Building a C-based processor

S. Woutersen

Software Technology
Information Technology & Systems
TU Delft

December, 2005

Abstract

Until today every compiler has been developed by the idea to modify source code in such a way the hardware can understand it. Over the years this has resulted in inefficient processors caused by backwards compatibility issues, strange assembly code constructions caused by the lack of required instructions and nice instructions supported by hardware, but never used by the software.

This research reverses the original design process. It starts by analyzing C code and starts working from there to a processing unit, thus supporting a minimal amount of instructions required to run C and no backward compatibility with other processing units.

To limit the design time, several existing retargetable C compilers are analyzed whether they have a useful intermediate language which is completely hardware independent. The `LCC` compilers `LCC Bytecode` was found to be the most acceptable language, even though a few modifications must be made to be able to efficiently run the code.

The next step is to convert the generated `LCC Bytecode` into a binary language. This is done by a standard set of programs including an assembler and a linker. This is the first time some input about the processing unit is required, since neither C nor `LCC Bytecode` specifies anything about, for example, opcode encoding, while this can make a big difference in execution performance. For now the encoding is simply "guessed" and remains open for future modifications.

The last step in the chain is the actual processing unit. A software interpreter programmed in C is build to extensively test compiled programs, and to compare the results with other existing solutions, such as the *MIPS R2000*, the *Intel 386 Architecture* and the *Intel 8051*.

Test results show that `LCC Bytecode` is a fine language which covers C completely. However some modifications where necessary to execute the code efficiently. The number of instructions required by `LCC Bytecode` is much smaller than those supported by most current processors, and therefore, a less complex, and thus cheaper and more reliable, processor design is possible. In terms of speed it is found that on average more instructions per C program are required than when using, for example, the MIPS instruction set, the processor will thus probably be slower than most traditional processors.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A few decennia ago, the first compiler was launched, while today, there are thousands of compilers available for dozens of programming languages. Even though a lot has changed since the first compiler, one thing is still the same. The design process always started out from two different points. One group started working on designing a programming language, and a complete other group, often much later or earlier, started working on a processing unit. When both where done, a bridge, in the form of a compiler, must be build between the designed programming language and language supported by the processing unit. This often resulted in strange inefficient code constructions to support various aspects of the programming language, as well as completely unused processing unit instructions.

This research takes another approach [3]. It's starts with looking at C code, and works towards the specifications of a not yet existing processing unit. The processing unit will therefore be completely be based on C, will support no instructions C does not support, and does not need any "tricks" to run any C program. The reason for this new approach is the fact that nowadays, processors are much easier to develop, and hardware limitations are no longer a real issue. Therefore, it is now possible to base the hardware on the code it is supposed to execute, instead of altering code in a way such that the hardware can execute it. Finally, the new processing unit, which will be designed for this research in the form of an interpreter, will be compared to existing processing units, to find out if this will benefit the size, complexity and speed of the new processing unit, compared to existing ones.

The second chapter of this report describes the search for a useful compiler. It looks into a few different existing compilers and compares them between each other, and a not yet existing but maybe required, newly developed compiler. Chapter 3 documents the assembly language which is used, followed by Chapter 4 which gives a detailed description on how the executable files are generated, and how the interpreter works and is created. Finally, in Chapter 5 and 6, the test results and conclusions are given.

Chapter 2

Generating assembly code

When designing a processing unit to execute C code, the extreme situation would be feeding the processing unit directly with C source and header files. The C language however has a few properties which makes it much easier to program, but much harder to execute. A few examples of these properties are the support for text labels, code placement in several files, and less obvious, redundancy in C code. The `for` statement, for example, may just as well be programmed using `while` or even `if` statements. A processing unit which can execute C code would thus require a complicated text parser, a lot of memory and a label resolver. Most of the work, such as label resolvement, can be done prior to executing the code, and only has to be done once for each program. Traditionally this is all done by compilers which generate simple assembly code, assemblers which convert assembly code into binary files and linkers who combine several files into one executable. Since simpler code will always result in a simpler processing unit, it is wise to stick with the traditional use of a compiler. The conversion process however, must be completely based on C code, and not be influenced by any knowledge about the processing platform which is eventually going to execute the code. The compiler must thus be completely independent of the targeted platform.

Since building a new compiler is a lot of work, this chapter mostly deals with looking for a useful existing compiler. Several compilers are examined in the first section of this chapter, and finally a conclusion is given which, if any, compiler would be the best choice for generating assembly code.

2.1 Compilers

Since there is a vast amount of compilers available, the search to a useful compiler must be narrowed a lot right at the start. Only a few compilers can be examined thoroughly, and for most compilers it can be seen in a flash whether they might be up to the task or absolutely not. The criteria for the compilers to be examined further are given here:

- Open source: The compiler must be open source to be able to examine its internals.
- Free: No money can be spent on the compiler.

- Portable: The compiler must be designed to be portable. This will ensure the compiler can be used for different kind of processing units, and thus (likely) be hardware independent[1].
- Documented: The compiler (and its source code) must be very well documented.
- Reliable: The compiler must be reliable and work correctly.

These criteria immediately eliminate a large amount of compilers. Most small "hobby" projects are not portable or very well documented, while most commercial products are not open source, free or target a specific platform. However, three existing compilers seem to hold up quit nice to these criteria, and will be examined further, namely `GCC`, `LCC` and `SDCC`.

These compilers will be looked into in the following sections, to see if one of them is able to generate valid hardware independent assembly code. The `GCC`, `LCC` and `SDCC` compilers are further discussed in Sections 2.1.1 to 2.1.3 respectively, and a comparison is given in 2.2.

2.1.1 GCC

The `GNU C Compiler` is probably the most popular compiler today. It runs on almost any system, and can create code for almost every system. Since it is so widely used, a lot of information is available on how it works, and how to alter it if necessary. However, Nilsson [12], states:

> `GCC` is specifically aimed at CPU's with several 32-bit general registers and byte-addressable memory. Deviations from this are possible. In short, you can make a port of `GCC` for a target with 16-bit registers, but not a decent implementation for a processor with only one general register.

This directly states that `GCC` is only partially hardware independent, since it needs a register machine to operate efficiently. Internally, `GCC` uses a so called "Register Transfer Language" or "RTL". First, the C code is converted into RTL code, in which an infinite amount of available registers exists. Second, the RTL code is optimized to a specific processor, and finally, the code is converted to processor specific assembly code. The last two steps can be stripped from the compiler such that RTL code is being emitted from the compiler. This RTL code is fairly hardware independent except for one assumption: the target processing unit must be a register machine.

2.1.2 LCC

The second compiler which is tested, `LCC`, is designed to be a completely retargatable compiler, no more, no less. Since any code optimization falls out of this project, this might be the perfect candidate. Just like `GCC`, `LCC` uses an internal language to which C code is converted, and contains several backends which convert the `LCC bytecode` to assembly code for a specific processor. Unlike `GCC RTL`, `LCC bytecode` is based on a stack language. Since any (useful)

[1] This must be validated in further research

processing unit will have some random access memory available, a stack based language will not limit the design of the processing unit (apart from performance perhaps).

A second advantage of LCC is that it's internal bytecode can be emitted without changing the source of LCC at all, but simply with a command line option which selects the "bytecode" backend.

2.1.3 SDCC

The Small Device C Compiler is a C compiler developed specially for building embedded applications, which run on small processors or microcontrollers. It is completely open source but unfortunately not very well documented. According to the SDCC programmers in [14] it is possible to build a new backend for their processor, and point out a few changes on where to modify the existing compiler. No real standard retargeting procedure exists, and nobody but the programmers themselves seems ever to have done it. Another problem with SDCC, is the original design concept: it was build to be specifically a compiler for embedded systems, which means it had a "hardware assumption" right from the start.

2.2 Conclusion

To summarize the previous sections, a few important properties of all compilers are given here, including a mark on how well they perform, from positive (++) to negative (--).

Table 2.1: Compiler comparison

Property	GCC	LCC	SDCC	New custom compiler
Documentation	++	+	--	++
Retargetable	++	++	-	++
Internal language	+	++	?	++
CPU dependency	0	+	0	++
Work to retarget	0	++	-	--
Work to build CPU	0	+	?	++

From this table it can be seen that either LCC or a custom self build compiler would be the best choice. The self build compiler wins it on every aspect because it can be tuned to all of these specifications; however, the double minus on work to build the compiler is a very important one. GCC loses from LCC mostly because LCC bytecode files are generated much easier then GCC RTL files, and because LCC bytecode files can be run on any processing unit without major modifications, while when using GCC RTL, at some point, the register count must be scaled down to an acceptable amount of registers. SDCC scores lowest on this chart, mostly caused by the lack of good documentation about its internals.

From this information, LCC is chosen to be the compiler, and LCC bytecode to be the new assembly language to work with. In the following chapter, the specifications of the LCC bytecode language are fully documented.

Chapter 3

LCC bytecode

This chapter describes the layout of the bytecode files generated by `LCC` and all assembler directives and instructions which may appear in the source files. The information used to make this document is gathered from Hanson and Fraser [5], [6], the Quake 3 implementation of `LCC Bytecode` [13] and the `comp.compilers.lcc` newsgroup.

`LCC Bytecode` consists of a set of instructions and assembler directives. All these instructions and directives are placed in a text file, one instruction/directive per line. The directives are printed in lower case, while the instructions are printed in upper case. Some directives and instructions have one ore more parameters. These parameters are separated by spaces, and all numerical parameters are printed in the decimal system. Note that this chapter is completely based on what `LCC` will ever generate. When building an assembler, it might be desirable to accept more then just what is described in this chapter, for example comment in bytecode files.

All directives are described in Section 3.1, all instructions in 3.2 and finally, in Section 3.3 some shortcommings of the `LCC Bytecode` language are discussed.

3.1 Directives

The following directives are generated by `LCC`:

`code` : segment information
`lit` : segment information
`data` : segment information
`bss` : segment information
`align` : segment information
`byte` : segment information
`skip` : segment information
`address` : segment information
`export` : procedure information
`import` : procedure information
`proc` : procedure information
`endproc` : procedure information
`file` : debug information
`line` : debug information

All directives are printed in lower case, while all instructions are printed in upper case. The different directives fall in three main categories. First, the directives `code`, `lit`, `data`, `bss`, `align`, `byte`, `skip` and `address` directives all control the different segments used by `LCC`. These are all discussed in Section 3.1.1. The `export`, `import`, `proc`, and `endproc` directives contain function specific information and are described in Section 3.1.2. Finally the `file` and `line` directives contain debug information and are described in the last section.

3.1.1 Segments

`LCC` supports four different segments for code and variables:

- `code`: All instructions are placed inside the code segment. This segment can be mapped onto read-only memory.
- `lit`: All constants (literals) are placed in the lit segment. This segment can also be mapped onto read-only memory.
- `data`: All initialized variables are placed in the data segment. This segment must be mapped onto writable memory.
- `bss`: The BSS[1] segment contains the uninitialized variables. This segment must also be mapped onto writable memory. No data should be stored in the bss segment prior to execution.

In a bytecode file, the start of a new segment is noted by a single line, with just the name of the segment. Note that some directives which apply to an item in a specific segment, may fall outside that segment. For example; the `export` directive applies to a specific function, but it might be placed outside the `code` segment.

While the `code` segment will merely consist of instructions, the `lit` and `data` segment contain preset values and labels, while the `bss` segment contains only labels. The labels themselves are generated as instructions and are therefore handled in Section 3.2.1.

`LCC` uses three directives to fill a segment with data. The first is `byte`. The `byte` directive has two decimal parameters containing the amount of bytes, and its numerical value of the data which should be added to the segment. The `byte` directive is mostly used to add constant strings. The second one is `skip`. This directive does not add any data to a segment, but just allocates space. The decimal parameter holds the number of bytes to be allocated. The `skip` directive is used to allocate uninitialized data in the `BSS` space. Finally, the `address` directive loads an address in a segment. A label name is passed as parameter.

