



**LUNDS TEKNISKA  
HÖGSKOLA**  
Lunds universitet

# Retargeting GCC for a DSP architecture

**Master Thesis**

by

**Jonas Paulsson**

**Submitted in March 2010**

**ST-Ericsson, Access Core Software**

**Department of Computer Science at LTH, Lund University**

---

**Supervisor:**  
Markus Lavin  
ST-Ericsson  
Lund, Sweden

**Examinator:**  
Dr. Jonas Skeppstedt  
Dept. of Computer Science  
LTH

## Abstract

This thesis examines porting GCC to the Flexible ASIC DSP (FlexDSP) which is developed and used by Ericsson. A new back end has been added to GCC by writing a GCC machine description for the target, and compiler passes have been added to the compiler for improving the DSP code.

The modern DSP has a set of characteristic features that a compiler should make use of. As GCC is not mainly aimed at DSP architectures this degree project has been done with the goal to investigate how well GCC can handle such targets. These features include VLIW scheduling, SIMD instructions, hardware loop instructions, and more.

The project has proved successful in that all main features of the DSP have been implemented, some with direct support from GCC, others with customized compilation passes. The resulting gcc compiler has been tested with a simulator<sup>[1]</sup> for the DSP, and for a subset of the existing benchmark suite the results have been found comparable to those of the existing compiler. This gives praise to GCC, as there is still opportunity for further improvements at the end of the project.

## **Acknowledgments**

This thesis work was carried out in the Access Core Software department at the ST-Ericsson facility in Lund.

I would like to thank Markus Lavin, my supervisor at ST-Ericsson, for guidance, skill and knowledge during the course of the project.

I would also like to thank all of the faculty of the department of Computer Science at LTH, including Lennart Andersson and Jonas Skeppstedt. The LTH courses have made this incredible project possible for me to undertake and succeed with.

# Contents

1	Introduction.....	6
	GCC .....	6
	FlexDSP, part of Flex ASIC concept.....	6
	Outline.....	6
2	The modern DSP.....	8
	Typical features and instruction set.....	8
	VLIW.....	8
	SIMD instructions.....	8
	Conditional execution of instructions.....	8
	HW-loops.....	8
	Modulo-addressing .....	8
	Split register file.....	9
	Unprotected pipeline.....	9
	Branch delay slots.....	9
	Wide data bus.....	9
	Mode based model for signed/unsigned data types.....	9
	Typical DSP code and its optimal implementation.....	9
	The FlexASIC DSP.....	10
3	The GCC compiler.....	12
	General.....	12
	Machine independent part.....	12
	Machine dependent part.....	12
	Machine description file.....	13
	Machine description header file.....	15
	Machine description C file.....	15
	DSP support.....	15
	MD-RTL.....	15
	Macros and target hooks.....	15
	GCC passes.....	16
	Adding a pass.....	16
4	Porting GCC to FlexASIC.....	17
	Basics.....	17
	Registers.....	17
	Predicates and constraints.....	18
	DSP ISA.....	19
	Miscellaneous.....	21
	VLIW scheduling .....	23
	Hardware looping instructions.....	25
	Unrolling loops with explicit unroll factor .....	26
	Accumulator variable expansion.....	28
	Tuning inner loops with ivopts.....	29
	Mac, mas .....	29
	Auto-incremented addressing.....	30
	Widening multiplication.....	30
	GIMPLE pass.....	32
	mulhi expansion.....	32
	define_insn_and_split.....	33

Predicated instructions.....	33
SIMD memory accesses.....	35
Mode switching.....	36
The reorg pass.....	37
5 Results.....	38
The dot product example.....	38
Compilation of benchmarks.....	43
6 Conclusions.....	45
GCC as a DSP compiler.....	45
Todo.....	45
Register reloading.....	45
Modulo scheduling.....	46
Nested hardware loops.....	46
Rescheduling of compare instructions.....	46
Instructions coverage, code acceptance.....	46
Passing of arguments on the stack.....	46
Modulo addressing.....	46
Unroll / variables expansion passes.....	46
Flag registers and configuration registers.....	47
References.....	48

# 1 Introduction

## GCC

The Gnu Compiler Collection is an open source compiler which is available for download on the Internet for free, by anyone. It is the result of an ongoing global project and the current stable version is 4.4.

With the source code, a new modified compiler can be built. Various things can be done for different purposes, as the source code is available and furthermore includes tools to facilitate such extensions. One could add a new source language, a new back end for a specific target, add an optimization pass, or make other modifications, depending on the task at hand.

A machine description for GCC is written mainly in the MD-RTL<sup>[2]</sup> language, which special generator programs use to produce C files for the new compiler. There are many different MD-RTL constructs to use for this purpose, as well as a vast number of macros and target hooks (function calls) that can be defined, so as to connect and insert the specifics of a target CPU into the GCC machinery at predefined places. In this way, many different aspects of the compilers work are set and tuned in a manageable way.

As GCC is free and open source, it is becoming an interesting option compared to traditional investments. It is a question of how well the resulting compiler behaves, and how much work is demanded to accomplish the desired results.

GCC with capital letters refers to the general compiler system, whereas gcc stands for the target specific compiler ready to be used.

## FlexDSP, part of Flex ASIC concept

The Flexible ASIC DSP (FlexDSP) is a digital signal processor (DSP) which is used in current Ericsson and ST-Ericsson designs. It is occupied between the antenna and the central system of the design and its job is to swiftly do computational intensive work on the digital signal, such as forward error correction code encoding/decoding, and thus save other resources from this work.

Certain typical DSP-algorithms can be considered the primary domain for FlexASIC. Such an algorithm is loop based (often with nested loops), and typically reads from memory and perform some kind of computation which is stored back or accumulated to a register.

There is an existing compiler which is used with the C language, as well as a cycle per cycle accurate simulator.

Due to the proprietary nature of FlexASIC, the report will not focus on the details of its ISA. However, it can be regarded as a general DSP and this thesis will treat it as such.

## Outline

This report will highlight the common characteristics of a general DSP as a background in chapter two. This provides a reference for the remainder of the chapters, as these DSP features are the basic motivation in adapting the compiler.

In chapter three, GCC will be outlined in general terms, and different strategies for supporting a DSP are listed.

Then, in chapter four, the implementations done on the compiler are accounted for. These are basically correlating to the basic DSP features found in chapter two.

Chapter five gives the results of the implementation, by incrementally adding support to the compiler for a simple DSP code example, and showing its benefits in terms of reduced CCL's (clock cycles), along with the produced assembler code for the inner loop body. This chapter also gives a comparison to the existing compiler on a subset of the benchmark suite currently in use.

Finally, chapter six gives conclusions and a summary of things unfortunately not incorporated into the compiler but which are obviously called for, except for the time limit.

## 2 The modern DSP

DSPs are a family of processors with a design that is different from ordinary CPU's, as they are used in specialized applications. They are smaller and optimized for a limited set of tasks.

A DSP with custom software is one approach for an embedded solution, which falls between a general purpose CPU and dedicated hardware. Most of the algorithms that the DSP is designed to handle has one element in common: the MAC (multiply-accumulate) operation<sup>[3]</sup>. This instruction can perform a multiply and addition on the same cycle.

