by copying the existing ones in different colors).

The hand-compiled version is able to run the assembly files generated by pBlazASM (see Section 2.6.1) well. All features in the GUI are working, too.

I contacted the upstream developer responsible for the changes that introduced the invalid constructions to fix them. The code was fixed by the developer in two days.

Command Line Simulator

A simple simulator without any user interface was introduced to the code-base recently. The officially provided build system information does not handle its compilation but due to the relative simplicity of the project, it is easy to compile this executable by hand by using the object files of the pBlaze.cpp module.

The simplest way possible (provided you use a C++ compiler that is able to link directly) to compile this binary is by using the following command:

```
${CXX} pBlaze.cpp pBlazSIMcl.cpp -o ${SIM_BINARY}
```

2.7.2 PBSim

A project that is being developed on the University, started as a Master's project in year 2012 [16].

Bound to SDCC (see Section 4.3) and tightly related to Eclipse, it is possible to use it as a part of its UI.

It is hosted under the GPLv2 license on Github [14] but there are no instruction to compile the project. In comparison to pBlazSIM (see Section 2.7.1), it's not active at all.

[5]WINE is not an Emulator - an open source Windows API implementation

[6]Graphical User Interface

Chapter 3

Programming language compiler

Most of the modern computer processors are programmed using a fairly complicated binary instruction set. To make writing more complex and powerful applications more pleasant and comprehensible, we are now using programming languages that are translated (*compiled*) into the low level binary form executable by the processor.

In this chapter, the most vital parts and terms in construction of a compiler are discussed, with focus on C language [1].

3.1 Compiler Structure

A typical programming language compiler consists of two to four main parts:

First part is an optional preprocessor which prepares the source code for the front-end, for example by removing comments or expanding macro definitions.

A language front-end, which transforms the code from a programming language to its simple intermediate and internal representation.

An intermediate code optimizer searches the representation for patterns that can be changed to more efficient, smaller or faster equivalents. This part is optional, too.

And finally, the back-end, that generates the target code for the processor itself, be it assembly code, virtual machine code or a processor-native code [1].

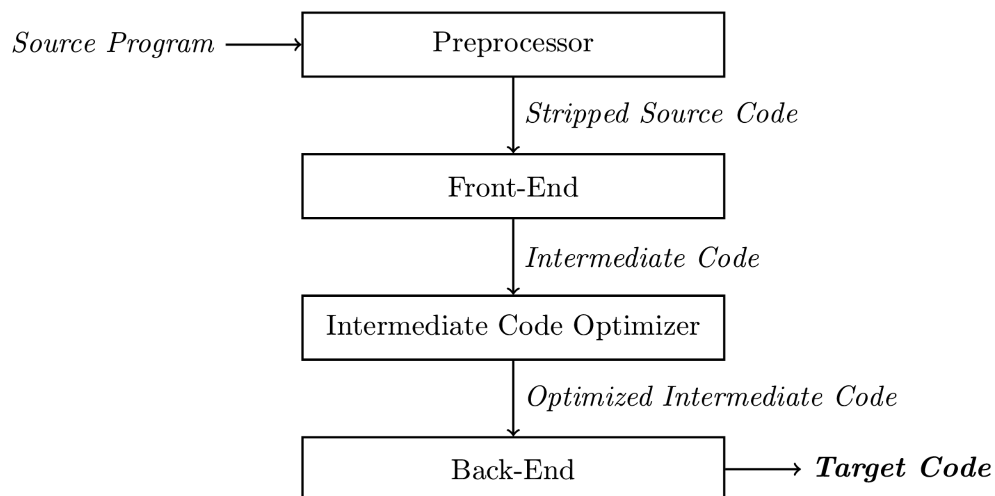


Figure 3.1: Structure of a compiler front-end

3.2 Preprocessor

Preprocessor performs a simple task of removing or replacing text in the input source code according to some pre-set rules.

The most common task is removing comments to leave only language defined tokens for the further stages of compilation.

In the C programming language, there are preprocessor macros, too. These serve a purpose of replacing and inserting text. However, describing the entirety of the C preprocessor is a task beyond limits of this thesis, only few directives are listed, along with their simplified descriptions [28].

`#include "file"` or `<file>`	Behaves as if the whole contents of `file` were inserted instead of it.
`#define MACRO ...`	Every occurrence of `NAME` in the code is replaced with what is substituted with `...` (until the end of line).
`#if cond`	The following lines until `#endif` are pasted if `cond` is met. `cond` supports C expression syntax.
`#ifdef MACRO`	Equivalent to `#if defined(MACRO)`. True if `MACRO` was defined.

Table 3.1: Basic C preprocessor macros

3.3 Front-end

The central and most important part of the compiler, it does a whole set of operations over the source code to produce its independent representation.

It is a language-specific part, that also checks the correctness of the input code.

Large compiler projects even aim to its complete and clear replaceability (see Sections 4.1 and 4.2 for examples) to benefit from the optimization in the following stages of compilation.

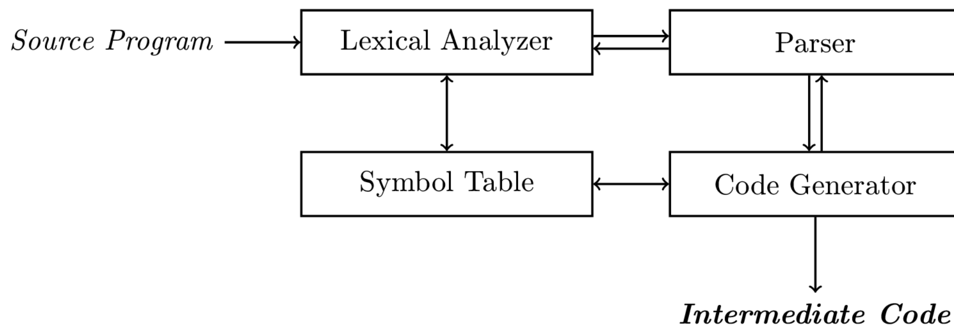


Figure 3.2: Structure of a compiler front-end

3.3.1 Lexical analysis

Lexical analyzer comes into contact with the raw source code.