3.1.2 Functions

`LCC` does not create any function prologue or epilogue, instructions for building a function stack frame do not exist. Instead, the start and end of a function is noted by two assembler directives, optionally combined with an `export` directive, which denotes a function that may be called from other files;

[1] Block Started by Symbol, originally an opcode, nowadays used as segment name

```
export <functionname>
proc <functionname> <localstack> <argumentstack>
...
endproc <functionname> <localstack> <argumentstack>
```

The three parameters functionname, localstack and argumentstack are the same for both the `proc`, and the `endproc` directive, and no function nesting is allowed. The `functionname` parameter is a string which contains the name of the function, exactly the same as it appears in the C code. The `localstack` parameter is a decimal number which denotes the total number of bytes occupied by the local variables (and thus the amount of space on the stack in bytes which needs to be reserved for local variables). Finally, the `argumentstack` parameter, also a decimal number, contains the maximum amount of space required for the arguments to functions which are called from the current function (not the total size of the arguments passed to *this* function). The `LCC` bytecode generator assumes stack space for variables is reserved for these arguments at the top of a function stack frame, and that arguments to functions called from within the current function are placed here. For example;

```
int add(int a, int b) {
    return a+b;
}
int main() {
    int sum;
    sum = add(3, 5);
    return sum;
}
```

is compiled into:

```
export add
code
proc add 0 0
...
endproc add 0 0
export main
proc main 4 8
...
endproc main 4 8
```

The stack frames right after the entrances of the two functions, main (on the left) and add (on the right), are shown in Figure 3.1.2. All stack frames in this chapter grow downwards, and only relevant items are shown. In this case, it is chosen to place the function arguments before the local arguments, this is however not required by `LCC Bytecode`. Both the room for arguments and local variables are suspected to be removed from the stack whenever the program leaves the current function.

The procedure main needs four bytes for local variables in the example (`LCC` defines an integer to be four bytes long), and eight bytes (two integers, for the two parameters to procedure add) for arguments.

If a function is called but not defined in the source file (e.g. a call to a library function), `LCC` adds the directive `import` to the bytecode file:

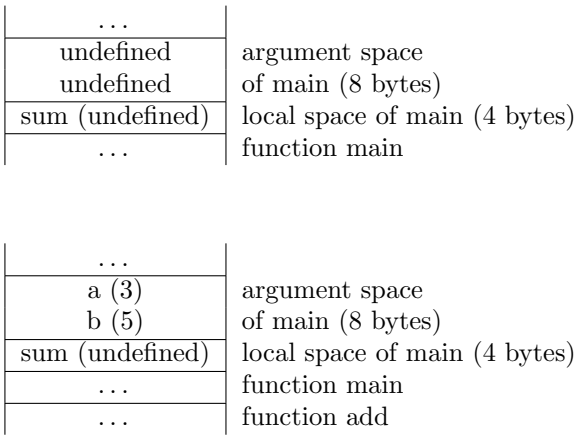

Figure 3.1: Stack frames after function entrances

```
import putchar
```

When including header files, all functions which are declared in the header file are imported. It might be desirable to check whether those functions are ever called before including them in the final executable.

3.1.3 Debug information

When `LCC` is passed the '-g' parameter, it starts adding debug information to its output files. The amount of information written however is very small and consists of only two extra directives: the `file` and the `line` directive. The `file` directive has one parameter, containing the file from which the following code is compiled, while the `line` directive denotes the line in the current file.

Unfortunately `LCC` does not generate any debug symbol information such as function return value types, local variable names etcetera.

3.2 Instructions

The `LCC Bytecode` language is based on a stack language. All (arithmetic) operators are executed on the top stack arguments. However, the stack is still assumed to be fully accessible, since locals and arguments are stored higher on the stack.

Each instruction consists of at least an opcode and a value type. Most instructions are also appended with a value/parameter size (in bytes) and some instructions have a parameter. The opcode, type and size are directly appended, if a parameter is required for the instruction, the instruction and parameter are separated by a space. The possible opcode types can be found in Table A.1. The structure and void type both do not have a size (for structure, it is unknown, for void it is zero). The addition instruction for two unsigned 32-bit integers thus becomes: `ADDU4`. Returning a 32-bit pointer will be `RETP4`, while returning a void will result in `RETV`. Not all instruction-type-size combinations are allowed, for example, additions of characters is done by first converting them to integers,

and do an integer addition. `ADDI1` therefore will never occur, but also the `MULP4` will never be generated by `LCC` since the multiplication of pointers is not supported by C.

In Appendix A a table is given with all possible combinations of opcode, type and size.

3.2.1 Labels

Labels are generated by the compiler as instructions. While all other instructions will only ever appear in the `code` segment, the `LABEL` instruction can occur in any of the four segments. An example of the declaration of global 32 bit integer i:

```
export i
align 4
LABELV i
skip 4
```

By default, the `LCC` aligns every variable, except for characters, by 4. The `skip` directive is the directive which actually allocates the space for the variable.

3.2.2 Loading and storing variables

Within compiled programs, variables can be stored at three different locations: in the local space of a function, in the argument space of a function, and in the global space of a program. For each of these, different instructions exist for loading and storing them.

A variable load consists of two instructions. The first one computes the address of the variable and puts it on top of the stack, the second one reads the location from the stack and copies the data from the location to the top of the stack. The first instruction is different for variables from all three spaces: `ADDRG` for global variables, `ADDRL` for local variables and `ADDRF` for parameters. The variable offset is stored in the parameter of the instruction. The base offset of local and global variables and parameters should always be known by the execution unit (as it can not be computed in advance). When, for example, three local variables are declared, `i`, `j` and `k`, the variable offset for `i` is zero, the offset for `j` is the size of `i`, and the offset for `k` is the size of `i` plus the size of `j`. The base offset in this case is the beginning of the local space of the function in which `i`, `j` and `k` are defined. In case of a global variable, the parameter of the instruction is not numeric, but instead a string containing the name of a label created with the `LABEL` instruction. In any case, the parameter may be extended by `+x` or `-x` where `x` is a decimal number containing an offset relative to the first parameter:

```
ADDRGP variable+12
```

is valid bytecode, it should add the address of variable, plus 12, to the stack.

Fetching the second function argument (when the first argument is 4 bytes long) is done with the following code:

```
ADDRFP4 4
INDIRI4
```

The stack frames for this piece of code (right after the `ADDRF` and the `INDIR` instructions) are shown in Figure 3.2.2 (which corresponds to fetching argument `b` from the example used in Section 3.1.2). The `ADDRF` instruction computes the absolute address of the argument, by adding 4 to the argument space offset (in this case 0x10). The result (0x14) is stored on the stack. The `INDIR` instruction pops the address, and replaces it with the actual value found on this address.

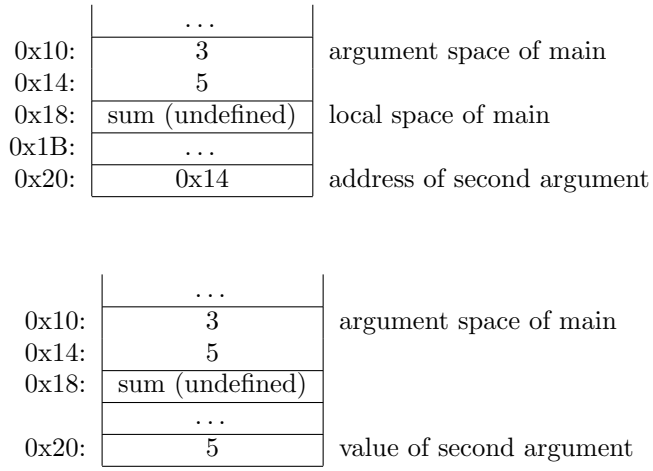

Figure 3.2: Stack frames for fetching variables

In the same way; fetching local variable *sum* inside the main procedure would be done with the following code:

```
ADDRLP4 0
INDIRI4
```

Storing variables in the local or global space of a program is done the same way, except that the `INDIR` instruction is replaced by the `ASGN` instruction, and the value to store is placed on the stack right after the address. The code

```
ADDRLP4 0
CNSTI4 10
ASGNI4
```

assigns the value 10 to the first declared local variable (which should be a 32-bit integer). The stack frames for assigning variables are shown in Figure 3.2.2. This code corresponds to setting variable `ret` from the previous example to constant 10.

Storing variables in the argument space is a bit different; instead of `ASGN`, the `ARG` instruction is used. The `ARG` instruction denotes the top of the stack should be moved to the argument space of the function. Unfortunately, the `ARG` instruction does not have a parameter, instead, the assembler or execution unit itself should hold a counter on where to store arguments. The first occurrence of `ARG` should move the top of the stack to the top of the argument space of the current function, the second occurrence should be placed right after the first. The counter can be reset on a function call (after a call, parameters to that function are no longer used). Again from the previous example:

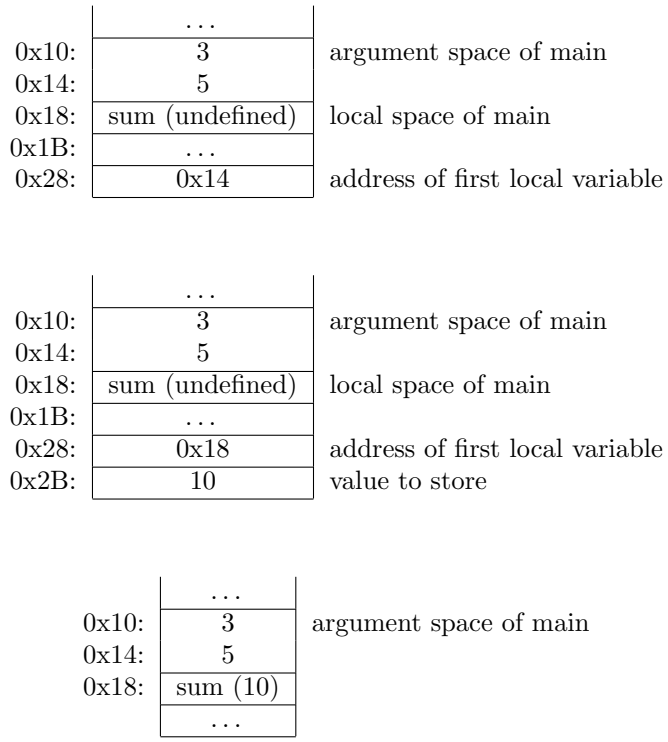

Figure 3.3: Stack frames for assigning variables

```
CNSTI4 5
ARGI4
CNSTI4 3
ARGI4
```

loads the constants five and three into the argument space of the procedure main. The arguments are now ready for the function call to `add` (the stack frames from before and after this piece of code are similar to Figure 3.1.2).

Finally, constants, as seen in the previous example, are loaded through the `CNST` instruction. The parameter of this instruction will hold the constant. Floating point values are first casted to integers (such that the binary representation of the integer is the same as the binary representation of the floating point number according to IEEE 754). Double precision floating point values are placed in two four byte integers. String constants are placed in the `lit` segment rather than the `code` segment, and are loaded using the `ADDRG` and `INDIR` instruction.

3.2.3 Calling to and returning from functions

A function call is made with the `CALL` instruction. `CALL` does not take any parameters, instead, the address of the function is assumed to be on top of the stack. The type and size of the `CALL` instruction are the type and size of the return value. The code

```
sum = add(3, 5);
```

is translated into

```
ADDRLP4 0
ADDRGP4 add
CALLI4
ASGNI4
```

Because of the `ASGN` instruction, the top of the stack after the `CALL` instruction should contain the address of `sum`, generated by the `ADDRL` instruction, and the return value of `add`, generated by the `CALL` instruction. The four stack frames taken right after the four instructions are shown in Figure 3.2.3.

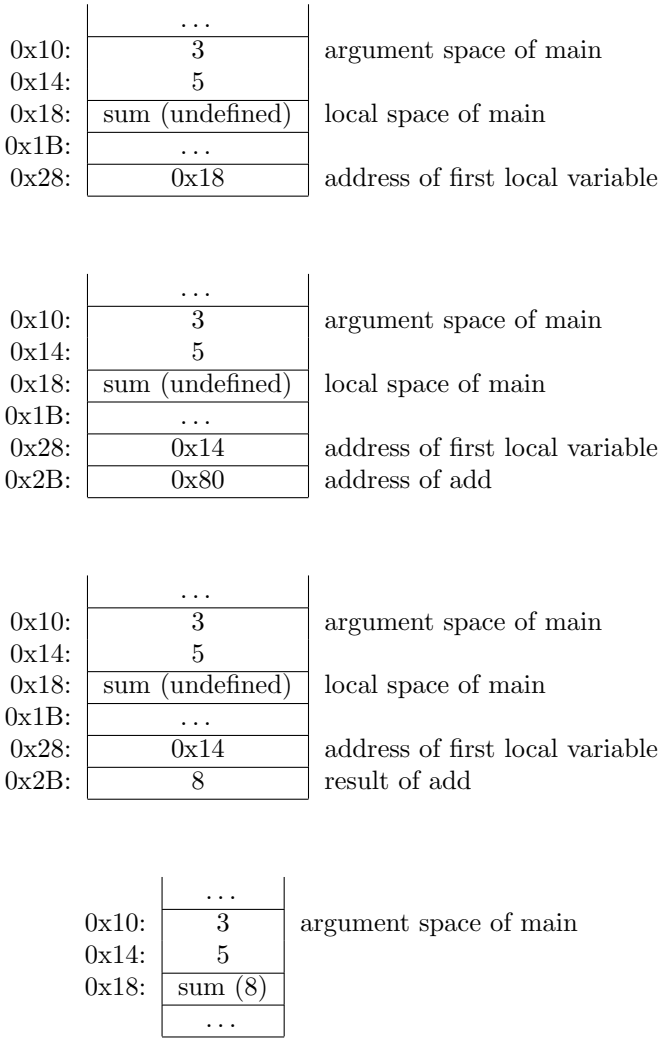

Figure 3.4: Stack frames for calling functions

To return a value from a function, `LCC` uses the `RET` instruction. The type and size of the `RET` instruction are those of the return value, and should be those

of the `CALL` instruction which called the function. The `RET` instruction uses the top of the stack as a return value. The `RET` instruction must make sure the stack and processing unit control registers looks just like it was before the function call, but with one extra value on the stack: the return value.