As the features of a DSP are the main challenge of this degree project they are listed below with brief descriptions. In later chapters, they will be handled one by one and finally the resulting gains are displayed.

### Typical features and instruction set

#### VLIW

VLIW means Very Long Instruction Word, and signifies one type of multiple-issue processors. In every cycle several instructions are handled in a larger instruction packet. For instance, a DSP could be designed to perform two MAC's in parallel every clock cycle. This type of architecture depends on the compiler for hazard detection and scheduling<sup>[3]</sup>, as opposed to super scalar processors.

#### SIMD instructions

Single Instruction, Multiple Data, or vector instructions. In this context, this refers to partitioning of operations, so that instead of for example a single 64 bit add operation which is unnecessarily big, four additions might be executed in parallel where each operation deals with one fourth of the operand space, 16 bits<sup>[3]</sup> (of course, only if 16 bits corresponds to the used data type).

#### Conditional execution of instructions.

To gain speed, so called predicated instructions can be used, meaning that they will only be executed if a condition is met. A predicated instruction can remove the necessity of making conditional jumps.

#### HW-loops

DSPs can gain in speed by handling smaller loops with constant iterations with special repeat instructions. This means that extra hardware supports looping by providing dedicated registers for start and stop address of the loop, which keeps unnecessary instructions out of the pipeline that slow down each iteration.

#### Modulo-addressing

means that loops are improved by treating addresses in an incremental manner with a wrap-around at a set boundary. Repeated accesses to a block of memory can thus be done from compact code.

### Split register file

In order to achieve high throughput, there are many registers, which are used differently in the instruction set. Some instructions take only data registers as operands, others demand address registers, or a special address increment register, etc. This makes the ISA harder to work with since many restrictions are put on the allowed assembly language. The reason for splitting the register file is that it is necessary in order to save hardware area.

### Unprotected pipeline

Large processors relies on dynamic hazard detection while issuing multiple instructions, but the DSP usually relies on the compiler to avoid hazards (static hazard detection).

### Branch delay slots

The instructions immediately after a branch instruction is typically executed no matter the branch outcome. This is because a branch causes a pipeline stall, and optimizing compilers shall either place useful instructions or nops to be executed here, depending on the architecture <sup>[4]</sup>. A DSP can have a quite long exposed pipeline with a branch delay slot of four to six instructions.

### Wide data bus

The data bus of a DSP facilitates powerful data fetching/storing to keep the computational units busy. In order to use parallel memory based operations in a loop body, there are means of accessing independent memory addresses on the same cycle.

### Mode based model for signed/unsigned data types

Mode flags are used to steer the execution results of instructions in a region-wise manner, as opposed to using separate instructions. This gives a smaller ISA and thus space-savings in the VLIW instruction packets.

### Typical DSP code and its optimal implementation

To get a clear picture of what the above means, here is an example showing the differences between an ordinary general purpose CPU and the DSP in the way they perform the task

```
sum=0;
for (i=0;i<256;i++)
    sum += a[i]*b[i]
```

<i>using standard instructions</i>	<i>using DSP instructions</i>
set index_reg = 256 sum_reg = 0 r0=a	set loop_count = 256/2 sum = 0 sum2 = 0

<i>using standard instructions</i>	<i>using DSP instructions</i>
<pre> r1=b LOOP: index_reg-- lw *r0, a0l lw *r1, a1l mul a0l, a1l,a2l add a2l, sum_reg add 1, r0 add 1, r1 blez index_reg, LOOP </pre>	<pre> r0 = a r1 = b lw *r0++, a0   lw *r1++,a1 block_rep begin lw *r0++, a0   lw *r1++,a1   mac a0l,a1l,sum   mac a0h,a1h,sum2 block_rep end mac a0l,a1l,sum   mac a0h,a1h,sum2 add sum2, sum </pre>

With the DSP features in use to the right, the same task is performed at a much higher rate using smaller code. Note the hardware loop instruction, the parallel mac instructions, the powerful memory accesses with two 32 bit words loaded in parallel with the macs (wide data bus, SIMD and VLIW features) and the auto increment operators, all which together make up a very powerful solution. The loop has been software pipelined, which means that there is the prologue and epilogue of the loop in order to load for the next cycle in parallel with the macs.

The DSP *could* have run the code to the left, but performance would be insufficient. The advantage of the DSP code is very clear with two macs per cycle using but half the number of iterations. Making GCC produce the output to the right instead of that to the left is the basic objective of this degree project.

## The FlexASIC DSP

The FlexASIC is a typically advanced DSP processor which exemplifies most of the points discussed above. The PCU (Program Control Unit) controls the execution and dispatches instructions to four computational units of varying capacity or to the DAAU, the Data Address Arithmetic Unit, in case of a memory access. All common instructions have a one cycle latency. A difference of FlexASIC compared to a typical DSP is the absence of a visible pipeline or any branch delay slots.

It uses 64 bit instruction words, which each can be used by up to four individual instructions. Instructions demand either 16 or 32 bits, where the common instructions are usually of the smaller kind, while for example using a less frequent register is likely to call for a 32 bit instruction. Loading a big constant to a register is also more space consuming than a small immediate that fits within the 16 bit issue slot. The VLIW packaging is thus limited by different instructions' widths, as well as by available resources and other scheduling constraints. It is the responsibility of the programmer or compiler to provide proper VLIW scheduling.

Most instructions can be predicable, meaning they can be conditionally executed depending on a condition attached to the instruction. Such a condition uses additionally 16 bits of the VLIW word. It is also possible to use a global condition for the whole VLIW word, and even combinations of a global condition and a local condition.

It can load up to two 32 bit words per cycle with only a one cycle latency. This data can be interpreted as either 32 bits, or as two 16 bit operands, meaning that a SIMD memory access is possible with a total maximum of four 16 bit words per cycle. Addressing is done with 16 bit registers.

It supports nested hardware loop instructions in three levels, as well as a special instruction for single instruction (64 bits) repetition. The block of code in such a loop must not, among other things, contain more than 256 VLIW packets.

It has a complex register file, with address registers, data registers, and many special registers such as hardware loop counters. Many registers can be used as 32 bits or in 16 bit parts. Each data register has a flag register which can be used in conjunction with predicated instructions, such as a branch instruction. There are also configuration registers for each such data register, setting different modes for sign/zero extension when loading 16 bits.

Multiply instructions are performed while interpreting the operands based on the setting of the Computational Unit Configuration (cuc) register. By using the cuc register for mode switching, the same instructions will perform sign or unsigned multiplication as called for. The mac and mas instructions are also subject to the cuc mode.

The DSP also goes by the alias "Cream XL", and many identifiers in the source code in following chapters are prefixed with `creamXL`.

## 3 The GCC compiler

### General

GCC is written in C and includes generator programs to facilitate writing back ends using the MD-RTL language. Many common targets (including x86, MIPS and ARM for example) have such .md files providing support for their different machine characteristics.

There are many passes (about 200) that transform the intermediate representation in some way. Each pass produces a debug dump-file which will display the essentials of the pass and at the end the IR dump after the pass is done.