Reading the input stream character by character, it breaks it down to segments called *tokens*, according to rules that are set by the language standard.

For example, there can be a token representing a *keyword* (like `if` or `for`), a literal value (`42` or `"string"`,...) or a type or symbol name (`foo`, `main`,...) [1].

A token of the `if` keyword will be always represented with the same string - `if`. However, a variable name in C can take form of any string that matches the following regular expression [2].

```
[A-Za-z_][A-Za-z_0-9]*
```

These names and values are then stored in the symbol table, which will be discussed in the next section.

3.3.2 Symbol Table

A *symbol table* is a container storing names of all symbols that the lexical analyzer has detected [1].

The values here are used further in the compilation, either to resolve the identity of tokens, or to actually assign the literal values and use them in the program itself.

3.3.3 Parser

Parser then takes the stream of tokens and arranges them into a tree-like structure. This process is called *syntactic analysis*. It consists not only of arranging the tree but as a side-effect, correct syntax of the input code is being checked.

Consider the following expression for demonstration of processes taking place in the parser:

```
1 * foo + 5 * bar
```

Listing 3.1: Example expression

The tree, as a result of the syntactic analysis, would take the form represented in the next diagram [8].

Note the operator precedence is honored in the same way as in regular mathematical expressions.

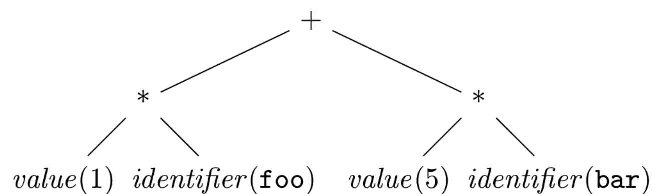


Figure 3.3: Syntactic tree

There is also other information about the tokens in this expression, like types in case of variables. It is necessary to check if they are used in the right context, like if functions are used like functions and not variables or if defined operations are being used upon them. This process is called *semantic analysis* [1].

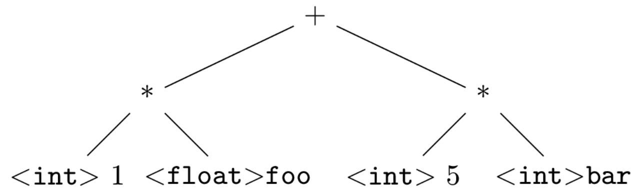


Figure 3.4: Operator and operand relation in semantic analysis

If `foo` was not a `float` but for example a `bool` variable the semantic analysis of the expression would fail because the operation of multiplication is not be defined between these two types.

The result types of expressions, are deduced as well. In this case, in C language, the whole expression's result would take form of a `float`.

3.3.4 Code Generator

While the trees are being constructed and checked, the intermediate code is created as well.

The *parser* feeds the completed subtrees and their nodes into the code generator that flattens the structure into a series of instructions, similar to the assembly language of a computer processor [1].

Only one kind of resemblance of the previous code will be left - the links in code branches for both conditional and unconditional jumps - this is required to make creating the links in the resulting code easier.

In case the compiler does not use any kind of intermediate code, directly the target language is being emitted in this section, making the code generator serve the purpose of a back-end (described in Section 3.5), too.

3.4 Intermediate Code

As mentioned in the previous section, the intermediate code resembles assembly. The difference between the two is still quite big though. Intermediate code contains a lot more information, like variable names, information about liveness of the variables, type information and much more.

Popular form of such output is called *three address code*. Every instruction in this simple language takes a maximum of three operands, first marks where the result will be stored and the other to are the source operands [1].

Consider the example we already used, defined in Listing 3.1.

The expression will be transformed to the following (simplified) series of instructions in three address code:

```
mul    temp1,       1,     foo
mul    temp2,       5,     bar
add    temp1,   temp1,   temp2
```

Listing 3.2: Example of a three address code

The instructions in the code can be limited to take only two operands. The first operand will be both the result and the first argument in this case. This code is equivalent to the previous, only printed in two-operand notation:

```
assign  temp1,      1
mul     temp1,    foo
assign  temp2,      5
mul     temp2,    bar
add     temp1,  temp2
```

Listing 3.3: Example of a two-operand intermediate code

Three address code as intermediate language is very common in compilers. This puts them close to some processor architectures, such as ARM, that uses a form of three address code as its assembly language. [12] In contrary, other architectures, like x86 [10], or, PicoBlaze [5], that this thesis targets, use only two operands in their assembly.

3.4.1 The Intermediate Code Optimizer

Optimizing the code during the compilation is not absolutely vital for the whole process. However, it is a welcome part, especially by the end-users of the tool.

The optimizer looks for patterns that can be reordered, changed for their faster equivalents or completely removed, while maintaining the same functionality as before.

To demonstrate on the three address code from Listing 3.2, consider this instruction:

```
mul     temp1,        1,      foo
```

Multiplying anything with 1 is redundant. After removing this unnecessary operation and changing the order to preserve the same result, the example would look like this:

```
mul     temp1,        5,      bar
add     temp1,    temp1,      foo
```

Listing 3.4: Optimized three address code from Listing 3.2

One whole instruction was saved in this case.

There are many more techniques that can be applied on any amount of code. One of the simplest ones is *dead code* and *dead variable detection*. When a variable is not used or a conditional branch cannot be entered at any time when the program will be executed, the optimizer can completely remove it [19].

3.5 Back-end

The target specific part of the compiler, generating the sought code, is the *back-end*.

For each instruction of the immediate code, it matches its equivalent, be it single instruction, or a whole function, in the target language.

Aside from this its main task is to allocate registers for each target instruction, as the immediate code does not use any. Because the number of the registers is finite, it also needs to handle management of other data storage forms, such as the stack [1].

More complex compilers are usually aiming to be *retargetable* - that means they are equipped to support multiple back-ends for different compilation targets. There is a list of some of them in Chapter 4.

Writing a compiler back-end is the goal of this thesis.