Returning a constant 32-bit integer with the value 10 simply looks like this:

```
CNSTI4 10
RETI4
```

Supplying a function with arguments is done by writing them to the argument space as is discussed in the previous section.

3.2.4 Arithmetic operators

LCC bytecode supports all arithmetic operations which are supported by C. Most take two parameters and generate one result, meaning it takes two arguments from the stack, and put the result back on. Some other instructions simply alter one parameter, in this case the stack size remains the same. The arithmetic operators supported by *LCC bytecode* are depicted in table 3.1 All arithmetic

Table 3.1: LCC Bytecode arithmetic operators

Instruction	Number of parameters	C operator
`ADD`	2	`+`
`SUB`	2	`-`
`MUL`	2	`*`
`DIV`	2	`/`
`MOD`	2	`%`
`RSH`	2	`>>`
`LSH`	2	`<<`
`BAND`	2	`&`
`BOR`	2	`|`
`BXOR`	2	`^`
`BCOM`	1	`!`
`NEG`	1	`-`

instructions only work on the "larger" values, for integer values `int`, `long` and `long long`. The smaller versions, for integer values `short` and `char`, the value is first converted to a 32-bit integer. The stack frames right before, and right after a `SUB` instruction, are shown in Figure 3.2.4.

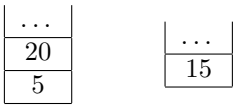

Figure 3.5: Stack frames for arithmetic operations

3.2.5 Program jumps

No program could exist without the help of (conditional) jumps. *LCC bytecode* supports seven types of jumps, six conditional and one unconditional.

The unconditional jump, the `JUMP` instruction and has no parameter. The address to which to jump is instead taken from the top of the stack. The six conditional jumps are depicted in table 3.2 They all have one parameter which

Table 3.2: LCC Bytecode conditional jumps

Instruction	C operator
EQ	==
GE	>=
GT	>
LE	<=
LT	<
NE	!=

contains a label to jump through if the condition evaluates to positive. The two parameters of the condition are taken from the top of the stack.

```
CNSTI4 3
CNSTI4 4
GE SOMELABEL
```

Will not jump to `SOMELABEL`. However:

```
CNSTI4 3
CNSTI4 4
NE SOMELABEL
```

will. All if statements, while loops, for loops, etc. will result in a bytecode construction which uses one or more (conditional) jumps. `LCC` will, if necessary, add extra labels named as a dollar sign followed by an unique number. For example, the following code:

```
01: int max(int a, int b) {
02:     int ret;
03:     if (a > b) {
04:         ret = a;
05:     } else {
06:         ret = b;
07:     }
08:     return ret;
09: }
```

will compile into:

```
proc max 4 0
line 1
line 3
```

```
ADDRFP4 0
INDIRI4
ADDRFP4 4
INDIRI4
LEI4 $2
line 4
ADDRLP4 0
ADDRFP4 0
INDIRI4
ASGNI4
line 5
ADDRGP4 $3
JUMPV
LABELV $2
line 6
ADDRLP4 0
ADDRFP4 4
INDIRI4
ASGNI4
line 7
LABELV $3
line 8
ADDRLP4 0
INDIRI4
RETI4
LABELV $1
endproc max 4 0
```

3.2.6 Variable type conversions

The last set of instructions handles the conversions of different variable types. These include the conversion from one type to a larger or smaller version of the same type, or the conversion between types. Instructions generated by `LCC` to convert types are called `CV*`, where * stands for the type it is being converted *from*. This can be `F` for floating point, `I` for signed integer, `P` for pointer and `U` for unsigned integer. The type and size of the instruction denote the type and size of the value it needs to be converted *to*. The size of the value before it is converted is stored in the instructions parameter. Conversion instructions always convert the value placed on top of the stack.

Not all conversions are supported. Converting one type to another may require multiple conversions. The possible combinations can be found in Appendix A.

Several conversions are displayed here:

```
signed char to int:
CVII4 1
signed char to float:
CVII4 1
CVIF4 4
```

```
pointer to double:
CVPU4 4
CVUI4 4
CVIF8 4
```

(note that pointer to double conversion is not directly supported by C, however, if it would, it would look like this)

3.2.7 Structures

Structures in `LCC Bytecode` are not separated in several primitives, but considered a special type. Only a few instructions can have the structure type, as can be seen in Table A.2, these instructions are `INDIR`, `ASGN`, `ARG`, `CALL` and `RET`. However, `LCC` supports *not* passing structures to, and *not* returning them from functions. Instead, pointers to structures are passed. Since this option is default turned on, the `ARGB`, `CALLB` and `RETB` instructions will never appear in bytecode files.

The `INDIR` and `ASGN` instructions are only used to copy one structure to another. Every occurrence of `INDIR` will be followed by an `ASGN` instruction. Copying a 16 byte global structure `a` to another global structure named `b` looks like this:

```
ADDRG b
ADDRG a
INDIRB
ASGNB 16
```

Note that the size of the structure is only given as parameter of the `ASGN` instruction.

`LCC` Requires that the first values within a structure also get the lowest memory addresses, since `LCC` uses pointer additions to compute the addresses of different values within structures.

3.3 Shortcommings

Unfortunately, `LCC Bytecode` has a few small shortcommings which makes it impossible, hard or just inefficient to run on a processing unit. It's very important to keep track of these shortcommings when building a processing unit for `LCC Bytecode`. Most of them can very simply be fixed by modifying the existing `LCC` backend, or by modifying the bytecode files slightly after they are generated by `LCC`. In the following paragraphs, each of these problems are discussed, as well as one ore more methods to fix them.

3.3.1 The LABEL instruction

The `LABEL` instruction is for some reason implemented as an instruction instead of a directive as it should be. No processing unit will ever do something when it encounters a `LABEL` instruction, instead, the information the instruction holds should be used by the assembler and or linker.

3.3.2 The ARG instruction

The `ARG` instruction is used to place an argument in the argument block to pass it to a function. However the `ARG` instruction has no argument denoting the location (or parameter index). Therefore, any processing unit must have an extra counter to keep track on where to place arguments, while this can be solved a lot easier by giving the `ARG` instruction a parameter containing its location.

3.3.3 Structures

`LCC Bytecode` has its own type for structures. Therefore, structures of infinite large size are copied with only one instruction. This will be impossible to implement on any hardware system, but it is the only possible way to keep moving structures around hardware independent. This must most likely be changed when hardware specifications are available.

The real problem with structures is the fact that a structure copy is done by two instructions: `INDIR`, which fetches the structure and `ASGN`, which writes the structure to another location. However the size of the structure is not known (the `B` type never has a size). For the `ASGN` instruction, this is solved by adding the size of the structure as a parameter, however this is not done for the `INDIR` instruction. A structure copy of a 16 byte structure thus looks like this:

```
INDIRB
ASGNB 16
```

This is a problem since the size of the structure must be known when fetching it from the memory (by the `INDIR` instruction). Since each `INDIRB` is followed by a `ASGNB` instruction, the assembler should look ahead when it encounters the `INDIRB` to find the size the `INDIRB` instruction should work on.

3.3.4 Debug information

Although `LCC` does support the '-g' parameter, hardly any debugging information is generated. When debugging is turned on, two new directives are added to the bytecode files: `file` and `line`, which are described in Section 3.1. To be able to efficiently debug programs, much more information is required.

3.3.5 Not using the return value

There is one shortcomming of the *LCC bytecode* that can actually be considered a bug in `LCC`. Consider the following peace of code:

```
int add(int a, int b) {
    return a + b;
}
int main(int argc, char** argv) {
    add(1,2);
    return 0;
}
```

which is perfectly good C code. The bytecode generated for this peace of code looks like this:

```
proc add 0 0
...
RETI4
endproc add 0 0
proc main 0 8
...
ADDRGP4 add
CALLI4
CNSTI4 0
RETI4
endproc main 0 8
```

It can be seen that the return value, which is put on the stack by the `CALLI4` instruction is not used in some subsequent instructions, and is therefore left on the stack forever. In the previous example this wouldn't be much of a problem, but the code

```
while(1) add(1,2);
```

will eventually cause an out of stack error, while this code should keep running forever.

This problem has been posted to the `comp.compilers.lcc` newsgroup, and several `LCC` users have confirmed this is a bug in `LCC`, but no response from the programmers has been posted to date (January 2006).

Chapter 4

Implementation

As stated earlier, C code has four important properties which are hard or inefficient for an execution unit to do: combining source files and libraries, label resolving, parsing and redundancy. With the use of `LCC` as compiler, the last property has vanished, and parsing is made a lot easier. Still any processing unit will find it much easier to work with one binary file instead of multiple text file. Therefore an assembler and linker are used to respectively convert the assembly files into binary assemblies and combine multiple assemblies into one executable file. This chapter contains a description on the modifications made to `LCC`, and the new programs created to assemble, link and execute `LCC bytecode` in the first seven sections. The last section contains several points which are not yet implemented, but may need to in the future.

4.1 Overview

The total design path from C source file(s) to the executable file is shown in Figure 4.1. All C sources are first compiled into `LCC bytecode` files by the `LCC` compiler described in the previous chapter. A small number of minor modifications to the `LCC bytecode` files are made by the `Bytecode converter`, and is described in Section 4.3. The assembling (converting the bytecode text into a binary form), and linking (joining several sources into one executable file) is done in two separate programs; the assembler and the linker. Therefore, the process remains transparent, and a small change in one source file doesn't require the entire program to be recompiled/reassembled. The assembler and linker are described in Sections 4.4 and 4.5.

4.2 The LCC bytecode generator

Unfortunately the `LCC bytecode` has to be slightly altered to be a good assembly language. To stay compatible with new versions of `LCC`, the choice has been made to make no critical changes to the `LCC bytecode` so `LCC` can be updated without breaking the system. Most changes to the `LCC bytecode` which have to be made are therefore moved to the next program in the code generation phase, and will be discussed in the next section.

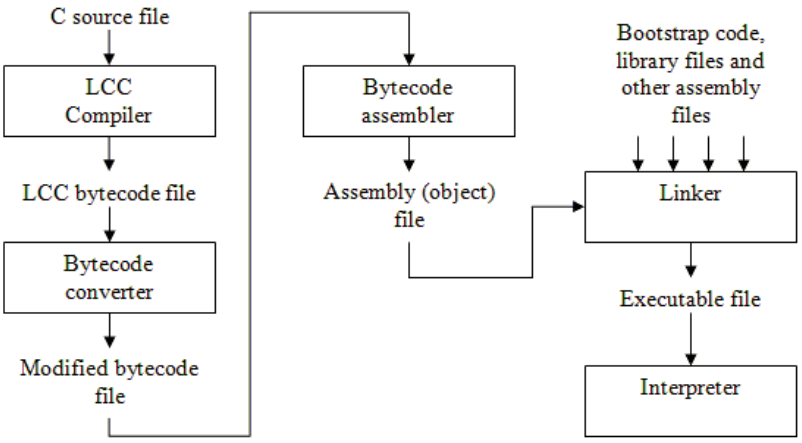

Figure 4.1: Design path

The first problem with `LCC bytecode`, the 'return value left on stack' problem discussed in the previous chapter, can only be solved inside `LCC` itself. Only then is known whether the value is ever going to be used again (or a complete code analysis program has to be written). Luckily, the problem can be solved pretty easily with only a single line of code. `LCC` has uses a forest [5] to internally represent the parsed C code. A forest consists of trees of DAGs [5], [4]. A simple tree for the statement `z = x + y` is shown in Figure 4.2. For function calls two types of trees exist: those for when a return value is used, and those for when a return value is not used. In Figures 4.2 and 4.2 two function call trees are shown. In the first figure, the result is stored in `z`, in the second, the result is discarded. It can be seen that when the return value is not used, the `CALL` instruction is the root of the tree. When the return value *is* used, the `CALL` instruction will *never* be the root of the tree.