Loop analysis, data flow analysis, basic block information and much more is available through C macros and functions. Standard passes use these tools, but they are also available while adding a customized pass.

Throughout the code there are macros scattered, which are defined differently among the machine descriptions. These can represent values, or functions (hooks), and embodies much customization of the compiler.

The current stable version is 4.4, and it is a continuously developing software project in a global community around the world. It is worth mentioning that this thesis will describe features for the current version, while there are more things under development, including DSP supporting features.

### Machine independent part

A central idea of compilers is separation of source language (front end) and target CPU (back end) from each other and the rest of the compiler in order to achieve efficiency in implementing compilers for different languages and different CPU's. In GCC, the source code is first parsed by a front-end which produces a GENERIC IR, which is quickly translated to the GIMPLE form.

GIMPLE is the first primary IR stage where many common transformations are done on the SSA form. At this stage, the code is treated without concern for target specifics, and is therefore independent of both source language and the target machine. This form is the more general and powerful one and is sometimes a viable option even with target specific requirements in mind.

Before the code is handed over for machine specific transformations at the end of the GIMPLE stage, a great number of SSA optimization passes (60-70) have been run<sup>[5]</sup>.

### Machine dependent part

After SSA, the code is optimized and now needs to be expanded to a machine specific state. Each GIMPLE statement is replaced by one or several machine expressions of IR-RTL<sup>[2]</sup>, which is a second IR language that is machine dependent. For historic reasons, a GIMPLE statement is first converted to a tree, which is then expanded to a RTL sequence. This process is referred to as RTL expansion and after it the semantics of the compiled program are captured directly by the RTL insns<sup>[6]</sup> (instructions) and their patterns.

Many things are done then with the RTL IR as well. For example, the combiner pass tries to find better instructions by for example combining an add and a mpy to a single mac insn. The loop passes include the doloop pass which tries to use a hardware loop pattern if available. The auto-inc-dec pass tries to use pre/post-increment/decrement/modify insns if the machine supports this.

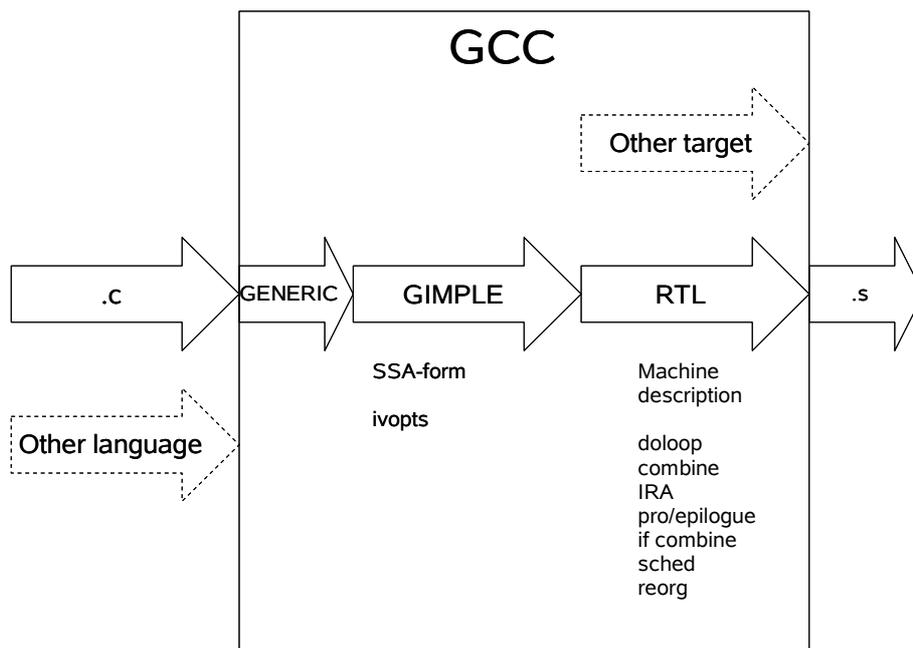


Figure 1: The basic architecture of GCC

The scheduler is run near the end of compilation twice, once before IRA and once after reload. Between these passes, the pro- and epilogue pass is also run. The IRA pass uses a coloring algorithm, and when it is done it also reloads registers to make insns valid according to register constraints. This could mean inserting a move instruction of a register if an instruction must use another register than that which was allocated by IRA.

Customary optimization passes are used on RTL as well, such as dead code elimination and copy propagation, to handle the insn list as it undergoes change.

At the very end, a machine reorganization pass takes place. This pass is the place where target specific fixes can be performed after everything else. For instance, nops could be inserted here, after the final scheduling, if there are special cases of pipeline conflicts the scheduler is unaware of.

With -O3, there are 40-50 RTL passes that improve and finalize the machine code. After this, the RTL list is printed out as assembler code.

### Machine description file

Instructions of the target are input into a .md (machine description) file<sup>[6]</sup>, which is handled by GCC generator programs to output C code for use during build of the compiler. An instruction pattern in this file contains both the semantics and the syntax of the ISA instruction. The semantics are captured with RTL operands and operators which form RTL expressions, which has a Lisp like notation:

```

(define_insn "tstsi"
  [(set (cc0)
        (match_operand:SI 0 "general_operand" "rm"))]

```

```

" "
" *
{
if (TARGET_68020 || ! ADDRESS_REG_P (operands[0]))
    return \"tstl %0\";
return \"cml #0,%0\";
} ) [6]

```

Here the name of the insn (RTL instruction) is “tstsi”. A `gen_tstsi()` function will be generated which is used when such an insn needs to be created. The semantics are captured in the RTL template after the name. In this case the `set` operator is used to assign a value to the `cc0` register. In the middle, there is an empty string which could have been filled with a condition which expresses if the pattern is available or not to the compiler, perhaps depending on a command-line option. At the bottom is the assembler output clause, checking a condition before returning alternate output template strings.

Present in the `insn-pattern` above is the predicate `general_operand` and the constraint `rm` inside a `match_operand` expression. In this way, the insn is allowed only a limited set of operands. Predicate are used in insn matching, while the constraint is a refinement of the selection. For instance, the predicate might dictate a register operand, and the accompanying constraint could limit this to a certain register or register class. There are standard predicates and constraints, but it is also possible to define new ones. This can be done with either MD-RTL logical expressions, or by a C function.

The `.md` file also contains constant definitions, insn attributes definitions and assignments and `define_expand` expressions, which provide a way to use C to make a more complex RTL expansion, depending on circumstances and usually resulting in several insns being emitted, as opposed to using a `define_insn` expression. Furthermore, a pipeline description can be defined as a means of adjusting the scheduler properly in regards to CPU resources. This description will generate a deterministic finite automaton which reflects the state of the CPU as insns are issued.

At the point where the GIMPLE code is converted to RTL, Standard Named Patterns are used to utilize standard expressions the compiler is aware of. The SPNs that are actually used at compile time are the ones defined in the `.md` file, a subset of all possible such patterns. These are defined by providing the right name and RTL template.

The RTL expansion is split by the middle with SPNs: GIMPLE to a SPN, and then the SPN to RTL. The first part is machine independent, the second is based on the `.md` file. When the compiler is built, a table of available patterns is filled, `struct insn_data[]`. The indexes into this table are stored in `struct optab[]`. These lists make up the two parts of the GIMPLE to RTL translation<sup>[2]</sup>.