Chapter 4

Compiler Front-End Choice

As starting a new compiler from scratch would not be possible in the limited time frame of a bachelor thesis, I had to choose an existing front-end and provide a corresponding back-end part for the target platform.

This basically rules out all of the proprietary compilers and the selection limited to those from the world of free and open source software. In the end, the choice consisted of the following three.

4.1 GNU Compiler Collection

The GNU Compiler Collection, more known as `GCC`, was started as a simple C compiler in 1985 by Richard Stallman. It is now one of the most widely used compiler suites not only in open source systems [7].

Due to its huge history and background, its code base is stable and mature but it also is very hard to read due to historical reasons and the fact basically everything is wrapped in several layers of macros.

The documentation of the inner functionality is hard to find and it is not very well arranged. Because of its heritage, the structure does not seem very transparent.

Free Software Foundation and the GNU Project are holding governance over the development and are prohibiting major changes to the architecture or code style which drives many new developers away.[1]

4.2 Low Level Virtual Machine

LLVM is a modern project with a gaining popularity in past years for implementing the features very fast and providing of interesting and useful tools, like static analyzer.

Its C and C++ front-end, Clang, is adding the latest features of the new language standards and their drafts sooner than the competitors.

Compared to GCC, LLVM is a really young project. It was founded in 2005. The codebase is dynamically changing, written in C++ with heavy use of templates and automatically generated code [29].

Its development is sponsored by companies like Google, for example to provide ability to run native applications in the browser (NaCl project, or especially its part PNaCl) or

[1] `http://gcc.gnu.org/ml/gcc/2014-01/msg00176.html`

Apple, which utilizes the ecosystem in the official development toolkit provided for their products [17] [30].

LLVM provides a very well documented intermediate representation of the compiled source code. Its documentation is publicly visible on their wiki page, every necessary detail is described and the community provides several easy ways to be approached.

4.2.1 Architecture

LLVM is strictly separated into front and back ends, divided by the LLVM intermediate code that is heavily optimized and is executable directly in a virtual machine (hence the name Low Level *Virtual Machine*)

4.3 Small Device C Compiler

SDCC is a simple (compared to the previous two) compiler aimed to be easily retargetable and provide a quality background for creating compilers for 8bit processors [26].

It is not a very large project (especially when compared to GCC and LLVM) and it uses parts (for example, the preprocessor) of GCC.

It optimizes the compiled source code with focusing on issues appearing on small devices.

The intermediate code is not documented very well (there is a list of all the iCodes on the project's wiki) but is simple enough to be understandable.

Chapter 5

Existing Solutions

The idea of writing C compilers for PicoBlaze is not new. There has been a few projects implementing C compilers directly, or compilers of languages based on C. The most exposed ones are covered in this chapter.

5.1 PCComp

PicoBlaze C Compiler, the project of Francesco Poderico, has its own page on SourceForge[1], yet there are no files to download or source code in the repository and the only relevant activity visible is a question where to actually download the compiler.

I managed to find a Windows binary in version 1.8.4 in a web archive and a user manual describing the compiler's features, both created in 2005 or 2006.

However, the limitations of the compiler are vast. It generates stack-based code. This is unfortunate because PicoBlaze lacks any stack [25].

5.1.1 Features

The compiler is not strictly following the C standard and implements only its small subset.

The supported cores are PicoBlaze, PicoBlaze-2 and PicoBlaze-3.

Only the support for byte and word (1 and 2 bytes) types was implemented.

One-dimensional arrays without any pointer arithmetic are supported by the compiler.

5.1.2 Limitations

Type conversions are missing, as are variable modifiers (e.g. `volatile`).

The compiler does not support any kind of resulting code optimization, except dead branch detection.

The compiled assembly is often buggy or even nonfunctional and the probability of getting broken code is increasing with the complexity of the input source code and the arithmetic expressions in particular.

5.2 PBCC by Bohumil Nováček

This bachelor thesis was written on Faculty of Electrical Engineering of Czech Technical University in Prague in 2008 when only PCComp (Section 5.1) existed.

[1] http://sourceforge.net/projects/pccomp/

A compiler was written as a goal of the thesis, resulting in a small application able to compile a limited subset of the C programming language [20].

Also, the source code is not to be found anywhere on the Internet, only the text part of the thesis was made public.

5.2.1 Features

The compiler only allows the user to compile a small subset of the actual ISO/IEC 9899:1999 standard.

Processor support is limited to PicoBlaze-3.

The types supported are `void`, `char` and `int`, again sized only 1 and 2 bytes.

Only one-dimensional arrays are available to the user.

Despite the simplicity of the compiler, there are some optimization methods implemented. For example, constant expressions are replaced by values directly.

5.2.2 Limitations

There is no support for any user-defined type, whether it is only an enumeration, a `typedef` type or a complex type (`struct` or `union`). This effectively limits the user to use only the basic types that are in this case integers sized one and two bytes.

There is no expression conditions, strings and multidimensional arrays.

These limitations are caused by the fact the author decided to write the compiler from scratch without use of any framework or front-end. The time needed to finish a complete C compiler is far beyond the time-frame of a bachelor thesis.

5.3 PBCC by Jakub Horník

PicoBlaze C Compiler is a project sponsored by Virtuální laboratoř aplikovaných mikroprocesorů realized on the Faculty of Information Technology, Brno University of Technology.

It was written in years 2010 - 2011 by Jakub Horník as a part of his master's thesis and is now maintained by Zbyněk Křivka, supervisor of this thesis [15].

The compiler is based on the Small Device C Compiler (SDCC) modified to provide support for the processor so it offers a subset of features of SDCC in version 3.0 [9].

5.3.1 Features

There is support for adding further optimization methods provided by SDCC, additionally to its own optimization procedures that are ran during the compilation process on the intermediate code.

Data types supported are integers large from 1 to 4 bytes, there is also no problem with converting them.

Use of arrays (even multidimensional) and pointers is implemented, including their use as function parameters.

PicoBlaze-6 was released only a few months after the inception of the thesis that was targeting the previous one, PicoBlaze-3. This topic is discussed in Chapter 2. The main focus of the thesis was the older iteration of the processor, therefore function pointers and all other new features were left unimplemented.