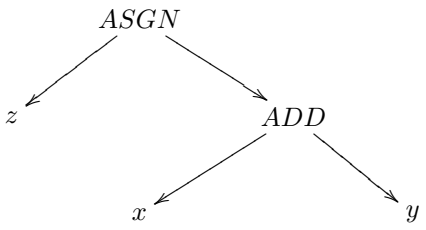

Figure 4.2: Tree for z = x + y

Inside `LCC`, a loop iterates through all tree roots and emits code for them. The roots themselves recursively emit code for their children. The code-emit function for roots is thus isolated for the code emit function for any other tree node. The loop is placed in the `I(emit)` function in the code generator, and is modified as follows:

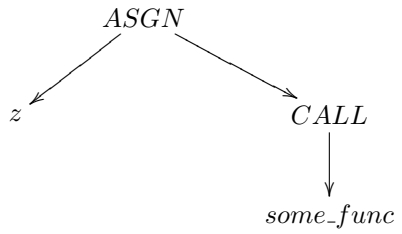

Figure 4.3: Tree for z = some_func()

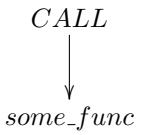

Figure 4.4: Tree for some_func()

```
dumptree(p);
```

to:

```
dumptree(p);
if (generic(p->op) == CALL && optype(p->op) != VOID) {
  print("DISCARD%s%d\n", suffixes[optype(p->op)], opsize(p->op));
```

As can be seen, a new instruction `DISCARD` is introduced which pops a value of the stack, and throws it away. The results obtained by this minor modification can be found in Section 5.1.1. Note that the `DISCARD` instruction is not printed when calling a `VOID` function.

4.2.1 Debug symbols

Another problem with `LCC` which could also not be handled outside of `LCC` was the poor generation of debug symbols. Since most of the info required to generate useful debug symbols is only known within `LCC`, the bytecode generator itself must be modified when debug symbols are required. Even though debug symbols are completely ignored by the assembler, the `LCC` bytecode generator has already been modified to generate them.

Because the original debug information generation functions in the `LCC` bytecode generator do not provide enough information to generate good debugging symbols, the real code generation functions are hacked to emit debugging information before, for example, emitting a local variable.

The new bytecode generator emits three new types of debugging information: information about the return value of functions, about the type of local variables, about the type of global variables and about the type of function parameters. They all start with a keyword (directive), which is either `local`

for local variables, `global` for global variables, `function` for function return values and `parameter` for function parameters. After the directive, the name of the variable is printed. In case of a parameter or a local variable, the name is prefixed by the name of the function followed by a colon. The variable type is printed after the variable name, and placed between triangular brackets. It consists of the type of the variable followed by the size of variable between square brackets. When a variable type is of the pointer type, the type which the pointer points to is also resolved, and `pointer[m] to` is added in front of the variable type.

For the following piece of C code:

```
int gbl;

int main(int argc, char** argv) {
    int lcl;
    /* some code */
}
```

The following debugging symbols are generated:

```
global gbl <integer[4]>
function main <integer[4]>
local main:lcl 0 <integer[4]>
param main:argc <integer[4]>
param main:argv <pointer[4] to pointer[4] to integer[1]>
```

The debugging symbols are generally placed before the actual code or directive which creates the label, the assembler should however allow debugging information to be placed anywhere in the file.

Note that each local variable gets an extra numeric parameter. This parameter contains the location of the variable in the function's stack frame. No such argument is added for parameters since within `LCC` it is not known, and can only be found by counting arguments and summing their sizes[1]. For global variables the locations are also not emitted, but their labels are preserved during the compilation and assembling process.

4.2.2 Updating LCC

Since multiple applications may depend on `LCC`, the original bytecode generator is left untouched, but instead, a copy was made. All the code for the bytecode generator is located in `src/bytecode.c`. A copy of this file is made, and is called `src/xbytecode.c`. Before attaching it to `LCC`, the name of the `Interface` structure must be changed to `xbytecodeIR`, or any other unique name: The line

```
Interface bytecodeIR = {
```

is replaced by:

```
Interface xbytecodeIR = {
```

To attach the backend to `LCC`, one line must be added in `src/bind.c`:

[1] The bytecode converter described in Section 4.3 computes the location of arguments, the converter can easily be converted to add these locations to the debug symbols

```
xx(xbytecode,    xbytecodeIR) \
```

can be placed right under the original

```
xx(bytecode,    bytecodeIR) \
```

Off course the `Makefile` must also be altered to compile and link the new source file.

4.3 The bytecode converter

The bytecode converter is a very small and simple program which only modifies a few `LCC bytecode` directives and instructions. It basically fixes a few "glitches". All these modifications could have been done by the assembler, but to honor the traditional roll of the assembler, and to keep the process transparent, the modifications are made by a different program. The modifications to the bytecode the converter makes are listed here:

- Replace procedure directives with real stack frame creation instructions
- Replace the `LABEL` instruction with a `label` directive
- The `ARG` instruction gets a parameter containing the parameter's offset
- The `INDIRB` instruction gets a parameter containing the size of the structure

Most of these items are hot fixes for `LCC bytecode` shortcommings described in Section 3.3. One new item is the replacement of procedure directives by real instructions. This is done to simplify the assembling process. The instructions which are currently used for stack frame creation are completely based on the working of the interpreter, and may need to be changed for a real hardware implementation. The actual instructions used to create function stack frames can be found in Section 4.6.3.

4.4 The assembler

The *assembler* is a small program which converts the bytecode into binary code, and handles all `LCC bytecode` directives. The conversion is a simple one-to-one conversion, meaning that theoretically the exact source file can be reproduced when disassembling the binary assembly file, only debug information and comment will be lost. Except for the `code` segment, the binary coding of the segments is exactly done as described by the assembler directives described in Section 3.1. The binary coding of the `code` segment, as well as the format of the assembly file can be found in Appendix D.2. All labels are left intact within the assembler, thus all addresses must be computed within the linker. All addresses within the binary code are replaced by unique identifiers, which correspond to entries in a table containing all labels. For example: when calling function `add`, the parameter of the `ADDRG` instruction (which is used to get the address of `add`), is replaced by an unique identifier which corresponds to a table entry containing the actual address and segment of `add`. To find all unique identifiers

in the binary code, a bitmap pattern is generated. Each bit represents one byte, and each bit being one represents that byte being part of an unique identifier (and thus should be replaced by the actual address. The linker should thus scan the bitmap for an unique identifier, read it, find the address and segment within the table, and update the code with the final (absolute) address. An example of the assembling process can be found in Appendix F. Debug information is currently discarded within the assembler. All directives introduced in Section 4.2 are accepted as valid input, but nothing is done with them yet.

4.5 The linker

The linker joins all object files into one executable file. In addition it also computes all label locations and updates the segments with these locations. In general the linker walks through the following steps:

- Load all files into memory, store them in `Assembly` structures
- Check all labels, and connect labels pointing to code in other assemblies
- Compute the combined size of the four segments (see Section 3.1.1)
- Compute all label locations
- Update the binary segment data
- Write all to segments to a single file

In addition to the assemblies given as input files, the linker may add several other assemblies if required. These are discussed in the following sections.

Upon execution, the entire executable should be copied to the CPU's memory. The program assumes it is placed on address 0, if not, the address of each memory operation must be incremented by the programs base address. The `CODE` segment is the first segment in the executable, directly followed by the `LIT` segment and the `DATA` segment. The `BSS` segment is again not written to the final executable, since it would only contain junk. Instead, a dummy label with the name `$endprog` is introduced in the linker who points to the first byte beyond the `BSS` segment. The location of this label can be used to correctly place the stack or heap within the processing unit's memory. Note that the label starts with a dollar sign, meaning the label is only accessible through bytecode, and will never interfere with any existing C variables or functions. The stack can be placed anywhere in memory, but needs to grow upwards in memory (see Section 4.6.2). The stack can therefore be placed right after the program, while all dynamically allocated data objects can be placed at the end of the memory.

4.5.1 Bootstrap code

The bootstrap code is a small piece of code placed at the very beginning of the executable file. The bootstrap code is merely a bridge between the startup of the processing unit, which will start executing at address 0, and the startup of the C code, which will start at the call to the main function. A simple bootstrapping code can thus exist of only a few lines of code calling main.

The standard bootstrap code which is used for this project contains in addition the setup of the two standard parameters to the main function: `argc` and `argv`. They will not contain any useful information, but they will be valid for any C program using them.

The bootstrap code also contains code to allocate the `BSS` segment. Since the stack grows upwards (see Section 4.6.2) on default, it is placed in memory right after the executable file, which is placed at address zero. To save space, the `BSS` segment is never included in the executable file, and the stack, when started directly after the program's data, will overlap the `BSS` segment. The bootstrap code starts by moving the stack beyond the `BSS` segment. To do this, some variables must be placed on the stack which thus overwrites the `BSS` segment. This is not a problem, since uninitialized variables contain junk when the program starts anyway. To move the stack a new instruction `NEWSTACK` is introduced, which moves the stack to the location pointed by the value on top of the stack. Any values left on the stack before using `NEWSTACK` are no longer valid after a `NEWSTACK` instruction.

4.5.2 Library files

Some functions may be so widely used they need to be put in library files. Most C compilers provide the programmer with a standard library containing a lot of functions for I/O, mathematical calculations, etcetera. Library files can be created and linked (see Appendix E.4) by the linker. A library file consists of several assembly files, which are added to the executable when required. If one library assembly needs another library assembly it is also automatically linked, however an assembly within a library is always added completely to the executable. When the functions `printf` and `putchar`, for example, are placed in a library in the same assembly, `printf` is always added to the executable when `putchar` is. However, when they are put in the same library, but in different assemblies, the use of `printf` automatically imports `printf` and `putchar` (assuming `printf` uses `putchar`), but the use of `putchar` won't import `printf`.

4.6 The interpreter

The interpreter is a program which interprets and executes binary instructions generated by the assembler. The engine of the interpreter, the part which actually executes the instructions, is separated from the user interface. This is done so the interpreter can also be embedded in other applications for testing or debugging purposes. How to embed the interpreter is described in Appendix B. The layout of the interpreter is shown in Figure 4.6. The interpreter is started from `interpreter.c` which continuously calls `execute` in `interpreter_engine.c`, which executes one instruction. I/O required by programs running on the interpreter is handled by the operation system specific `io.c`. The `execute` function relies on several sub functions which handle, for example, comparison, arithmetic and memory operations. These operations are all mapped on the native operations supported by C, the OS or the running processor.

The first section of this paragraph describes the interpreter engine, how programs are loaded and executed. The second section describes the stack,

how it is built and maintained. Finally, Section 4.6.3, describes the few new instructions the interpreter supports.

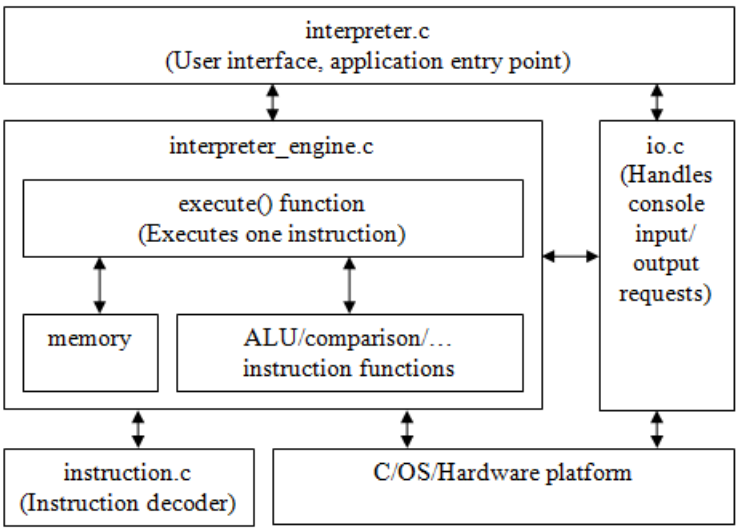

Figure 4.5: Interpreter

4.6.1 The interpreter engine

The engine of the interpreter is the part of the interpreter which loads programs into its memory, and executes the instructions. It also has several helper functions to convert variables as they appear in the interpreter memory to C variables, which can be used by programs embedding the interpreter for maximum control.