A very minimum set of instructions is mandatory, such as basic move instructions. For cases where a voluntary insn is missing, GCC makes the best it can by using the available RTL-patterns while producing (suboptimal) code.

From the `.md` file various complex switching functions are built for efficient use during compilation, for example to get the `INSN_CODE` for a particular insn (which would be `CODE_FOR_tstsi` in the case above).

## Machine description header file

Apart from the RTL patterns, much information is needed in order for the compiler to produce code for a target. A model of the register file is used so that the register allocator knows about the available registers, and so that later the correct register names are output. Various other things that are machine specific - word length, pointer size, addressing modes and more must be defined. Basically, the compiler have suited itself to become general by means of inserting into its source code a wealth of macros which are to be defined by the programmer. For instance, by writing

```
#define FIRST_PSEUDO_REGISTER 32
```

the compiler will know there are 32 registers on the machine. Whenever there is such an abstraction, a macro or target-hook can be defined. The latter would be a C function that does some particular job, for example return true/false if a certain register can be used as a base address register.

## Machine description C file

The defined hooks and functions are placed in a .c file in the same directory as the .md and .h files. This file is compiled and used during the build of the compiler. Inside this file, there is great opportunity to work with the IR, using all the different internal functionality of GCC as needed.

## DSP support

Ideally, the compiler would model all details of the DSP in the .md and .h files, and thus produce excellent code for it without further work. It is however not true that GCC is mainly a DSP compiler, and thus have pieces missing from this reference image.

GCC is big enough to provide alternative strategies for implementing features. These will be described below in order of increasing complexity.

## MD-RTL

In the .md file, most of the ISA is captured, in terms of available instructions and their terms of execution. One can add a basic instruction and it will be used by the compiler. Each instruction has operand predicates and constraints, and thus the different types of registers and the oddities of the DSP ISA is handled.

The .md file covers what's needed for VLIW-packing by means of a pipeline description. The hardware loop instructions are handled in part by adding the doloop-patterns. The mac insn pattern, if present, will make the combine pass use it whenever possible. However, just working with MD-RTL will not suffice for a DSP, but it's a good start.

## Macros and target hooks

The mandatory parts of the .h file let the compilation proceed with the proper machine attributes. There are also a great many optional macros and hooks that can be used to reshape the compiler as desired. There is a hook that is called just prior to output-time, where VLIW-packing is finalized during the output stage, for instance. The hook for machine dependent reorganization can be implemented to do some minor changes to the code at the end. Cost hooks can be implemented giving preferences for what type of code works best, and so on.

This is an important part in supporting the DSP. For instance, the `ivopts` GIMPLE pass will rewrite loop bodies according to address and RTX (RTL Expression) costs defined in two related

hooks. Scheduling can be adjusted as well to for instance allow parallel pushes/pops if this is supported. Mode switching can be accomplished by defining modes for entities (such as a cuc register) and two simple functions.

### **GCC passes**

There are passes as described above that work towards DSP optimization as well. The doloop and auto-inc-dec directly performs DSP related work, if the .md file is supplied with the right insn-patterns. This helps as well, but there are not a complete set of DSP passes provided. Neither are the available passes always providing optimal solutions for a specific DSP target.

### **Adding a pass**

There is opportunity to add passes both during GIMPLE and RTL. There are numerous interfaces (macros and functions) to data structures to work with while implementing an algorithm.

Adding a pass is an important opportunity for supporting the DSP as a last resort as anything can be done to help the code with the power of C.

General transformations are best performed on the SSA form with a GIMPLE pass, while machine specific actions call for a RTL pass.

As an example, the loop unrolling passes of GCC did not work well for the DSP. Therefore, a GIMPLE pass was added to the compiler during the project, as will be described in the following chapter.

## 4 Porting GCC to FlexASIC

Here follows a report from the process of implementing support for the DSP in GCC. First, the mandatory parts are covered, and then sections of the optimizations follows, with many code examples.

### Basics

Since GCC is made to support a wide variety of processors, the work began with specifying various mandatory macros and hooks. These are the basis of the port such as word-width, addressing modes, legitimate addresses, register file, assembler output format and all else that is required to make a first non optimizing compilation.

The .md file was as well filled with `define_insn` and `define_expand` expressions, corresponding to instructions in the DSP ISA document. The irregular register usage was handled by adding new constraints.

### Registers

The available registers were defined and with them a set of register classes. The register classes are used when defining instruction patterns so that operands can be matched with only a limited set of registers. For example, a particular instruction might only use a few certain registers as destination registers, and for these a register class might be defined. Then, based on the register class, a register constraint is defined which can be used in a `match_operand` expression, which means that GCC will limit the operands to registers from just this class. Of course, prior to IRA any pseudo register is allowed as a temporary operand.

.h file:

```
//(40 hard registers)
#define FIRST_PSEUDO_REGISTER 40
#define REGISTER_NAMES {"r0", "r1", "r2", "r3"
                       ... "r39"}
enum reg_class {FIRST8_DATA_REGS,
               ...
               LIM_REG_CLASSES}
//reg_class contents are given by bit vectors for each class
#define REG_CLASS_CONTENTS{ {0x00ff0000,0x0},
                           ... }
```

.md file:

```
(define_register_constraint "r1" "FIRST8_DATA_REGS")
(define_insn "mac"
  [(set (match_operand:HI 0 "register_operand" "=r1")
        (plus:HI
```

```

(mult:HI (match_operand:HI 1 "register_operand" "")
         (match_operand:HI 2 "register_operand" ""))
  match_dup 0))
)]
""
"mac %1, %2, %0"
)

```

Above the mac instruction is added to the compiler. The ISA only allows the mac instruction destination to be any of the first 8 of the data registers, and this is achieved by limiting the possible operands by means of the "r1" constraint.

With

```

#define BITS_PER_UNIT 16
#define UNITS_PER_WORD 1

```

is in the header file, the `match_operand:HI` expression above signifies a Half Integer operand of 32 bits. A Single Integer (SI) is made of 4 units or 64 bits in this case, and QI would mean Quarter Integer and 16 bits. SImode is not used for the DSP.

FlexASIC has many registers that can be used as pairs (32 bits) or by the high/low parts (2\*16 bits). This was solved by giving each such 32 bit register two hard registers in the .h file, and adding code in the hooks so that such a register, when accessed as 32 bits, is only allowed to have an even number. In this way, one can have `reg:HI 16`, `reg:HI 18`, or `reg:QI 16`, `reg:QI 17`, `reg:QI 18`, and so on. The `HARD_REGNO_NREGS(REGN, MODE)` hook was then made to return 2 for HImode, and 1 for QImode, and so the register allocation was handled.

### Predicates and constraints

The DSP ISA has many restrictions which must be adhered to by the compiler. For instance, an instruction might only allow a memory reference with a small offset. A constraint is defined by

```

(define_constraint "MsO" "Memory ref: dp, offset from -16 to 15"
  (and (match_code "mem")
        (match_test "dp_small_offset(XEXP(op,0))"))
  )
)

```

where "MsO" is an arbitrary name for the constraint, the second string is a text description, and after it comes the expression that must be true for the particular instruction to match. The condition here calls the `dp_small_offset()` function which was written in the .c file.