5.3.2 Limitations

There is a list of known problems and limits list in the official documentation. To name a few, there is no support for getting or setting values on a memory address and limited support for global variables and interrupt vectors.

The main reason to rewrite the compiler from scratch is to avoid carrying all the legacy instructions and features and to focus on the cleanest possible implementation of the current revision of the processor.

The author also suggests allocating the registers by coloring them and using the information for better results when memory access frequency is taken in question.

Unclear code copied over from other ports that is not very comprehensible is the reason why I wrote the whole program again while using just a few parts from the original code.

Chapter 6

Implementation Design

In this chapter, the technical details of the project are discussed.

The port itself is written in C++ with a layer wrapping the C internals of SDCC.

6.1 New Port Addition

As this process is not documented anywhere in the SDCC documentation and doing it properly would require deep and good understanding of the GNU autotools toolchain, the following procedure was used to add the new port to the SDCC source:

1. Create a port source directory in `src/`, in this case, I was calling it `pblaze`.
2. Add the basic source files in the port directory, for example `main.c` and `main.h`. `main.c` has to contain an instance of `PORT` structure containing information about the port specifications and pointers to functions that will be called during the compilation.
3. A new (unique) port ID needs to be inserted into `src/port.h`:

   `#define TARGET_ID_PBLAZE 16`
4. And create an `extern` reference to the `PORT` instance from `src/pblaze/main.c`, for example:

   ```
   #if !OPT_DISABLE_PBLAZE
               extern PORT pblaze_port;
               #endif
   ```
5. In `src/SDCCmain.c`, insert a reference to the structure defined in `src/pblaze/main.c`.

6.1.1 Automation of new port addition

These tasks are automated in the included `glue.sh` script. When it is executed in Bash[1] with the `SDCC_HOME` environment variable set to point to the directory with both PBCC and SDCC source code, it completes all the necessary tasks.

[1] Bourne Again Shell, http://www.gnu.org/software/bash/

6.2 Compilation

After the port was added, SDCC can be compiled. The steps to achieve successful compilation are:

1. `autoconf` creates a `configure` script to configure the components and compiler options of the final binary
2. `./configure` is a script that compiles a Makefile for the compilation itself. It is possible, for example, to modify the optimization of the compiler binary or disable compilation of ports of architectures we will not need.
3. `make` is the compilation script itself. You can speed the whole process by using the `-jX` argument specifying that `X` compiler processes should run at the same time.

6.3 SDCC Internals Wrapper

Using pure C library calls and macros in a C++ project would be a waste of potential of the language, therefore the project is built upon a wrapper library for the SDCC internals instead of using them directly.

The whole wrapper library is included in the `wrap` modules.

6.3.1 Approach

Every SDCC structure that is vital for the process is wrapped in its own class. These classes are `Set`, `EbbIndex`, `EbBlock`, `SymLink`, `Symbol`, `Value`, `Operand` and `ICode`. Their SDCC counterparts are named the same, except they have lower-case initials.

To provide the ability of implicit up-cast, the classes are directly inheriting the structures themselves.

Each of the classes has its methods derived from the functions and macros that are operating over them in the SDCC internal library. The methods are partially hand-written and partially generated from the definitions in the header files.

Aside from the methods, nothing was added to the classes. None of the methods is virtual. That means the memory footprint and binary compatibility with original structures is kept.

6.4 Utilities

The `util` module contains code for making the code in other parts of the projects easier to read and understand.

6.4.1 Emitter

Every target code output in the C++ part of the port is handled using this class.

It implements a `std::ostream`-like[2] (left shift operator overloading) API to be easy to spot in the code.

[2]Output stream class in the C++ Standard Template Library

There is one static instance of the class that is used to output from all other modules. The constructor was left open in case another separate output was needed. This option was left unused so far though.

The class also provides a static member variable `i` for iterating when an instruction is being output. This allows the `Instruction` class that is being handled to be able to check the byte position in the multi-byte operands. This makes changing instruction forms (for example between `ADD` and `ADDCY`) possible. It is also used in the overloaded left shift operator for operands, to get their current needed byte.

6.4.2 Function

`Function` is a class containing public static members only. It provides information about the current function the compiler is processing such as parameter count and their sizes.

Its main purpose is to compute which function parameters will need to be stored on the stack and which parameters will be passed through registers.

The processing method is called every time the `FUNCTION` iCode is reached.

6.5 Register and Memory management

The code providing the functionality described in Section 3.4 is located in the `ralloc` module.

The entry point is the `Allocator` class, precisely its static method `assignRegisters` that takes the basic block index as its parameter.

6.5.1 Type support

PicoBlaze-6 supports one byte integer operations only, some with possibility to reuse the carry bit (see Appendix C for the list of available instructions).

Implementing floating point operations on such a simple device is neither feasible, nor actually usable. Emulating hardware support for any standard defining the format would result in huge memory use and each operation would take huge amounts of processor power.

Advised usage in this case is to connect a hardware FP circuit over the I/O pins.

`char`	`short`	`int`	`long`	`long long`	`void*`	`void(*)()`
1	2	2	4	4	1	2

Table 6.1: Basic types supported by the compiler

6.5.2 Register Allocation

The register management is handled in the `Register` class. Two sets of `Register` instances are aggregated each in one `Bank` instance to provide the ability to switch the banks and pass operands between them.

When there is no free register available, the LRU[3] algorithm is applied to find the one that is to be freed and stored in the scratchpad memory until the next use.

[3] Least Recently Used

6.5.3 Stack Management

The stack is growing up, starting at the zero address. All stores and operations on both function entry and leave are processed in the `Stack` class which implements all necessary moves on the stack pointer (SP).

Each of the stored variables has an instance of `StackCell` class assigned, containing the information about the offset from the pointer and the start of the variable.

All variables are stored in big endian order, consecutively.

The stack pointer is stored in the `sF` register, in both banks. On bank switch, it is propagated to contain the changes done to it.