The interpreter engine uses an `Interpreter` information structure which holds the state of the interpreter, and a pointer to its memory. The `create_interpreter` is used to create a new interpreter information structure. The desired size of interpreter memory is passed to this function, and can not be changed afterwards. Loading an executable into the interpreter's memory is done by the `load_program` function, which accepts a file pointer. The file is loaded into the lower part of the memory (address 0-...), and thus all precomputed pointers in the program are also valid for the interpreters memory. However, this makes it impossible to load multiple programs on the interpreter.

The interpreter always starts executing at address 0, which should hold some bootstrap code which calls the `main` function. See Section 4.5.1 for more information about the used bootstrap code. The `execute` function executes the instruction pointed to by the program counter: `PC`. The `execute` function returns an error code (defined in `src/interpreter_engine.h`) if the instruction could for some reason not be executed, `ERR_EXIT_PROGRAM` when the `HALT` instruction is encountered or `ERR_NOERROR` when the instruction was executed successfully.

4.6.2 Stack frames

`LCC` makes a few assumptions about the stack frame when generating `LCC bytecode`, and to be able to correctly execute `LCC bytecode`, the stack frame must be designed by these assumptions. The assumptions are listed here, some are already discussed in previous paragraphs, and some others are new.

- Space for local function variables is allocated when entering a function.
- Space for function arguments for *callees* is allocated when entering the *caller* function.
- The first value in a structure is assumed to have the lowest absolute memory address of all values in a structure (see Section 3.2.7).
- Because of the previous assumption, the first of all local variables, and the first of all arguments, must also have the lowest absolute memory address.
- A function return instruction should rollback the stack to the point right before the call, with just the return value added: from the caller's point of view, a `CALL` instruction acts just like a `CNST` instruction.

The stack frame is further optimized, to require as small possible amount of pointers as possible. The first pointer which is required for any stack is a pointer pointing to the top of the stack. This pointer is from now on called the stack pointer, or `SP`. The design is started by assuming some values already exist on the stack, and a function call is encountered. The `CALL` function must write the return address to the stack, since after the call, the return address is no longer known. Right after a function call, the stack frame thus looks like Figure 4.6.2.

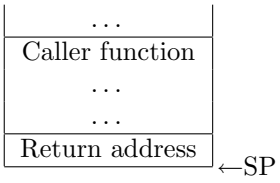

Figure 4.6: Stack frame step I

Note that the stack frame here grows downwards, this can either be to a higher or a lower memory address. Next, space for both the arguments to subfunctions and space for the function's local variables must be allocated. To be able to access these areas, pointers must be set to the lowest address of these spaces. These pointers are called the local pointer (`LP`) and the argument pointer (`AP`). An additional pointer is needed which denotes the beginning of a function's stack frame. This pointer is called the frame pointer, or `FP`. Since it is not yet known whether the stack frame should grow up in memory, or down in memory, in the following pictures some pointers appear twice, once with an up pointing arrow, to denote this pointer is required when the stack grows up in memory, or with a down pointing arrow, which is required when the stack grows down. This is shown in Figure 4.6.2

Note that placing the argument block above the local block is a random choice, and they can be exchanged when needed.

Since the pointers must be able to be restored on function return, they must also be placed on the stack. These should be stored before the argument and local blocks, because the creation of these blocks also generate the new pointers, as shown in Figure 4.6.2

The pointers are saved in the processing unit state block. The state block is accessible through the `FP` pointer, and thus no new pointer is required. Since the state block has a fixed size, the location of `FP` can be reached from `AP` when the stack grows upward in memory. Therefore, one of those registers can be left out when letting the stack grow upwards in memory. The result is show in Figure 4.6.2.

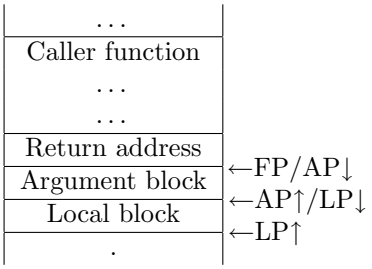

Figure 4.7: Stack frame step II

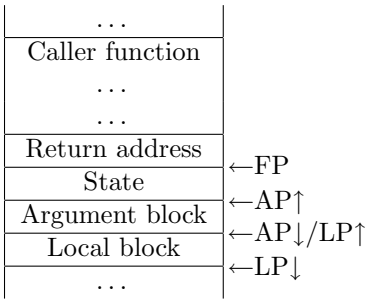

Figure 4.8: Stack frame step III

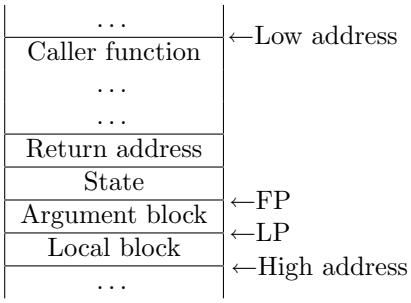

Figure 4.9: Stack frame step IV

The argument pointer is reintroduced by letting it point to the argument block of the previous function. This location is required to be able to load the arguments passed to the current function. The final stack frame is shown in Figure 4.6.2. All stack frames for the example introduced in Section 3.1.2, are shown in Section 4.7.

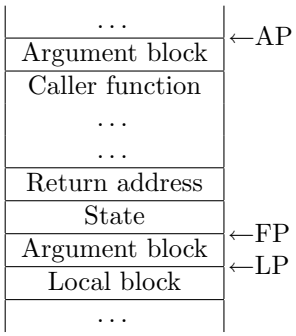

Figure 4.10: Final stack frame layout

4.6.3 New instructions

The interpreter has support for five new instructions, which can also be found in Table A.2. These instructions are:

- `DISCARD`: discard (pop) top of stack.
- `HALT`: stop the interpreter, is used by the default `exit()` function.
- `SAVESTATE`: function preamble, save the entire state of the processor to the stack (used on function calls). The `SAVESTATE` implementation on the interpreter saves the argument pointer (AP), the local pointer (LP) and the frame pointer (FP) in this order to the stack.
- `ARGSTACK`: function preamble, allocate space (amount of bytes in parameter) for arguments to sub-functions. Should be the second instruction of every function (right after `SAVESTATE`) and must be followed by `VARSTACK`. The `VARSTACK` implementation on the interpreter copies (besides allocating the space for arguments) the frame pointer to the argument pointer, and the stack pointer to the frame pointer (see Section 4.7). Because of the pointer change, this instruction must always be executed, even when the size of the argument stack is zero.
- `VARSTACK`: function preamble, allocate space (amount of bytes in parameter) for local variables. Should be preceded by `ARGSTACK`. The interpreter copies the current stack pointer to the local pointer on execution of this instruction (see Section 4.7). This instruction must therefore again always be executed when entering a function.
- `SYSCALL`: calls a system function.

An example of the usage of the `SAVESTATE`, `ARGSTACK` and `VARSTACK` instructions can be found in Section 4.7.

The `SYSCALL` instruction is introduced to interface with the environment, for example to read and write characters from and to the console, or read some interpreter properties. The `SYSCALL` instruction takes one or two parameters from the stack, and always places one parameter back. The top of the stack should be a constant 32-bit integer denoting the type of system call to be made. All parameters and return values of the `SYSCALL` instruction are 32 bit integers. The following types are supported:

- `0x01` read and return one character from the console (blocking).
- `0x02` write one character to the console, returns character written (takes one extra 32 bit integer value from the stack).
- `0x03` read and return one character from the console (non-blocking). Returns -1 when no character is present.
- `0x04` generate and return a random number.
- `0x05` return the number of instructions executed.

The interpreter has a special I/O library rather than using standard C functions directly. The main reason for this is the support for non-blocking reads. The standard `getchar` function provided by the C library does not return until a character is written. This is very nice on a multitasking environment, but when simulating an embedded application this is highly non-desirable.

The I/O library, stored in `src/IO.c`, supports three functions to access the console: `read_char`, which reads one character, and blocks when no character is available, `read_char_non_blocking`, which reads one character, or returns `-1` when no character is available, and `write_char`, which writes one character to the console. In addition, the functions `init_io` and `term_io` are also called at the startup and exit of the interpreter. Since non blocking reads are not supported by C itself, this is done by directly manipulating the operation system. The I/O library currently supports Windows and Linux, and chooses it's OS at compile-time by checking if the `_WIN32` macro is defined.

4.7 Program stack frame example

This section contains the stack frames which will be generated by the interpreter when running the code `sum = add(a, b);`. The first stack frame, shown in Figure 4.7, is taken right before the function call to `add`. On the top of the stack, the address of add can be found which is needed by the `CALL` instruction (see Section 3.2.3. Before that, the address of the `sum` variable can be found, which is required to assign the result of `add` to `sum` (see Section 3.2.2).

Next, the `CALL` instruction is executed. The address of `add` is taken from the stack, and the return address (next instruction in caller function) is placed on the stack (see Section 3.2.3). The stack frame taken right after the `CALL` instruction is shown in Figure 4.7.

The program counter is now pointing to the first instruction of `add`, which should be a `SAVESTATE` instruction. This instruction saves the `AP`, `LP` and `FP` pointers to the top of the stack, as shown in Figure 4.7.

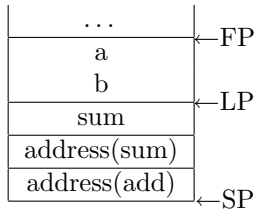

Figure 4.11: Stack frame example Step I

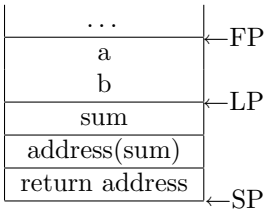

Figure 4.12: Stack frame example Step II

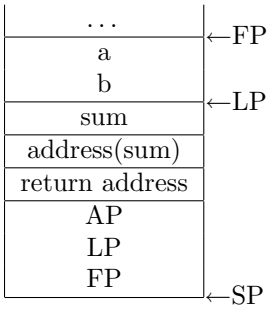

Figure 4.13: Stack frame example Step III

Now that the processor state is saved, a new stack frame must be created using the `ARGSTACK` and `VARSTACK` instructions. The two stack frames taken after each of these instructions are shown in figures 4.7 and 4.7. Note that `add` does not call any other functions, or use any local variables. The parameters of both `ARGSTACK` and `VARSTACK` are therefore zero, and no space is allocated for local variables and arguments to callee functions.

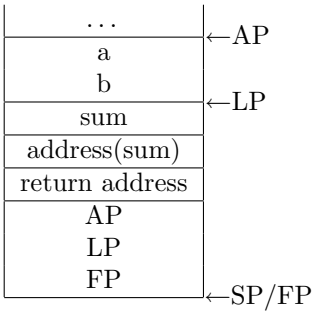

Figure 4.14: Stack frame example Step IV

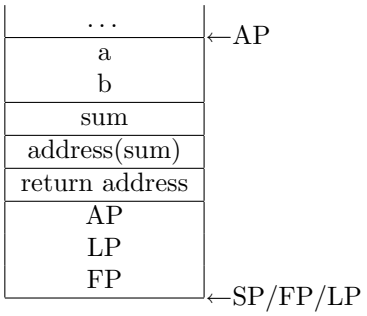

Figure 4.15: Stack frame example Step V

The next stack frame, shown in Figure 4.7, is taken just before the `RET` instruction. The return value is placed on top of the stack.

The final stack frame is shown in Figure 4.7, and is taken right after the return from `add`. All pointers are restored to there original values from before the function call, and the return value is the only result left from the call to `add`.

4.8 Future work

Although many plain C programs can be compiled and executed on the interpreter, some issues are still open for future work. A few of these issues are given in the following list, with a brief description on how this can be accomplished.

- Debugging: debugging is not supported by the interpreter, and debug symbols are discarded by the assembler. To turn the application in a

complete development environment, some way of debugging must be implemented in the interpreter, and therefore debugging symbols must be passed through the code conversion chain to the interpreter.