There are many different ranges of immediate values that the DSP distinguishes between. These were defined by constraints similar to the above, for example

```

(define_constraint "A" "Immediate -16 to 15"
  (and (match_code "const_int")
        (match_test "(ival >=-16 && ival <=15)"))
)

```

```
)  
)
```

ival is the value of the immediate operand which is available inside the match\_test expression<sup>[6]</sup>. After the genpreds generator program have worked on the above RTL entry while building the compiler, the following C code becomes part of the build:

```
static inline bool  
satisfies_constraint_A (rtx op)  
{  
    HOST_WIDE_INT ival = 0;  
    if (GET_CODE (op) == CONST_INT)  
        ival = INTVAL (op);  
    return (GET_CODE (op) == CONST_INT) && (  
#line 99 "../trunk/gcc-4.4.1/gcc/config/creamxl/creamxl.md"  
((ival >=-16 && ival <=15)));  
}
```

Predicates were defined as well, for example:

```
(define_predicate "sym_operand"  
  (match_code("symbol_ref"))
```

## DSP ISA

Each instruction of the DSP that the compiler should use, must be expressed in RTL with RTL-operators. Operands are usually defined with abstraction, and then matched during compilation according to predicates. Many different DSP instructions fit into the same RTL-instruction, due to the fact that there are often several versions of the same DSP instruction which varies somewhat, often in the allowed operands. Separate instruction-templates had to be defined for HI and QI mode operations.

For the case of copying a register to another in 16 bits, there are four different cases and instructions:

```
(define_insn "reg_to_reg_qi"  
  [(set (match_operand:QI 0 "register_operand" "=f,Wa,Wb,e")  
        (match_operand:QI 1 "register_operand" "f,f,f,e")  
        )]  
  ""  
  "@  
  mv \\t%1, %0  
  mv \\t%1, %0  
  mv \\t%1, %0
```

```

        mv \\t%1, %0"
[(set_attr "type" "move,move,move,move")
 (set_attr "n_islots" "1,1,1,2")]

```

The four lines in the middle are the output (that here happens to be identical between lines) that correspond to the four different constraint columns in the pattern. As explained above, the constraints have been defined to express correct uses of the DSP instructions. The fourth entry is 32 bits in width (2 issue slots), which is why it comes last so that if possible a cheaper alternative will be found. Alternatives, as well as instructions as found in the .md file, are chosen by the first legal match. For this to be possible, the column of constraints for an alternative must hold on all operands.

The `n_islots` and `type` attributes, which are used with the pipeline description, are set on the last lines. These added attributes were introduced in the md-file, with for example

```

(define_attr "n_islots" "1, 2, 3, 4" ;one issue slot = 16 bits
 (const_string "2"))

```

The default value of two has been provided at the end here. Their further role in scheduling will be explained below.

As another example, addition of 16 bit operands looks like

```

(define_insn "addqi3"
 [(set (match_operand:QI 0 "register_operand" "=ka,kc,c,c,b,b,h")
 (plus: QI (match_operand:QI 1 "register_operand" "0,0,0,0,0,0,0")
 (match_operand:QI 2 "nonmemory_operand" "Wa,Wc,J,D,a,C,N")))]
 ""
 "@
 mv \\t*%0++m
 mv \\t*%0++m
 mv \\t*%0%C2
 mv \\t*%0%C2
 addh \\t%2, %H0
 addh \\t%c2, %H0
 addsp\\t%c2"
 [(set_attr "type" "move,move,move,move,ari,ari,move")
 (set_attr "n_islots" "1,1,1,2,1,1,1")]
 )

```

The first two output-lines have a register plus register syntax, the next are instead adding a constant to the register, the next two uses not the `mv` instruction but the `addh` instruction which adds to the high part of a 32 bit register. The last line prints out the `addsp` instruction which adds a constant to the stack pointer. Each output-string has the operators with their numbers after the percentage sign for output substitution.

## Miscellaneous

The target compiler is a C compiler, and the widths of C data types are defined to suit the DSP with macros such as

```
#define LONG_TYPE_SIZE 32
#define INT_TYPE_SIZE 16
```

Calling conventions of the existing compiler dictate where arguments and return values are placed, as well as saving and restoring of registers. It was not necessary to use the stack for arguments for any of the programs compiled, so this was not implemented (there are eight argument registers). The hook

```
#define FUNCTION_ARG(CUM, MODE, TYPE, NAMED)
```

is used to define which registers are used for passing arguments to functions. Each time the hook is called, CUM is updated so that the next argument register can be computed. The function body contains a switch, because an argument would be passed differently due to its TYPE (integer/pointer):

```
switch (TREE_CODE(type)) {
  case INTEGER_TYPE:
    if(mode==HImode)
      return (cum->a_regs < 4) ? gen_rtx_REG(mode,
        CREAMXL_REGNO_FIRST_A + cum->a_regs)
        : NULL_RTX;
    else if(mode==QImode)
      return (cum->a_regs < 4) ? gen_rtx_REG(mode,
        CREAMXL_REGNO_FIRST_A + cum->a_regs+1)
        : NULL_RTX;
    break;
  case POINTER_TYPE:
    return (cum->r_regs < 4) ? gen_rtx_REG(mode,
      CREAMXL_REGNO_FIRST_R + cum->r_regs)
      : NULL_RTX;
    break;
  default:
    return NULL_RTX;
    break;
}
```

For integer type data, there is also the matter of size. 16 bit arguments are passed in the high part of the 32 bit accumulator, therefore the '+1' increment from the cum value. The cum variable is incremented in a separate target hook, and is a struct of a\_regs and r\_regs, so that there are separate counters for each type of argument register. A so called RTX is generated and returned, an RTL eXpression for the register allocated, which is stored in the RTL IR list during compilation.

The prologue/epilogues are also part of the calling conventions and are generated in a pass after scheduling and IRA is done when the needed information is available, such as used registers. All hard registers which are live and callee saved are found with the following loop (fixed registers are those not considered for allocation, like the stack pointer):

```
for(i=0;i<FIRST_PSEUDO_REGISTER;i++) {
    if(df_regs_ever_live_p(i) && !call_used_regs[i] &&
        !fixed_regs[i])
        emit_insn(gen_push(gen_reg_RTX(QImode,i)));
    ...
}
```

The prologue/epilogue also allocates space for any local variables, which are found by the `get_frame_size()` function. Which registers are call used and/or fixed are defined in the header file.

Addressing modes concern the specifics of memory accesses on the machine. Which registers are used as address base registers is defined by

```
#define REGNO_OK_FOR_BASE_P(REGNO)
```

Further,

```
#define GO_IF_LEGITIMATE_ADDRESS(mode,x,label)
```

is another hook which is used by GCC by making queries regarding how to compose addresses in a legal form. It contains code like

```
if(CONSTANT_ADDRESS_P(X))
    return_val= 1;
if(GET_CODE(X)==REG && creamxl_regno_ok(REGNO(X)))
    return_val= 1;

if(GET_CODE(X)==PLUS)
{
    op1=XEXP(X,0);
    op2=XEXP(X,1);

    if(GET_CODE(op1)==REG && GET_CODE(op2)==CONST_INT &&
        creamxl_regno_ok(REGNO(op1)) && INTVAL(op2)>=0)
        return_val= 1;
    ...
}
```

so that for for instance there can be memory references by a single address register as well as by a register with a positive offset.