6.5.4 Static Memory

Static memory, used for storage of global variables, is implemented in the `Memory` class.

In most aspects, including the implemented interface, it is similar to the `Stack` (see Section 6.5.3). The most important difference is no necessity to do anything on function calls and different implementation using only position in memory.

It grows in the opposite direction to the stack, from the highest possible address in the memory - address `0xFF`.

As in the case of `Stack`, the variables are stored consecutively in big endian order.

6.6 Code Generator

To avoid as much code duplication as possible, every instruction is being emitted using iteration over the `Emitter::i` variable in the manner displayed in the following code snippet.

```
for (Emitter::i = 0; Emitter::i < left->getType()->getSize(); Emitter::i++)
    emit << I::Xor(result, right);
```

Listing 6.1: `Emitter` and `Instruction` example use

All instruction classes inherit from the generic `I` class that also acts as their parent and enclosing class both. It defines the `toString` virtual method that the children implement. This method is used in the overloaded left shift operator of `Emitter` (see Section 6.4.1) and `I` to obtain the whole string representation of the operation and write it into the output file.

6.6.1 Calling Conventions

The calling conventions were designed to utilize as much of the variables as possible because of the limited capabilities of the processor.

Caller saves strategy is implemented by the compiler. That means the registers are saved to the function stack by the caller, not the callee function. Variable liveness is considered, dead variables are omitted.

The arguments are stored consecutively in little-endian order[4] in the registers.

By default, the first 8 registers are used for passing the arguments directly and the rest is stored on the stack of the callee. Their count can be modified by the compiler command line argument `--argregs=N`, with the maximum value of `N` being 13.

[4]The least significant byte is stored first, the most significant one last

For example, consider the following function:

```
int function(int l, long r);
```

Listing 6.2: Called function prototype

Its initial register utilization would be as follows:

s0	s1	s2	s3	s4	s5	s6	s7
l[0]	l[1]	r[0]	r[1]	r[2]	r[3]	free	free

s8	s9	sA	sB	sC	sD	sE	sF
free	free	free	free	free	free	free	SP

Listing 6.3: Register bank state when function from Listing 6.2 is called

The returned value is stored in the first registers in the same way as the parameters were stored. Little-endian order limits the register use in the caller to the bare minimum. Returning structure values is not supported, pointer use is necessarry in this case.

After the function has returned, caller (according to the caller saves strategy) restores its variables back into registers.

6.6.2 Function Pointers

As mentioned in new feature overview (this particular change is discussed in Section 2.4), the processor now supports jumping to labels and calling functions that have their address loaded on run-time.

To make use of the new specification as much as possible, the compiler supports calling function pointers using the `CALL@` instruction.

Prior to the instruction invocation itself, the function pointer is stored in the `sD` and `sE` registers. As the address of a function is larger than the size of one register, it has to be stored in two of them - `sD` contains the upper 4 bits of the address and `sE` the lower 8 bits.

6.6.3 Register Bank Utilization

From the perspective of register banks, there are two types of functions (as there are two banks).

The first type is either function `main` or a state when the program has not entered `main` yet - on initialization of global and static variables. This code has access only to variables stored in registers in Bank A.

The other types are all other functions. These have access to registers in Bank B.

The bank selection is handled from the code of the first type. This makes the other functions able to call any other function including itself. Any kind of recursion including `main` is not supported though.

Before and after the call, bank has to be switched to the appropriate one. Also, if the function returned any value, it has to be transferred from the other bank to the current one before the bank is switched.

In effect, this makes having most of the program logic in the `main` function, as it does not have to store most of its registers before calling a function, thus reducing the number of necessary `FETCH` and `STORE` instructions to move the local variables in the stack memory.

6.6.4 Assignment Generation

There are different cases we have to handle on assignment in the compiler.

Function Call Parameters

The compiler detects when the assignment is done in a function call. This means the available operand registers are already full, so the following arguments need to be stored on stack.

These assignments are done preemptively, the variables go on stack directly. The stack pointer in the function will handle these as if they were saved there in the function body directly.

Dereferenced Pointer Assignment

The PicoBlaze-6 has only 256 bytes of scratchpad RAM (see Section 2.2). This means the pointers to variables take only one byte of memory.

Assignment to these variables is done using the `STORE` instruction. However, `STORE` cannot take a literal as its source data argument. Therefore, a temporary operand is allocated and used for temporary storage of the data being stored.

Temporary Variable on the Right Side

This case occurs when a temporary variable with a live scope that does not reach beyond the sequence number of the iCode containing this assignment. This usually occurs on an immediate computation in an complex expression.

Instead of moving the variable, the registers are only reassigned to the result operand.

Other Cases

The other cases cover assignments with regular variables and literal values as operands. The temporary variables with longer life scope are included, too.

There is nothing special on this case - the values are moved using the `LOAD` instruction. A `FETCH` is used first if the variable has been saved to memory.

6.6.5 Stack and Variable Storage

Local variables of the program are initially created in the registers. Only once there is not enough room for any other variable needed for an operation in future, it is freed from the register and stored (*spilled*) on the stack.

Stack pointer is stored only in register `sF` only in Bank B. Stack pointer is not tracked in the `main` function. As does register bank selection, this also makes recursive calls to `main` impossible.

6.6.6 Arithmetic Operations

Only basic arithmetic supported by the processor is implemented. More specifically, there is no basic support for neither multiplication nor division of the variables.

The multiplication functions can be extracted from the previous PBCC (see Section 5.3) and inserted as inline assembly in a separate function, according to the calling conventions.

The other arithmetic and logic operations of the C language are implemented, with focus on utilizing the new and improved instructions of PicoBlaze-6 to save computational cycles.

6.7 Compiled Assembly Properties

The compiler is producing commented assembly to be given to an assembler which then in turn produces its various binary equivalents or other formats, for example suitable for simulation.

In this section, the vital properties, required for getting grasp of the code (for its further modification by hand or integrating hand tuned assembly code, for example), are discussed.

6.7.1 Comments in the Code

To improve the readability and comprehensiveness of the code, there are explanatory comments included in the compiled assembly.