- Libraries: several standard C library functions are already implemented, but most are not. To increase the power of the application, the entire standard library should be ported to this platform.
- Base address: it is useful to build support for a "base" address for applications making it possible to run them from any point in memory. This will be required for ROMable code, and also to load multiple programs on the interpreter, for example to be able to run a simple operating system.
- Structures assignments: infinitely large structures are copied with only one instruction. No real hardware system will ever be able to do this. Splitting up the structure in parts can both be done by the hardware itself, or the converter/assembler. The second one may be a logical choice since it requires less complicated hardware, but then a feedback from hardware specifications to the converter/assembler is required.
- Function preambles: these are currently created by the bytecode converter, however, these depend heavily on the processing unit's implementation. Some feedback from the hardware specifications might be required to build stack frames for future processing units.

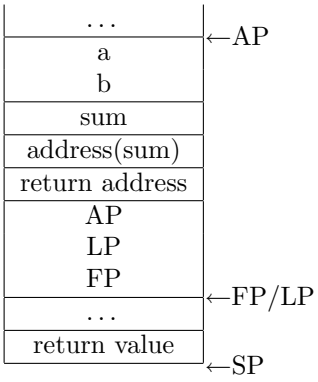

Figure 4.16: Stack frame example Step VI

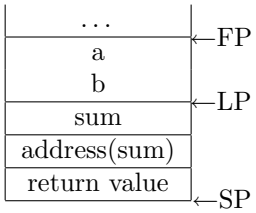

Figure 4.17: Stack frame example Step VII

- Floating point: currently the interpreter does not support the floating point type. Since floating point numbers are stored as decimal values in bytecode files, the assembler and linker can act on the floating point type just as they act on integer types, but support for the floating point arithmetic and conversion instructions must be built in the simulator to fully support floating point operations.
- Beyond 32-bit: the ALU functions used by the interpreter are mapped onto the ALU of the executing processor, meaning that on a 32-bit processor, overflow will occur when executing arithmetic instructions larger than 32-bit.

Chapter 5

Test results

This chapter contains the test results obtained by analyzing compiled programs, executing them and comparing them to programs compiled for other processing units. In addition, the instruction set and register usage for de `LCC Bytecode` interpreter is compared to those of a few existing processing units. The first section of this chapter shows some analysis of the `LCC Bytecode` language. In the second section, the assembly files of programs compiled for different processing units are compared, and in the last section, different processing units are compared to the `LCC Bytecode` interpreter written for this research.

The test programs which are used to generate these results are a simple game `zeeslag` and a fast Fourier algorithm `fft` written by the author, a memory monitor application written by Arjan van Gemund, `mymon`, and the test program which comes with the softfloat library, `float`, written by John R. Hauser.

5.1 LCC Bytecode analysis

In this section, `LCC Bytecode` is further analyzed. In the following sections, the results for the stack-leak fix are shown, and an overview of the occurrence of all instruction types is given.

5.1.1 Results of the DISCARD instruction

As mentioned in section 3.3.5, unused return values stay on the stack forever. To fix this problem, a new instruction was introduced: the `DISCARD` instruction, which discards the top `n` bytes of the stack. This method slightly increases the amount of instructions to be executed, and thus decreases the speed of the application, but it fixes a memory leak. It should therefore be considered a necessary fix, rather than an optimization. Using the `-nd` command line parameter on the interpreter, the `DISCARD` instruction can be ignored (the interpreter will not execute the instruction, nor will it increase the instructions executed count). Several test files which can be found in the projects test directory where executed with and without the use of the `DISCARD` instruction, the results can be found in Table 5.1.

The test files used for these results can all be found in the `test` directory in the projects directory. The `retvaltest` is a test program written to show the

problem, and should not be considered a real program. However, it shows that the `DISCARD` instruction is able to get rid of 99% of the (wasted) stack space at the cost of only a 6% slower program. This program is also not used to compute the final averages.

Table 5.1: Discard instruction results

Program	stack peak decrement	instructions executed increment
retvaltest	99%	6%
zeeslag	49%	0%
mymon	38%	1%
fft	51%	1%
float	92%	0%
averages	58%	1%

5.1.2 Instruction usage

In Table 5.2, the occurrence percentage of each single instruction is shown in percentage. To obtain these results, results from the `zeeslag`, `mymon`, `fft` and `float` programs are combined. In total over a 150 million instructions where executed. It can be seen that memory read and write operations make up for a huge part of an average program. All instructions are tested several times (although some very often compared to others), except for all floating point instructions, which are not yet implemented in the interpreter.

Table 5.2: LCC Instruction frequency

Instruction	Frequency	Instruction	Frequency
ADDRF	7,96%	SUB	0,62%
ADDRG	2,30%	ASGN	10,30%
ADDRL	22,87%	EQ	0,38%
CNST	6,16%	GE	0,72%
BCOM	0,06%	GT	0,30%
CVF	0,00%	LE	0,01%
CVI	4,84%	LT	0,11%
CVP	0,00%	NE	1,06%
CVU	1,88%	ARG	4,36%
INDIR	21,42%	CALL	1,48%
NEG	0,15%	RET	1,48%
ADD	1,99%	JUMP	0,47%
BAND	0,87%	LABEL	0,00%
BOR	0,54%	VARSTACK	1,48%
BXOR	0,09%	ARGSTACK	1,48%
DIV	0,09%	SYSCALL	0,01%
LSH	1,28%	SAVESTATE	1,48%
MOD	0,00%	NEWSTACK	0,00%
MUL	0,53%	DISCARD	0,09%
RSH	1,14%	HALT	0,00%

5.2 Assembly files

In this section, the assembly (text) files are compared for several processing units. `LCC` is used to generate i386 code, MIPS code and `LCC Bytecode`. `SDCC` to generate code for the 8051. In table 5.3 the number of assembly lines for several programs are displayed.

Table 5.3: Assembly file lines comparison

Program	i386	MIPS	8051	LCC Bytecode
zeeslag	1320 (63%)	1258 (60%)	2042 (98%)	2088 (100%)
mymon	3642 (74%)	3259 (66%)	3795 (77%)	4954 (100%)
fft	450 (67%)	424 (63%)	?	670 (100%)
float	3390 (68%)	3236 (65%)	?	4972 (100%)
average (perc.)	68%	64%	?	100%

At first glance, the number of lines in an assembly file may give any information, but when the following things are considered, they do:

- Each line in an assembly file contains one instruction
- Assembler directives can be neglected
- If a program is compiled for platform A and B, and assembly file A contains `n` times as much instructions as assembly file B, the number of instructions executed on platform A will also be roughly `n` times as much as the number of instructions executed on platform B, when the extra instructions are uniformly distributed through the code.

It can be seen both `MIPS` and `i386` assembly code files are about 2/3 the size of the `LCC Bytecode` files. The main reason for this is the fact that both the `MIPS` and the `i386` support reading and writing from and to a memory location relative to a register with only one instruction, where `LCC Bytecode` needs at least two. As can be seen in Table 5.2, these instructions make up for a very large part of any program. Unfortunately, `SDCC` was not able to compile the `float` and `fft` programs, thus no test results are available. This method is chosen over looking at the sizes of the object or executable files, since both can contain a huge amount of information besides the actual code, and are very depended on the size (8-bit 8051, 32-bit i386) of the processor.

5.3 Processing unit comparison

Finally, in this section, several processing units are compared to the written `LCC Bytecode` interpreter.

It can be seen a lot can be won on the complexity of the processing unit when designing it from the viewpoint of C. Both MIPS and i386 are very powerful processor which have come a long way, but clearly support a lot more instructions than necessary to execute just C code, which will definitely make those processors a lot more complicated. One should remember however that the i386 and MIPS where designed to be much more than just a C processor. To

be able to build operating systems like Linux in `LCC Bytecode`, the instruction set must be slightly expanded to support threading and interrupts.

Table 5.4: Instruction count and size

Processing unit	Supported instructions	Instruction size (bits)	General purpose registers	Control registers
MIPS R2000[8]	60+	32	28	4
8051[1]	43	8-24	8[1]	2
i386[9]	100+	?	4	4
LCC bytecode interpreter	38	16-48	0	3

Chapter 6

Conclusions

In Chapter 2, it is shown that `LCC Bytecode` is the best choice for an assembly language, when building a processor based on C in limited time. Although several modifications had to be made to efficiently execute the code generated by the `LCC` compiler, test results show that, if not all, most standard C programs can correctly be converted to `LCC Bytecode` and executed.

A software interpreter was written to execute the generated `LCC Bytecode` programs, and was used to analyze the code generated by `LCC`. To fully support the `LCC Bytecode` language, 31 instructions needed to be implemented, plus seven new instructions for stack frame manipulation and interpreter control. The total number of 38 required instructions is much lesser than the number of instructions supported instructions by most existing processors. Basing a new processor on `LCC Bytecode` will thus result in a much simpler processor, which will in the end result in a smaller, more reliable processor which can be developed in lesser time.

Even though no real speed tests where done, it can be seen from the generated assembly files that the average number of instructions per C application is higher when using `LCC Bytecode` than most other assembly languages. Thus a `LCC Bytecode` processor with the same amount of instructions per second as, for example, a `MIPS` processor, will be slightly slower. This is mostly caused by the fact that most assembly languages allow memory access in one instruction, while `LCC Bytecode` requires two. Since these instructions make up for about one third of an average C program, a lot of instructions are lost here. In terms of instructions per second, the new processor will most likely be a lot slower than most modern processors, which are all mostly developed to be fast, not small or simple.

In conclusion, it is seen from the previous chapters that basing a new processor completely on the C programming language will result in a less complex processor, which makes the development easier and thus faster. In the end this will require smaller (and thus cheaper) chips. When lowering development and product costs have a higher priority than using the fastest processor available, designing a processor based on C might therefore be a better solution than designing a processor based on hardware properties.

Appendix A

LCC bytecode instructions

In Table A.2, quoted from *The lcc 4.x Code-Generation Interface* [6], show all possible instructions generated by the `LCC Bytecode` generator. The first column denotes the opcode as it appears in `LCC Bytecode` files. The second and third column contain the possible type and sizes for this opcode. The size column contains blocks of sizes, each block shows the possible sizes for one type. If an instruction supports two types, two blocks will be found in the size column. Since the structure and void types have no size, they will also have no corresponding allowed size block in the third column. The types and sizes are denoted as letters which are defined in Table A.1. Finally, the last column gives a brief description of the instruction.

The last few entries in the instruction table are not generated by `LCC`, but are added to successfully execute standalone programs.

Table A.1: LCC Instruction type and size suffixes

Type suffix	Meaning (C)
F	Floating point
I	Signed integer
U	Unsigned integer
P	Pointer
V	Void
B	Structure

Size (bytecode)	Size (C)
f	sizeof(float)
d	sizeof(double)
x	sizeof(long double)
c	sizeof(char)
s	sizeof(short)
i	sizeof(int)
l	sizeof(long)
h	sizeof(long long)
p	sizeof(void*)

Table A.2: LCC Instructions

Operator	Type Suffixes	Sizes	Operation
ADDRF	...P..	p	address of a parameter
ADDRG	...P..	p	address of a global
ADDRL	...P..	p	address of a local
CNST	FIUP..	fdx csilh csilh p	constant
BCOM	.IU...	ilh ilh	bitwise complement
CVF	FI....	fdx ilh	convert from float
CVI	FIU...	fdx csilh csilhp	convert from signed integer
CVP	..U...	p	convert from pointer
CVU	.IUP..	csilh csilh p	convert from unsigned integer
INDIR	FIUP.B	fdx csilh csilh p	fetch
NEG	FI....	fdx ilh	negation
ADD	FIUP..	fdx ilh ilhp p	addition
BAND	.IU...	ilh ilh	bitwise AND
BOR	.IU...	ilh ilh	bitwise inclusive OR
BXOR	.IU...	ilh ilh	bitwise exclusive OR
DIV	FIU...	fdx ilh ilh	division
LSH	.IU...	ilh ilh	left shift
MOD	.IU...	ilh ilh	modulus
MUL	FIU...	fdx ilh ilh	multiplication
RSH	.IU...	ilh ilh	right shift
SUB	FIU...	fdx ilh ilhp p	subtraction
ASGN	FIUP.B	fdx csilh csilh p	assignment
EQ	FIU...	fdx ilh ilhp	jump if equal
GE	FIU...	fdx ilh ilhp	jump if greater than or equal
GT	FIU...	fdx ilh ilhp	jump if greater than
LE	FIU...	fdx ilh ilhp	jump if less than or equal
LT	FIU...	fdx ilh ilhp	jump if less than
NE	FIU...	fdx ilh ilhp	jump if not equal
ARG	FIUP.B	fdx ilh ilh p	argument
CALL	FIUPVB	fdx ilh ilh p	function call
RET	FIUPVB	fdx ilh ilh p	function return
JUMP	....V.		unconditional jump
LABEL	....V.		label definition
HALT	....V.		Stop interpreter
ARGSTACK	....V.		Create argument stack
VARSTACK	....V.		Create local variable stack
SYSCALL	FIUP..	fdx ilh ilh p	Call special function
SAVESTATE	....V.		Saves processor state on stack
NEWSTACK	....V.		Creates a new stack
DISCARD	FIUP..	fdx ilh ilh p	Discard top of stack

Appendix B

Embedding the interpreter

The interpreter is fully embedable, meaning it can be linked with any application which can then run code on the interpreter. This feature makes it possible to test only parts of a program on the interpreter, while running the running the rest of the code natively. The source code used in this chapter can be found in the `test/embed` directory of the project.