When running the compiler will try different sorts of memory references of various forms:

```

ALLOWED:      (plus:QI (reg/f:QI 78)                (const_int 1 [0x1]))
NOT ALLOWED: (plus:QI (reg/v/f:QI 76 [ sum ]) (reg:QI 77))
...

```

To the left is the returned value of the function above. It has allowed a register plus offset, but rejected a register as offset (the DSP actually supports this but was not needed). After having tried various forms, cost functions for different such RTX's are considered, and then a memory reference will be chosen for code generation.

The actual assembler syntax is defined by a series of macros and hooks.

```

#define ASM_OUTPUT_LABELREF(stream, name) \
fprintf(stream, "%s", name);

```

and many such others are defined to print out parts of the assembler output correctly. The

```

#define PRINT_OPERAND(STREAM, X, CODE) \
print_operand(STREAM, X, CODE)

#define PRINT_OPERAND_ADDRESS(STREAM, X) \
print_operand_address(STREAM, X)

```

hooks are implemented to print out the operands of the assembler output templates during operand substitution. Here, a function was added to get a register name dependent of the accessing mode, register 16 in HImode would be returned as "a0", while the same register number in QImode is "a01".

It is possible to turn of certain GCC passes which are not helpful for a DSP. The subregs passes were turned off, because they made things worse for the DSP as it was better not to split 32 bit registers. This might otherwise be beneficial for register allocation purposes, but did not suit the HI/QI-QI model described above. The block reorder pass was also omitted, as it was not suited for the nested loop programs that were handled.

## VLIW scheduling

Parallel execution is presumed by the DSP assembler when the '|' sign stands before an instruction (or rather, between two instructions), as seen in the example of chapter two.

VLIW bundling is a rather complicated task as it involves not only considering available hardware resources (for example CU's), but as well the VLIW issue packet which allows only up to 4 instructions, or 64 bits, in a one-cycle instruction word. Fortunately, GCC provides good support for implementing this critical DSP feature.

As part of the .md file, a pipeline description was added which is used when generating a DFA that the scheduler uses to decide whether it is necessary to advance to the next cycle before issuing the next instruction. This description is a model of available hardware resources (CU's, memory units etc) where (for FlexASIC) insns are mapped by their `type` and `n_islots` attributes to a certain resource reservation which is as well defined. Since the DSP is so simple in that all instructions have a one cycle latency, the pipeline description became as well simple.

An automaton is given a name and cpu units:

```
(define_automaton "creamxl")
(define_cpu_unit "pcu" "creamxl")
(define_cpu_unit "mv0" "creamxl")
...
```

Then, instructions are given resources by means of reservations. Here comes into play the `insn` attributes mentioned earlier:

```
(define_insn_reservation "ari1" 1 (and (eq_attr "type" "ari")
(eq_attr "n_islots" "1")) "(cu0|cu1|cu2)+one_islot")
```

Here, `ari1` is the name of a reservation with a one cycle latency for all insns of type `ari` which require a CU and one issue slot.

The VLIW issue slots were modeled as CPU units, which worked well as they can be considered resources with a one cycle latency:

```
(define_cpu_unit "islot0,islot1,islot2,islot3"
"creamxl")(define_reservation "one_issue_slot"
"(i_slot1|i_slot2|i_slot3|i_slot4)")
(define_reservation "two_issue_slots"
"i_slot1+(i_slot2|i_slot3|i_slot4) |
i_slot2+(i_slot1|i_slot3|i_slot4) |
i_slot3+(i_slot1|i_slot2|i_slot4) |
i_slot4+(i_slot1|i_slot2|i_slot3)")
```

This worked well during scheduling and appeared intuitive in the debug dumps and pipeline description, as can be seen above. The needs of different reservations were defined by the CPU resource needed along with a `+one_issue_slot` or `+two_issue_slots` as required.

With these two attributes, the scheduler will produce a legal arrangement of the instructions with consideration of the normal dependency-issues between instructions as well as of hardware resources in terms of CPU units and VLIW issue slots. The trick then is that the scheduler sets a mode for an instruction to be `TImode` (Tetra Integer mode, not used elsewhere) if it cannot be issued on the same cycle as the previous one and thus considered to start an issue group.

This flag is then checked in the `FINAL_PRESKAN_INSN` hook, which is called just prior to outputting an instruction:

```
void
creamxl_final_prescan_insn (rtx insn, rtx *opvec,
                           int noperands ATTRIBUTE_UNUSED)
{
    ...
    creamxl_curr_inst_parallel_bit_set = (GET_MODE(insn)!=TImode) ?
    1 : 0;
```

```
...
}
```

When the `creamxl_curr_inst_parallel_bit_set` is set, the '|' sign is output before the instruction in `ASM_OUTPUT_OPCODE`:

```
const char *
creamxl_asm_output_opcode (FILE *f, const char *ptr)
{
...
  if(creamxl_curr_inst_parallel_bit_set)
    fprintf(f, "| ");
...
}
```

This was all done without complications and thus GCC deserves credit for supporting VLIW scheduling very well.

## Hardware looping instructions

FlexASIC supports hardware loops similar to the ones seen in the example code. Three levels of loops can thus be executed, and there is as well an instruction for a single instruction packet repetition. There are however conditions on the loops which can use this facility, for instance a maximum number of instructions inside the loop body. The loops must be stored in I-memory with the code laid out with nested loops inside the outer loops.

GCC has many loop passes, both on GIMPLE and RTL. There is the `doloop` pass on RTL that supports hardware loop instructions. To make use of this pass, two patterns were defined:

```
(define_expand "doloop_begin" ...
and
(define_expand "doloop_end" ...
```

The `doloop` pass finds a loop that is suitable and replaces the normal looping instructions such as the typical compare/branch insns at the end of the loop body, with the two patterns. The `doloop_begin` pattern is optional but it was needed as the DSP uses loop counter registers which must be initialized.

After the pass, the RTL-list has changed with the new `doloop`-patterns inserted. There is still however the problem of the DSP conditions for using loop instructions. As this depends on the final code composition, it is not possible to perform this test during the `doloop`-pass, unfortunately.

As a consequence, this must be handled at the very end, during the `TARGET_MACHINE_\DEPENDENT_REORG` pass. Here, the entire RTL-code is scanned and all conditions of the ISA can be checked as needed. If possible, the pattern will be used as is, but it can also be replaced with a single-instruction repetition RTL-instruction if there is only one VLIW instruction in the loop body. In the worst case, the `doloop` pass must be canceled by reinserting the normal compare/branch instructions. This is a bit troublesome, as the loop counter register cannot be used with compare branch instructions, and IRA has already run. It has however not yet appeared necessary to handle as it was judged best to use `doloop` only on inner loops.