Instruction Comments

Lines of most of the instructions contain a short comment explaining the instruction and its operands. This covers moves and arithmetic operations especially.

To demonstrate, when the following code snippet is compiled:

```
int baz = 42 + foo - bar;
```

Listing 6.4: Example assignment

Then, assuming the variable was not loaded in the registers beforehand and the other variables were already used (`foo` is in `s0` and `s1` and `bar` is in `s2` and `s3`), the resulting assembly will take the following form:

```
load    s4,     s0              ; iTemp0[0]=foo[0]
load    s5,     s1              ; iTemp0[1]=foo[0]
add     s4,     0x2a            ; iTemp0[0]+=42[0]
addcy   s5,     0x0             ; iTemp0[1]+=42[1]
                                ; iTemp1=iTemp0
sub     s4,     s2              ; iTemp1[0]-=bar[0]
subcy   s5,     s3              ; iTemp1[1]-=bar[1]
                                ; baz=iTemp1
```

Listing 6.5: Assembly output compiled from code in Listing 6.4

Each byte in the operation is covered by the regular C-style array notation.

Moves on temporary variables to next variables in a more complex expressions are optimized out, too. They are marked in the assembly to point out that another variable now resides in the particular registers.

Function Comments

Every function is prepended with a comment that states its name with a list of its arguments. Every argument's name is written in the list along with the registers it is stored in, in case it is not stored on stack directly.

Demonstrated on an example:

```
char func(long arg1, int arg2, char arg3);
```

Listing 6.6: Example function to be compiled

```
; Function func, arguments:
[arg1:{s0,s1,s2,s3,},arg2:{s4,s5,},arg3:{s6,},]
```

Listing 6.7: Explanatory assembly comment before the label generated for Listing 6.6

Chapter 7

Port Features

7.1 Code Clarity

The extensibility and readability of this port is not comparable to any other port. The code generation itself takes only about 5 lines of code for each `ICode`, compared to tens to hundreds in PBCCv2(see Section 5.3).

Providing a C++ API also helps by providing more syntax sugar for anybody who wants to further modify the compiler. Using class methods instead of macros and functions also adds the possibility to use an IDE[1] with method suggestion.

7.2 PicoBlaze-6 Support

There is no other C compiler designed to produce code exclusively for the newest revision of the PicoBlaze processor.

The most modern ones support only PicoBlaze-3 so they miss the opportunity to save not only the program space (meaning less CPU cycles for the same program) but also the scratchpad memory and registers (which means the program will utilize less CPU cycles again).

One of the cases when this is true is multi-byte variable comparison - equality check had to be done using many jumps and storing intermediate results, the new `COMPARECY` reuses the carry and zero values that are already in the flags.

7.3 Built-in Functions

The necessity to communicate with connected peripherals is satisfied by using the `char port_in(char port)` and `void port_out(char port, char value)` built-in functions.

They are compiled into `OUTPUT`, `OUTPUTK` and `INPUT` instructions, respectively. `OUTPUTK` is used when it is possible. That is, when both operands are constant and the port is in the range `0x0 - 0xF`.

As they are not really functions, when compiled to the assembly, there is no need to use the registers designated for function calling. The first available registers are used instead of `s0` and `s1` (as specified in Section 6.6.1).

[1] Integrated Development Environment

Chapter 8

Testing and Evaluation

It is necessary to evaluate the overall success of a compiler project by testing it and comparing it to the alternatives. In this section, the testing and comparison methods are described.

8.1 Testing

Several example source files were compiled and ran in a simulator to prove the resulting assembly corresponds to the input source code.

8.1.1 Tools and the Testing Process

`PBlazSIM` (described in Section 2.7.1) was chosen to simulate the resulting code due to its simplicity, maturity and liveliness of the upstream developers.

Obtaining the Simulation Result

The simplicity of `PBlazSIM` cleared the way for modifying of the (small) codebase and including it as a part of the test-suite in this project.

The only modification done is as follows: When the simulation ends because of an error or reaches the correct final state of the simulation (this happens once it the `BREAK` internal opcode is processed), it prints the state of the processor to the standard output.

The output is formatted for easy parsing and use in the attached test suite.

Assembler

The input to `PBlazSIM` are rich assembly language files enhanced with binary representation of the code it is about to simulate. This format is compiled using `PBlazASM` (see Section 2.6.1) using the `l` flag and results in a file with `.LST` extension.

There were no special modifications required to get the needed results except the ones mentioned in Section 2.6.1.

8.1.2 The Framework

All scripts are ran recursively for each `.c` and `.tst` file in the `test` directory in the root of the code tree.

Every `.c` file has to have its `.tst` counter-part with the same base name to be considered a valid test.

It tests the memory and the registers for presence of any value given (with the exception of omitted zero values, considering major part of the memory will be free in most cases), be it on a specifically given position or anywhere.

It is also possible to test `ZERO` and `CARRY` flags of the processor.

Test File Format

The `.tst` file format requires every such file to include declaration of every of the following variables:

`stored_somewhere`	Array, bounds undefined. Each value contained in it is looked for in the memory and register dump. The value can be contained anywhere.
`stored_there`	Array of 256 values. If there is a value X on position Y, the memory has to contain value X on position Y too.
`bankA`	Array of 16 values. Contains a value for each of the registers in the bank.
`bankB`	Analogic to `bankA`, for bank B.

The variables are declared and defined in pseudo-C style (without the type specification), that means: enumerations take a brace-enclosed list as their initializer and Boolean values can be either `true` or `false`.

Every definition has to be ended with a semicolon. Double definitions will result in a failing test.