The first section of this chapter describes how to embed the interpreter in an application: how to link the interpreter with the application and how to start it. The second paragraph describes how to call a function in the embedded code.

B.1 Embedding the interpreter

To embed the interpreter in an existing C program, the header `interpreter_engine.h` which is placed in the projects source directory, must be included. From then, several functions are available to control the interpreter.

The first step is to create a new instance of the interpreter, which must be done by calling `create_interpreter`. This function returns a pointer to an interpreter information structure (`Interpreter*`), or NULL if something went wrong. It takes one argument: the total amount of memory the interpreter can access. This can not be altered later, and must be large enough to hold the program's executable, the uninitialized variables and the programs stack. The interpreter will generate an error when it runs out of memory, so when it runs out, increase the memory size and try again.

When the interpreter is created succesfully, an executable file must be loaded. This is done with the `load_program` function, which takes two arguments: a `FILE*` to the executable and an `Interpreter*` to the interpreter info structure created by the previous function. This function will return zero when it failes (not enough memory or file access violation). From this moment, the interpreter is ready to execute instructions.

The `execute` function executes one instruction and sets the program counter to the next. It returns an error code (see interpreter_engine.h for all codes) when an error occures, or `ERR_NOERROR` when it succeeds. A very simple interpreter (without any error checks) would look like the following:

```
Interpreter* interpreter;
interpreter = create_interpreter(1024*1024);
```

```
load_program(executable_file, interpreter);
while(execute(interpreter) == ERR_NOERROR);
```

To compile your program, three files must be linked with it: `interpreter_engine.o`, `io.o` and `instructions.o`. The first file contains the actual interpreter, the second some extended io functions[1] and the third the instruction decoder.

B.2 Calling a function

The previous example shows how to run a complete program on the interpreter. More usefull is the possibility to run only a part of the program on the interpreter. When embedding the interpreter it is possible to run only one specific function. Several actions must be performed to call a function on the interpreter. These are all described in the following sections.

B.2.1 Gathering information

Since neither the debugger, nor the linker, nor the interpreter supports debugging symbols (yet), it is up to the programmer to pass all relevant information about the function to the interpreter and to make sure this information is correct. At first the complete function declaration must be known: the type and count of the parameter as well as the type of the return value must be passed correctly to the interpreter. This can off course be directly extracted from the running C source file. Finally also the function's location in the executable must be provided. All final addresses are computed during the linking stage of the build, and only after this stage, the addresses are valid. Luckily, the linker can write the addresses of all public variables and functions[2] to the screen while linking the program. This is done by supplying the linker with the -v command line switch. For example:

```
linker -v func_serv.co -lib stdio.cl -o testcode.ce

Public functions:
Name:                   Location:
  exit                  0x0000003A
  mult                  0x00000100
  main                  0x00000080
  set_i                 0x000001EA
  add                   0x000000B0
  sub                   0x000000D8
  pow                   0x00000150
  div                   0x00000128

Variables:
Name:                   Location:
  i                     0x000002F0
```

[1] one may wants to make his/her own io.o when embedding which does not use stdin and stdout directly.

[2] private/local functions and variables are not shown.

Here the addresses of all public functions in the original func_serv.c are printed, as well as the single public variable declared in `func_serv.h`.

B.2.2 Preparing the call

Before making a function call, the stack must be prepared for it. All parameters must be placed on the stack and the stack pointers must be set correctly. This is all be done by the `setup_call` function, which is exported by the interpreter engine. Besides the interpreter to run the call, the function takes three parameters: an array of `Value*`, the number of items stored in the array and a `Value*` which denotes the type of the return value. This last parameter is only required to allocate the right amount of space for the return value. The function returns `ERR_NOERROR` when it succeeds or an error code if otherwise.

The `Value*` is a pointer to a structure which can hold any type of variable supported by the interpreter. Several functions to create values are supplied by the interpreter engine, and are all called `create_type`, where type is a C type, such as `create_int`, `create_ulong` etcetera. They all take one parameter being an `int` for `create_int`, and an an `unsigned long` for `create_ulong`. To destroy a value structure, use the `destroy_value` function.

Passing pointers to the interpreter is possible (as long as the pointer points to something inside the interpreters memory, not in the systems memory), but it will most likely be much easier to build a wrapper function to handle the pointers.

B.2.3 Making the call

The function call itself consists of a simple jump to the address got from the linker, which is done with the `jump` procedure. This function takes only one argument: the address of the function to jump to. The interpreter will not start executing itself, but you must write your own execution loop. When the function is completed it will return to address 0, and thus restart the complete program. Since no (correct) function will ever jump to address 0, it is safe to check the program counter for being larger than zero. A complete function call to the `mult` procedure is given here:

```
/* int mult(int a, int b) {
 *     return a * b;
 * }
 */
Value* params[2],c;
params[0] = create_int(4);
params[1] = create_int(8);
c = create_int(0);
setup_call(interpreter, params, 2, c);
jump(interpreter, 0x014E);
while(interpreter->pc && execute(interpreter) == ERR_NOERROR);
pop_value(interpreter, c);
```

The final call, `pop_value`, takes one variable from the top of the stack (and removes it from the stack). After a function call, the return value is placed on the top of the stack, and can thus be read using the `pop_value` function. The

value structure for the return value must be created in advance, to hold the return variable, but its value will be overwritten, and can be set to anything.

When calling multiple functions on the same interpreter, one must make sure to reset the interpreter's state from time to time. The `setup_call` function allocates new stack space for the parameters each time it is called, and the stack will thus run out of space after several calls. Resetting the interpreter is done by the `reset` function:

```
reset(interpreter);
```

Appendix C

Source files

This appendix contains lists of all source files used by the converter, assembler, linker and interpreter, and a short description of its function. Each C source file has a header file which contains further information on the C file, and a brief description of its functions.

C.1 Common source files

These sources are shared through multiple programs.

- `hashtable.c`: Contains functions to create and manage a hashtable.
- `list.c`: Contains functions to create and manage a linked list.
- `memstream.c`: Contains functions to create and manage a memory stream.
- `textparser.c`: Contains functions to parse a text file.
- `instructions.c`: Contains functions to parse, decode and build instructions. All opcodes are defined in the header file.

C.2 Converter

- `converter.c`: Main converter source file

C.3 Assembler

- `assembler.c`: Main assembler source file

C.4 Linker

- `linker.c`: Main linker source file
- `library.c`: Contains functions to load library files.

C.5 Interpreter

- `interpreter.c`: Main interpreter source file
- `interpreter_engine.c`: The actual interpreter (contains execution code)
- `io.c`: Contains several IO functions for Windows and Linux

C.6 Bootstrap code

- `bootstrap.bc`: Bootstrap code for standard programs (calls main)

Appendix D

File formats

This chapter contains the format specifications of all intermediate files generated.

D.1 Bytecode file format

Bytecode files (both the original `LCC` generated files, as the modified bytecode files consist of plain textfiles. Each line contains one instruction or assembler directives, the supported instructions and directives, as well as there format can be found in chapter 3 and appendix A.

Any additional spaces, tabs and whitelines are discarded by the assembler, however they are never generated by `LCC`. Comment can be added using the "`#`" character. All text after a "`#`" is ignored, until the end of the line. Note that comment is not part of the `LCC Bytecode` specifictations, but is accepted by the assembler.

D.2 Assembly file format

The assembly files are binary representations of bytecode files. An assembly file can be disassembled into a bytecode file without losing any information (this excludes comment and formatting). Each assembly file contains a table of labels found in the bytecode file, binary data for each segments, and bitmaps for each segment which point out the location of label positions within the segments. Each bitmap bit represents one segment byte, and each one in the bitmap denotes the representing byte in the segment contains a label location. Before resolving all addresses, the segments contain unique label id's instead of addresses. The address resolver can thus check the bitmap for ones, read the label id from the segment on the same byte-location as the bitmap ones bit-location, find the labels address and write the address back to the segment. The formats of the assembly file, the label table and the individual labels are shown respectively in tables D.1, D.2 and D.3. In table D.4 the different label flags codes can be found. Since several fields in a file have sizes which depend on the data, the size of a field, and thus the offset of the next field, is not known. Therefore, in all tables, the sizes of each field are placed rather than the locations where they can be found.

Table D.1: Assembly file format

`4 bytes`	Magic marker	"SOBJ" or 0x534F424A
`n bytes`	List of labels	
`n bytes`	Code segment	
`n bytes`	Lit segment	
`n bytes`	Data segment	
`n bytes`	BSS segment	

Table D.2: Label list file format

`4 bytes`	Number of labels	Little Endian (Intel) Integer
`n bytes`	List of labels	
`n bytes`	Label 1	
	...	
`n bytes`	Label n	

Table D.3: Label format

`1 byte`	Size of label name	
`n bytes`	Label name	
`1 byte`	Label flags	
`1 byte`	Label segment	
`4 bytes`	Label (unique) id	Little Endian (Intel) Integer
`4 bytes`	Label location	Little Endian (Intel) Integer

Table D.4: Label flags

0x01	label is exported
0x02	label is imported

Any function or variable which is accesible in other files, will be marked as exported. Any function or variable which is accessed, but not declared in the current file will be marked imported. For every imported variable or function, an exported variable or function must exist in any file also passed to the linker.

Table D.5: Segment codes

unkown	0x00
code	0x02
lit	0x04
data	0x08
bss	0x01

Table D.6: Segment format

`4 bytes`	Segment size	Little Endian (Intel) Integer
`n bytes`	Segment data	
`n bytes`	Label bitmap	Will be eight times smaller than the segment data

Finally, in tables D.5 and D.6 the segment data format is shown. Since the BSS segment does not contain any usefull information (uninitialized variables), only the size for this segment is stored in the assembly file, and the segment data and label bitmap are omitted.

D.3 Code segment format

The code segment is filled with instructions. Each instruction is coded into a binary form and then written to the binary assembly file. Each instruction has a 16-bit opcode, containing the actual opcode, the type (pointer, integer, etc...) and size (in bytes), just as they are generated by `LCC`. Some instructions also get a parameter. The encoding of instructions can be found in the tables D.7, D.8 and D.9. The size and coding of parameters can also be found in D.9 (the characters in this column are the same as those used in Table A.2).

Table D.7: Instruction format

Bit index	Description
15-09	opcode (see Table D.9)
08-06	instruction operand type (table:instructiontype2)
05-02	log2 of instruction operand size
01-00	not used

The `INDIR` and `ASGN` parameters are only generated by `LCC` when fetching or assigning a structure. In the binary assembly files, this parameter is always generated, and should be set to 0 for all other operand types.

D.4 Library file format

The library file is a simple container format for several assembly files. The assembly files are dumped right after eachother in the library file. Therefore the library file must be read sequencially, since the exact position of the second file is only known when the first file is completely read. The format of a library file is shown in table D.10.

D.5 Executable file format

The executable file generated by the linker contains the combined binary segments. It has no header, nor a real format. The segments are written in the following order: `code`, `lit`, `data`. The `bss` segment is not placed in the executable file, instead the bootstrap code which is placed at address 0 allocates space for the `bss` segment.