The savings in clock cycles from this pass is significant, as there are for each iteration at least three extra overhead instructions (increment, compare, branch), as well as three cycles of jump penalty.

However, the `doloop` pass is not entirely sufficient for the target, as it basically only handles the innermost loop, and for this it may be over conservative and avoid the transformation in some cases where it should not. The other loop passes transform the outer loops CFG layout so that they no longer are suitable for the FlexASIC looping instructions, and it is not possible to turn all these passes off, it seems. GCC gets credit for a DSP supporting pass, but it is not a complete solution for FlexASIC.

## Unrolling loops with explicit unroll factor

Unrolling of loops is supported by GCC at both GIMPLE and RTL. These passes are however not sufficient for the DSP. On GIMPLE, the “complete unroll” pass takes all loops with a small constant number of iterations and unrolls it completely when this is feasible. On RTL, the unroll pass is conservative and did not help any of the DSP test cases.

The idea then was to implement a loop unroll pass with an explicit unroll factor on SSA. The manual tuning is likely to pay off as any change in an inner loop is noticeable in performance. This would be achieved using a `__builtin_unroll` function call in the DSP source program. Builtins are function calls that the compiler treats specially by recognizing them as such. On the IR of the program, these calls have reference to a function declaration with a status of “builtin”. Thus, the IR code can be scanned for such calls in a pass and its arguments can be retrieved. If this is done on GIMPLE, as in this case, the call will disappear from the program during expansion to RTL if no expander definition is supplied in the `.md` file.

The pass was added on GIMPLE, because it is easier to work with and there was the induction variables optimization pass `ivopts`, which can be used after the unrolling pass.

The `__builtin_unroll` function was added by

```
add_builtin_function("__builtin_unroll", fn_void_int, CREAMXL_UNROLL
, BUILT_IN_MD, NULL, NULL);
```

in the `TARGET_INIT_BUILTINS` hook.

The next step was to add a GIMPLE pass in the proper place that extracts the unroll factor from the source code, and performs the unrolling. To add such a pass, a `GIMPLE_opt_pass` struct was declared and initialized:

```
struct GIMPLE_opt_pass pass_creamxl_builtin_unroll =
{
  {
    GIMPLE_PASS,
    "unroll_builtin",          /* name */
    creamxl_gate_unroll,      /* gate */
    creamxl_unroll_builtin,   /* execute */
    NULL,                     /* sub */
    NULL,                     /* next */
    0,                        /* static_pass_number */
```

```

0,          /* tv_id */
PROP_cfg,   /* properties_required */
0,         /* properties_provided */
0,         /* properties_destroyed */
0,         /* todo_flags_start */
TODO_dump_func |
TODO_cleanup_cfg |
TODO_update_ssa      /* todo_flags_finish */
}
};

```

The gate function is the function which decides whether or not to run the pass. The execute function is the function which is the entry point to the pass. Then follows various flags used by the compiler while managing all the passes, for example the `TODO_dump_func`-flag, which makes sure a debug dump will be written of the IR after the pass.

To insert the pass, the file `passes.c` is opened, and modified:

```

...
NEXT_PASS (pass_parallelize_loops);
NEXT_PASS (pass_loop_prefetch);
NEXT_PASS (pass_creamxl_builtin_unroll); /* added pass */
NEXT_PASS (pass_iv_optimize);
NEXT_PASS (pass_tree_loop_done);
...

```

Writing passes in GCC is done with the help of a wealth of macros and functions providing various interfaces. To iterate over all loops of the current function, one can use loop analysis with

```

loop_optimizer_init(LOOPS_HAVE_PREHEADERS |
                   LOOPS_HAVE_RECORDED_EXITS);
FOR_EACH_LOOP (li, loop, LI_ONLY_INNERMOST)
{
...
}

```

The loops are handled by first finding the `__builtin_unroll` function in a previous block. The call is found, and handled by

```

...
fun = GIMPLE_call_fndecl(stmt);
if(DECL_BUILT_IN_CLASS(fun) != BUILT_IN_MD ||
DECL_FUNCTION_CODE(fun) != CREAMXL_UNROLL)
    continue;

```

```
arg = GIMPLE_call_arg(stmt,0);
return int_cst_value(arg);
```

Then there is a decision to unroll the loop perfectly, or with trailing iterations, depending on number of iterations and the unroll factor. The latter alternative was ready to use from the other passes, whereas the perfect unrolling is done with a short added function based on the other passes as well.

Now code like this can be written for the DSP:

```
long dot_product(int *a, int *b)
{
    long s = 0;
    int i;
    __builtin_unroll(4);
    for (i = 0; i < 256; i++) {
        s +=(long) a[i]*b[i];
    }
    return s;
}
```

This feature could potentially give better performance as a result of comparing different unroll factors, as the optimal unroll factor may be difficult to guess beforehand. Unrolling gives the scheduler opportunity to reorganize code and achieve more parallelism.

### Accumulator variable expansion

The unroll pass is a partial solution, because ILP is inhibited between iterations due to the data flow. The next idea was to add another pass immediately after the unroll pass:

```
NEXT_PASS (pass_creamxl_builtin_unroll); /* added pass */
NEXT_PASS (pass_creamxl_expand_accv); /* added pass */
...
```

As can be seen, this first attempt was aimed at the mac instruction, but of course any variable can be expanded. Functions were added to make multiple accumulator variables inside the loop body and then sum them up after the loop has finished. As a result, parallel mac instructions can be executed inside the loop, each with its own destination register.

As this was done on GIMPLE SSA-form, it involved adding PHI functions and PHI operands for the expanded variables. Each loop which has been unrolled is searched for variables which are only defined inside the loop body. Then new variables are created along with several SSA-versions. In the unrolled loop body, each (except the first) instance of the original accumulator variable was thus replaced with a new SSA-variable. Add statements at the loop exit node are created by

```
add_stmt = GIMPLE_build_assign_with_ops (PLUS_EXPR,
                                         next_final_res, curr_final_res, result);
```

and added to the successor block by

```
gsi_insert_before(&add_pos, add_stmt, GSI_NEW_STMT);
```

## Tuning inner loops with ivopts

It was found that the addressing generated on the SSA-passes was not optimal for the DSP, and it was therefore tuned with the `ivopts` pass, which rewrites loop bodies according to cost functions defined in target hooks. `Ivopts` was thus made to use offset addressing instead of the register plus index form.

## Mac, mas

To rewrite a sequence of an add and a multiply instructions into a single mac-instruction, the combine pass was first investigated, and in the debug dump was found:

Failed to match this instruction:

```
(set (reg/v:HI 73 [ s ])
      (plus:HI (mult:HI (sign_extend:HI (reg:QI 80))
                       (sign_extend:HI (reg:QI 81)))
        (reg/v:HI 73 [ s ])))
```

It was trying to combine the arithmetic operations into a mac, but could not do so because it had not been added to the `.md` file yet. Thus, the mac pattern was added:

```
(define_insn "mac"
  [(set (match_operand:HI 0 "register_operand" "=jf")
        (plus:HI (mult:HI (sign_extend:HI (match_operand:QI 1
                                           "register_operand" "b"))
                       (sign_extend:HI (match_operand:QI 2
                                           "register_operand" "b")))
          (match_operand:HI 3 "register_operand" "0")))
  ])
"
"mac \\t%1, %2, %0 \\t /* 3.4.1.31,1 islot */"
[(set_attr "type" "mac")
 (set_attr "n_islots" "1")]
)
```

Now the compiler would generate unrolled loops with parallel macs, which was quite excellent at least for dot products. The `mas` instruction is the equivalent for subtraction, which was also added along with `umac` and `umas` for the unsigned operations.