For example, the following declaration will check all memory locations to contain the values 1, 2 and 3:

```
stored_somewhere = {1, 002, 0x3};
```

And the following declaration says register `s2` has to contain value `0xFF` - other registers are omitted (because they are equal to zero).

```
bankA = {0, 0, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

8.1.3 Test Cases

This is the list of the included and applied test cases that were also ran and checked in the simulator.

`null`	Does not compute anything, tests if the whole stack works
`basic`	Basic variable assignment
`arithmetics`	Addition and subtraction
`array`	Array (pointer) data assignment and reading
`bitoper`	Binary operations on the variables
`cond_simple`	Simple condition evaluation
`cond_complex`	Complicated condition and cycle evaluation
`func_simple`	Basic function calling
`func_recursive`	Recursive function calling
`func_ptr`	Function pointer calling - functionally equivalent to `func_recursive`
`struct`	Structure operations

The code of the tests is completely synthetic, with no real world purpose. Some variables had to be defined as `volatile`, to avoid their optimization which would remove some of them completely, defeating the purpose of testing the mentioned features.

8.2 Evaluation

Due to the fact neither the source code of PBCC by Bohumil Nováček nor PCComp is publicly available, also considering the simplicity of the other compilers, only the previous PBCC projekt by Jakub Horník was used in comparison (see Chapter 5 for more details).

The same test cases were used as in the functionality testing (described in Section 8.1).

The tests were designed to be compileable in all other compilers, except the cases when an unavailable feature is being tested.

The quality metric (in case the program compiles and is valid) was chosen to be the count of used instructions.

test case	PBCCv3 (this project)	PBCCv2 (Horník)
`basic`	✓(**23**)	✓(62)
`arithmetics`	✓(**19**)	✓(26)
`array`	✓(**70**)	×(internal error)
`bitoper`	✓(**29**)	✓(49)
`cond_simple`	✓(**56**)	✓(87)
`cond_complex`	✓(**33**)	✓(53)
`func_simple`	✓(**35**)	✓(51)
`func_recurs`	✓(**44**)	×(infinite recursion)
`func_ptr`	✓(**51**)	×(no support)
`struct`	✓(55)	✓(**52**)

✓*(instruction count)* for succeeded cases, ×*(reason)* for failed cases

Table 8.1: Test results

PBCCv3 gives shorter code for most test cases, except the cases when PBCCv2 fails. In PBCCv2, cases `array` and `func_ptr` failed in the compilation phase and no output

was produced. The test case `func_recurs` compiled fine. However, when simulated, the processor got stuck in an infinite call loop and stopped eventually.

The only case when PBCCv2 gives shorter code is for the `struct` test case. This is caused by the slight overhead needed for calling a function between register banks (explained in Section 6.6.3).

Chapter 9

Conclusion

The back-end project was started from scratch except for some output wrapper code, therefore the expected functionality was not a complete C compiler. The implemented subset is able to compile a wide range of test applications.

Considering the produced code quality, the compiler is better, compared to the alternatives. On the applied test set, the produced code is on average 33% shorter than the code compiled with the previous PBCC. It is also more understandable by the extensive usage of generated explanatory comments.

If taken as an exercise and exploration of the limits of the processor, the project succeeded, too. Function pointers are really usable and testing applications utilizing them work correctly. There is also basic support for pointer assignment which is then utilized in array and structure usage.

9.1 Future Development

The compiler was tested on synthetic cases only. Real world complex applications are needed to be tested with the compiler to have potential issues or inefficiencies fixed.

Some features lack at this moment, too, such as an implementation of integer division and multiplication or interrupt table generation.

The back-end was also designed to be partially portable to other front-ends. It could be used with LLVM, for example, to achieve the ability to compile programs written in other languages. too.

Bibliography

[1] Alfred V Aho et al. *Compilers: principles, techniques, & tools*. Vol. 1009. ISBN 0-321-48681-1. Pearson/Addison Wesley, 2007.

[2] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. pub-ANSI, 1999. URL: http://webstore.ansi.org/ansidocstore/product.asp?sku=ANSI%2FISO%2FIEC+9899%2D1999.

[3] Ken Chapman. *PicoBlaze 8-bit Microcontroller for CPLD Devices*. Xilinx Ltd. 2003. URL: http://www.xilinx.com/support/documentation/application_notes/xapp387.pdf.

[4] Ken Chapman. *PicoBlaze 8-bit Microcontroller for Virtex-II Series Devices*. Xilinx Ltd. 2003. URL: http://www.xilinx.com/support/documentation/application_notes/xapp627.pdf.

[5] Ken Chapman. *PicoBlaze for Spartan-6, Virtex-6 and 7-Series (KCPSM6)*. Xilinx Ltd. 2014. URL: http://www.xilinx.com/ipcenter/processor_central/picoblaze/member/KCPSM6_Release8_31March14.zip.

[6] Ken Chapman. *PicoBlaze KCPSM3: 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-IIPRO*. Xilinx Ltd. 2003. URL: http://www.xilinx.com/ipcenter/processor_central/picoblaze/member/KCPSM3.zip.

[7] *GCC, the GNU Compiler Collection*. URL: http://gcc.gnu.org.

[8] Allen I Holub. *Compiler design in C*. ISBN 0-13-155045-4. Prentice Hall, 1990.

[9] Jakub Horník. "Compiler Back-End of Subset of Language C for 8-Bit Processor". MA thesis. Brno University of Technology, 2011.

[10] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[11] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. pub-ISO, 2011. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

[12] Peter Knaggs and Stephen Welsh. *ARM: Assembly Language Programming*. Bournemouth University, 2004. URL: http://www.eng.auburn.edu/~nelson/courses/elec5260_6260/ARM_AssyLang.pdf.

[13] Pablo Bleyer Kocik. *PacoBlaze*. 2007. URL: http://bleyer.org/pacoblaze/.

[14] Zbyněk Křivka. *Pbsim Github Repository*. URL: https://github.com/krivka/pbsim.

[15] Zbyněk Křivka. *PicoBlaze C Compiler*. URL:
http://www.fit.vutbr.cz/research/view_product.php.en?id=126¬itle=1.

[16] Zbyněk Křivka and Jiří Šimek. *PicoBlaze Instruction Simulator*. 2013. URL:
http://www.fit.vutbr.cz/research/prod/index.php.en?id=309.

[17] *LLVM Compiler Overview*. URL:
https://developer.apple.com/library/iOs/documentation/CompilerTools/Conceptual/LLVMCompilerOverview/index.html.

[18] Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. ISBN 987-3-540-72612-8. Springer, 2007.

[19] Robert Morgan. *Building an Optimizing Compiler*. ISBN 1-55558-179-X. Elsevier Science, 1998.

[20] Bohumil Nováček. "C Compiler for PicoBlaze Microcontrollers". BA thesis. Czech Technical University in Prague, 2008. URL:
https://dip.felk.cvut.cz/browse/pdfcache/xnovaceb_2008bach.pdf.

[21] *pBlazASM*. URL: http://www.mediatronix.com/pages/pBlazASM.

[22] *pBlazASM Google Code Repository*. URL: http://code.google.com/p/pblazasm/.

[23] *pBlazSIM*. URL: http://www.mediatronix.com/pages/pBlazSIM.

[24] *PicoBlaze 8-bit Microcontroller*. URL:
http://www.xilinx.com/products/intellectual-property/picoblaze.htm.

[25] Francesco Poderico. *Picoblaze C Compiler: User's Manual 1.1*. 2005. URL:
http://www.ux.uis.no/~karlsk/ELE610/dok/pccomp_manual.pdf.

[26] *SDCC - Small Device C Compiler*. URL: http://sdcc.sourceforge.net.

[27] *Spartan-3E FPGA Starter Kit Board Design Examples*. URL: http://www.xilinx.com/products/boards/s3estarter/reference_designs.htm.

[28] *The C Preprocessor*. 2014. URL: http://gcc.gnu.org/onlinedocs/cpp/.

[29] *The LLVM Compiler Infrastructure*. URL: http://llvm.org/.

[30] *Welcome to Native Client*. URL: https://developer.chrome.com/native-client.

Appendix A

CD Contents

The attached CD contains the source code, with SDCC included, the tests and installation and compilation scripts.

Appendix B

Installation and Usage

SDCC lists the following dependencies: Boost, Yacc, Bison.

To compile the `pblaze` port, you have to use a C++ compiler that supports the ISO/IEC 14882:2011 (C++11) standard.

The port can be built against any recent snapshot of SDCC and version 3.4.0. It is possible to compile it against SDCC 3.3.0, too but on some machines, `configure` does not generate the required `Makefile`. The CD includes version 3.3.0 with the `Makefile` already generated.

If you compile the project from new source directly, please follow the steps listed in Section 6.1 now.

To run the compiled binary (especially on `merlin.fit.vutbr.cz`), you need to link against the correct `libstdc++` on runtime. To do so on `merlin`, add the following path to your `LD_LIBRARY_PATH`:

```
/pub/tmp/gcc/gcc-4.9.0/.x86_64-linux/x86_64-linux/libstdc++-v3/src/.libs
```

B.1 Usage

To use the `pblaze` port, specify the `-mpblaze` command line option to the `sdcc` binary.

To change the size of the function arguments to be passed via registers, use the `--argreg=N` option, specifying the count of the registers.

Appendix C

PicoBlaze-6 Instruction Set

r - register, p - address in register, cX - constant, size, l - label

LOAD	r	r	Moves contents of registers
LOAD	r	c	Moves contents of registers
STAR	r	r	Moves contents of registers between banks
AND	r	r	Binary and
AND	r	c	Binary and
OR	r	r	Binary or
OR	r	c	Binary or
XOR	r	r	Binary xor
XOR	r	c	Binary xor
ADD	r	r	Addition
ADD	r	c	Addition
ADDCY	r	r	Addition, reuses carry
ADDCY	r	c	Addition, reuses carry
SUB	r	r	Subtraction
SUB	r	c	Subtraction
SUBCY	r	r	Subtraction, reuses carry
SUBCY	r	c	Subtraction, reuses carry
TEST	r	r	Binary and without result storage
TEST	r	c	Binary and without result storage
TESTCY	r	r	Binary and without result storage, reuses carry
TESTCY	r	c	Binary and without result storage, reuses carry
COMPARE	r	r	Operand comparison
COMPARE	r	c	Operand comparison
COMPARECY	r	r	Operand comparison, reuses carry
COMPARECY	r	c	Operand comparison, reuses carry
SL0	r		Shift left, fill 0
SL1	r		Shift left, fill 1
SLX	r		Shift left, fill LSB
SLA	r		Shift left, fill carry
RL	r		Rotate left

r - register, p - address in register, cX - constant, size, l - label

SR0	r		Shift right, fill 0
SR1	r		Shift right, fill 1
SRX	r		Shift right, fill LSB
SRA	r		Shift right, fill carry
RR	r		Rotate right
REGBANK	A		Select bank A
REGBANK	B		Select bank B
INPUT	r	p	Read data from a port
INPUT	r	c8	Read data from a port
OUTPUT	r	p	Write data to a port
OUTPUT	r	c8	Write data to a port
OUTPUTK	r	p	Write constant data to a specific port
STORE	r	p	Write data to scratchpad memory
STORE	r	c8	Write data to scratchpad memory
FETCH	r	p	Read data from scratchpad memory
FETCH	r	c8	Read data from scratchpad memory
DISABLE INTERRUPT			Disable interrupt
ENABLE INTERRUPT			Enable interrupt
RETURNI DISABLE			Disable return from interrupt handler
RETURNI ENABLE			Enable return from interrupt handler
JUMP	l		Jump to label
JUMP	Z	l	Jump to label if zero
JUMP	NZ	l	Jump to label if not zero
JUMP	C	l	Jump to label if carry
JUMP	NC	l	Jump to label if not carry
JUMP@	p	p	Jump to label pointer
CALL	l		Call label
CALL	Z	l	Call label if zero
CALL	NZ	l	Call label if not zero
CALL	C	l	Call label if carry
CALL	NC	l	Call label if not carry
CALL@	p	p	Call label pointer
RETURN			Return from call
RETURN	Z		Return if zero
RETURN	NZ		Return if not zero
RETURN	C		Return if carry
RETURN	NC		Return if not carry
LOAD&RETURN	r	c	Load a constant and return
HWBUILD	r		Get harware information