Table D.8: Instruction type format

Type	Coding
Floating point (F)	0x03
Signed integer (I)	0x02
Unsigned integer (U)	0x04
Pointer (P)	0x01
Void (V)	0x05
Structure (B)	0x06

Table D.9: LCC Instructions

Operator	Type Suffixes	Parameter size	Coding
ADDRF	...P..	p	0x01
ADDRG	...P..	p	0x02
ADDRL	...P..	p	0x03
CNST	FIUP..	fdx csilh csilh p	0x04
BCOM	.IU...		0x05
CVF	FI....	i i	0x06
CVI	FIU...	i i i	0x07
CVP	..U...	i	0x08
CVU	.IUP..	i i i	0x09
INDIR	FIUP.B	i i i i i	0x0A
NEG	FI....		0x0B
ADD	FIUP..		0x0C
BAND	.IU...		0x0D
BOR	.IU...		0x0E
BXOR	.IU...		0x0F
DIV	FIU...		0x10
LSH	.IU...		0x11
MOD	.IU...		0x12
MUL	FIU...		0x13
RSH	.IU...		0x14
SUB	FIU...		0x15
ASGN	FIUP.B	i i i i i	0x16
EQ	FIU...	p i i	0x17
GE	FIU...	p i i	0x18
GT	FIU...	p i i	0x19
LE	FIU...	p i i	0x1A
LT	FIU...	p i i	0x1B
NE	FIU...	p i i	0x1C
ARG	FIUP.B	i i i i i	0x1D
CALL	FIUPVB		0x1E
RET	FIUPVB		0x1F
JUMP	....V.		0x20
HALT	....V.		0x7F
ARGSTACK	....V.	i	0x22
VARSTACK	....V.	i	0x23
SYSCALL	FIUP..		0x27
SAVESTATE	....V.		0x28
NEWSTACK	....V.		0x29
DISCARD	FIUP..		0x30

Table D.10: Library format

`4 bytes`	Magic marker	"SLIB" or 0x534C4942
`4 bytes`	Number of assembly files	Little Endian (Intel) Integer
`n bytes`	Assembly file 1	
	...	
`n bytes`	Assembly file n	

Appendix E

User manual

This chapter contains very brief user interfaces on how to run a C project on the interpreter.

E.1 LCC

The `LCC` manual can be found on http://www.cs.princeton.edu/software/lcc/. A standard installation of `LCC` can be used to create bytecode files, but it is recomended to install the new bytecode backend which is discussed in section 4.2. Generating bytecode can either be done by the `RCC` compiler part of `LCC`, or by `LCC` itself. The difference is that `LCC` first calls the C preprocessor, then *RCC*, and then the systems assembler and linker, which can be skipped by using the '-S' argument. To select the bytecode backend, the parameter '-target=bytecode' must be given to *RCC*, or '-Wf-target=bytecode' to `LCC`. 'bytecode' may need to be replaced by 'xbytecode' when the new backend is used.

Note that when using `LCC`, either the `LCC` C preprocessor, or the GNU C preprocessor can be used. Both need to be pointed to the right header file directory (not the systems header files) found in `libs`. The `LCC` driver may need to be modified for this. Information about modifieng the `LCC` driver can be found on the `LCC` website.

E.2 Converter

The converter converts `LCC Bytecode` files into slightly customized `LCC Bytecode` files. The assembler will only accept customized `LCC Bytecode` files, so the converter must be used on `LCC Bytecode` files before passing them to the assembler. The converter must be provided an input and output file:

```
converter <inputfile> -o <outputfile>
```

E.3 Assembler

The assembler converts modified bytecode files into binary assembly files. The binary assembly files can later be fed to the linker, which combines several

binary assembly files into one executable file. The assembler must be provided with an input and output file:

```
assembler <inputfile> -o <outputfile>
```

E.4 Linker

The linker combines several binary assembly files, the bootstrap code, and several library functions if needed, and places them into a single executable file. In addition, the linker can also be used to generate library files. All input files specified must be binary assembly files, generated by the assembler, the output file is either a library file, or an executable file:

```
linker <inputfile> [<inputfile>...] [-lib <libdir>]
    [-buildlib] [-v] -o <outputfile>
```

If the `-buildlib` parameter places all input files in a library file denoted by `outputfile`. The `-lib` and `-v` arguments are ignored. When an executable file is created, the linker will search in each directory specified by the `-lib` argument (multiple directories require multiple `-lib` arguments) for library functions and includes them into the output file when required. When the `-v` argument is used, the linker will print all addresses of public functions and variables to the console, which can be used when embedding the interpreter.

E.5 Interpreter

The interpreter executes executable files generated by the linker. When it is used as a standalone application it can be executed from the commandline:

```
interpreter <inputfile> [m <memorysize>] [-nd]
    [-t <tracefile>] [-a <analysisfile>]
```

Besides an input file, which must be specified, two output files can be specified, an analysis file and a trace file. When an analysis file is specified, it will be filled with information on the amount of instructions executed, the amount of memory used, and how many each individual instruction type was executed. The result is stored in a tab delimited text file. When a trace file is specified, on each clocktick the current instruction, machine state and stack is written to the trace file. Note that this file will become very large on big programs. Finally, the nd parameter can be used to ignore the `DISCARD` instruction and should only be used for testing purpses.

Appendix F

Example compilation run

This chapter shows an entire sample run from C source file to executable file and explains the data in all intermediate files.

F.1 C File

The following C file is used to generate an executable file, which also served as an example in the previous chapters.

```
int add(int a, int b) {
    return a + b;
}
int main() {
    int sum;
    sum = add(1, 2);
    return sum;
}
```

F.2 Bytecode file

The bytecode file is generated by `LCC` during the compilation phase. The bytecode file is slightly different from the original example, since `LCC` copies the `sum` variable to another local variable before returning it. This was omitted from the original example for simplicity (the result is the same).

```
export add
code
proc add 0 0
ADDRFP4 0
INDIRI4
ADDRFP4 4
INDIRI4
ADDI4
RETI4
LABELV $1
```

```
endproc add 0 0
export main
proc main 8 8
CNSTI4 1
ARGI4
CNSTI4 2
ARGI4
ADDRLP4 4
ADDRGP4 add
CALLI4
ASGNI4
ADDRLP4 0
ADDRLP4 4
INDIRI4
ASGNI4
ADDRLP4 0
INDIRI4
RETI4
LABELV $2
endproc main 8 8
```

F.3 Custimized bytecode file

The customized bytecode file is a slightly altered version of the bytecode file, for easier assembling. All changes made can be found in Section 4.3.

```
export add
code
### procedure add:
label add
SAVESTATEP4
ARGSTACKV 0
VARSTACKV 0
ADDRFP4 0
INDIRI4
ADDRFP4 4
INDIRI4
ADDI4
RETI4
label $1
RETV
### end procedure add
export main
### procedure main:
label main
SAVESTATEP4
ARGSTACKV 8
VARSTACKV 8
CNSTI4 1
ARGI4 0
```

```
CNSTI4 2
ARGI4 4
ADDRLP4 4
ADDRGP4 add
CALLI4
ASGNI4
ADDRLP4 0
ADDRLP4 4
INDIRI4
ASGNI4
ADDRLP4 0
INDIRI4
RETI4
label $2
RETV
### end procedure main
```

F.4 Binary assembly file

The binary assembly file contains the binary representation of the customized bytecode file. It is printed completely in hexadecimal, since most of the data is binary. Using the '#' character, comment is added to the file to explain all data.

```
# file id: SOBJ
0x0000: 53 4F 42 4A
# total of 4 labels
0x0004: 04 00 00 00
# label main
#   main is 4 characters long
0x0008: 04
#   "main"
0x0009: 6D 61 69 6E
#   the flag of main is 0x01 (exported)
0x000D: 01
#   the segment of main is 0x02 (code)
0x000E: 02
#   the unique id of this label is 0x03
0x000F: 03 00 00 00
#   the address of main in the code segment is 0x2C
0x0013: 2C 00 00 00
# label $1
0x0017: 02 24 31 00 02 02 00 00 00 2A 00 00 00
# label $2
0x0024: 02 24 32 00 02 04 00 00 00 8C 00 00 00
# label add
0x0031: 03 61 64 64 01 02 01 00 00 00 00 00 00 00
# 90h bytes code segment size
0x003F: 90 00 00 00
# label add
```

```
# SAVESTATEP4
0x0043: 50 48
# ARGSTACKV 0
0x0045: 47 40 00 00 00 00
# VARSTACKV 0
0x004B: 45 40 00 00 00 00
# ADDRFP4 0
0x0051: 02 48 00 00 00 00
# INDIRI4
0x0057: 14 88 00 00 00 00
# ADDRFP4 4
0x005D: 02 48 00 00 00 04
# INDIRI4
0x0063: 14 88 00 00 00 00
# ADDI4
0x0069: 18 88
# RETI4
0x006B: 3E 88
# label $1
# RETV
0x006D: 3F 40
# label main
# SAVESTATEP4
0x006F: 50 48
# ARGSTACKV 8
0x0071: 47 40 00 00 00 08
# VARSTACKV 8
0x0077: 45 50 00 00 00 08
# CNSTI4 1
0x007D: 08 88 00 00 00 01
# ARGI4 0
0x0083: 3A 88 00 00 00 00
# CNSTI4 2
0x0089: 08 88 00 00 00 02
# ARGI4 4
0x008F: 3A 88 00 00 00 04
# ADDRLP4 4
0x0095: 06 48 00 00 00 04
# ADDRGP4 add
#   the parameter is 0x01, the unique id of label add
0x009B: 04 48 00 00 00 01
# CALLI4
0x00A1: 3C 88
# ASGNI4
0x00A3: 2C 88 00 00 00 00
# ADDRLP4 0
0x00A9: 06 48 00 00 00 00
# ADDRLP4 4
0x00AF: 06 48 00 00 00 04
# INDIRI4
```

```
0x00B5: 14 88 00 00 00 00
# ASGNI4
0x00BB: 2C 88 00 00 00 00
# ADDRLP4 0
0x00C1: 06 48 00 00 00 00
# INDIRI4
0x00C7: 14 88 00 00 00 00
# RETI4
0x00CD: 3E 88
# label $2
# RETV
0x00CF: 3F 40
# alignment bytes, to make segment size a multiple of 8
0x00D1: 00 00
# label bitmap for code segment (0x90/8 = 0x12 bytes)
#   the 3C on 0x00DE (4 ones starting at bit 90)
#   represent a label reference at byte 90 in the code
#   segment (which starts at 0x43). A label reference
#   is placed at 0x9D; the parameter of 'ADDRGP4 add'
0x00D3: 00 00 00 00 00 00 00 00
0x00DB: 00 00 00 3C 00 00 00 00
0x00E3: 00 00
# size of lit segment (0 bytes)
0x00E5: 00 00 00 00
# the size of the lit segment is 0, thus there is no
#    lit segment or bitmap data present in the file
# size of data segment (0 bytes)
0x00E9: 00 00 00 00
# size of bss segment (0 bytes)
0x00ED: 00 00 00 00
```

F.5 Executable file

The executable file contains the combined binary segments of the loader and the assembly file shown in the previous section. It starts with the code segment of the loader, followed by the code segment of the assembly file from the previous section. This piece of data is the same as the data from the previous section, except for the parameter of the `ADDRGP4 add` instruction, it is now correctly set to the absolute address of the add function. After the code segments, the lit and bss segments of the loader are located, since the example does not use these segments, they also don't appear in the executable.

Bibliography

[1] 8051 Instruction set: http://www.win.tue.nl/ aeb/comp/8051/set8051.html (October 2005)

[2] Cipher Game Engine Implementation of LCC Bytecode: http://www.rik.org/articles/ciphervm.htm

[3] A.J.C. van Gemund, Personal Communication (September 2004)

[4] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs and Koen G. Langendoen: *Modern Compiler Design*, John Wiley & Sons Ltd. (2003)

[5] David R. Hanson and Christopher W. Fraser: *A Retargetable C Compiler*, Addison-Wesley (1995)

[6] David R. Hanson and Christopher W. Fraser: *The lcc .4x Code-Generation Interface* (2003)

[7] John L. Hennessy and David A. Patterson: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann (2003)

[8] John L. Hennessy and David A. Patterson: *Computer Orginazation & Design: The Hardware/Software Interface*, Morgan Kaufmann (1998)

[9] i386 Instruction set: http://www.logix.cz/michal/doc/i386/chp17-00.htm (October 2005)

[10] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*, Prentice Hall P T R (2004)

[11] Jens Kretzschmar: *Extending the porting of the GCC-Backend for TMS320C6x VLIW-Architecture* (2004)

[12] Hans-Peter Nilsson: *Porting The Gnu C Compiler for Dummies* (2004)

[13] Quake 3 Virtual Machine Implementation of LCC Bytecode: http://www.icculus.org/~phaethon/q3mc/q3vm_specs.html (October 2005)

[14] SDCC Programmers: *SDCC Compiler User Guide* (2005)