## Auto-incremented addressing

FlexASIC supports post modification of address registers, but not pre modifications. The macros

```
#define HAVE_POST_DECREMENT 1
#define HAVE_POST_INCREMENT 1
#define HAVE_POST_MODIFY 1
```

signal that the target can use this feature. For increment/decrement this will mean a change of 16 or 32 bits depending on the data type. A post-modify insn has a step greater than one.

To make the GCC `auto-inc-dec` pass rewrite the RTL IR to use auto-incremented addressing, the macros were defined, and in the `.md` file was added patterns like

```
(define_insn "mem_to_reg_qi_with_post_inc"
  [(set (match_operand:QI 0 "register_operand" "=f,e,e")
        (mem:QI(post_inc (match_operand:QI 1
                          "register_operand" "c,c,h"))))]
  ""
  "@
    mv \t*%1++, %0 \t /* 3.3.1.3, 1 islot */
    mv \t*%1++, %0 \t /* 3.3.4.5, 2 islots */
    pop \t%0 \t /* 3.3.1.12,1 islot */"
  [(set_attr "type" "move,move,move")
   (set_attr "n_islots" "1,2,1")])
```

This is a pattern with an RTL-operator for post increment.

This feature lets the DSP save an instruction for modifying the address register and is basically supported well by GCC. A drawback is however that `auto-inc-dec` does not seem to work with offsets used in addressing. If `ivopts` is made to produce offset addressing, `auto-inc-dec` will not combine the increments for such addresses.

## Widening multiplication

When it came to multiplication, some issues arose. The DSP supports true widening multiplication, meaning that a 32 bit result will be computed and stored when multiplying two 16 bit operands. GCC on the other hand, will perform a sign extension of the 16 bit product and deliver it as a 32 bit “widening” result. As the result of an overflowing multiplication is undefined per standard C, this is not an actual error on behalf of GCC.

The following simple test function

```
void fun(int *x, int a, int b)
{
    long real=0;
```

```

    real += a * b;
    real += a * b;
    *x = real;
}

```

results in the following GIMPLE code after the SSA-passes:

```

fun (int * x, int a, int b)
{
    long int D.1194;
<bb 2>:
    D.1194 = (long int) (b * a);
    *x = (int) (int) (D.1194 + D.1194);
    return;
}

```

The multiplication result is converted to long, as is reflected in the RTL-expansion dump for the statement:

```

;; Generating RTL for gimple basic block 2
;; D.1194 = (long int) (b * a);

(insn 8 7 9 /home/ejonpan/gcc/trunk/testsuite/test11.c:4 (set
(reg:HI 58)
  (mult:HI (sign_extend:HI (reg/v:QI 57 [ b ]))
    (sign_extend:HI (reg/v:QI 56 [ a ])))) -1 (nil))

(insn 9 8 10 /home/ejonpan/gcc/trunk/testsuite/test11.c:4 (set
(reg:QI 59)
  (subreg:QI (reg:HI 58) 0)) -1 (nil))

(insn 10 9 0 /home/ejonpan/gcc/trunk/testsuite/test11.c:4 (set
(reg:HI 54 [ D.1194 ])
  (sign_extend:HI (reg:QI 59))) -1 (nil))
...

```

The DSP and existing compiler however implements true widening multiplication, where the (wider) result is stored unmodified in the 32 bit destination. With a (long) cast in the C-code, like `real += (long) a * b;...`, the GIMPLE would become `D.1195 = (long int) b * (long int) a;...`, which actually gave the desired RTL-expansion with a widening multiplication. At this point it was reasonable to insert this type casting into the benchmarks for GCC, as it made sense from a C perspective.

This was however not all. As seen above, without the `(long)` cast, the DSP result of a widening multiplication would not be achieved. However, with this cast, code started to appear which would perform the multiplication as HI multiplication. This happened with multiple uses within a loop body. In the case of a true 32 bit multiplication, there are a series of instructions involved for performing this, and this was part of the `.md` file as a HI multiplication. But in this case this would lead to the same type of error as described above.

It so happened that different ways of solving this was explored.

### **GIMPLE pass**

The first remedy that was tried was a simple GIMPLE-pass that helped the code just on this detail. It was discovered that the `out-of-ssa` pass was rewriting expressions with 16 bit operands by means of an extra 32 bit variable that was assigned the 16 bit value. This would appear in a type conversion statement and a resulting 32 bit operand would be used instead in the multiplication statement.

This happened in two places: the `ssa-pre` pass would introduce a new temporary before the loop and the `out-of-ssa` pass would insert a new assignment statement for a new temporary of the long type. Both cases would lead to an explicit type conversion performed on RTL and the use of the `mulhi` pattern.

The `ssa-pre` pass was turned off, and the new GIMPLE pass was then successful in avoiding the temporary. By simply introducing extra copies of the 16 bit variables this rewriting was avoided and as a result, the multiplication was performed without the type casting, after expanding to the widening RTL-pattern.

### **mulhi expansion**

It was interesting to experiment with GIMPLE, but it was not a very elegant solution. PRE is not to be turned off (even though it did not affect performance for the DSP benchmarks at the time), and rewriting the IR should be avoided if possible.

Next, a more reasonable way of handling this was approached, by working in the `.md` file instead of with the GIMPLE code. The idea was to check each time a (RTL) 32 bit multiplication was to be performed, if this was two operands that had just been type casted from 32 bits to 16 bits, as would happen for the programs compiled with the `(long)` cast inside loops with multiple uses. If so, replace the operands with their 16 bit versions found in the conversion instruction and perform the `mulqihi3` pattern instead (widening). This was possible to do in the `.md` file with:

```
...
(define_expand "mulhi3"
...
qi1=find_qi_original(operands[1]);
qi2=find_qi_original(operands[2]);
if(qi1 && qi2){          //both operands have been extended
    emit_insn(gen_mulqihi3(operands[0],qi1,qi2));
    DONE;
```

```
}
```

This code emits the desired instruction and ends with `DONE`, which is a type of return macro used so that there will be nothing else done after that point.

The `find_qi_original` function was written in the `.c` file and simply checks for an instruction performing a sign extension as the definition of the pseudo register.

```
...
if((insn=NEXT_INSN(insn)) &&
    INSN_CODE(insn)==CODE_FOR_sext)
    return orig_reg;
```

This worked equally well, without the need to add a pass to the compiler or to turn `ssa-pre` off.

The reason that one first would introduce the `(long)` cast and then rid the code of the 32 bit RTL conversion, is as explained above that without the cast GCC would interpret the multiplication undesirably, and with multiple uses inside loops, the type conversion becomes explicit with a temporary variable, leading to the HI multiplication expansion.

### **define\_insn\_and\_split**

The best solution possible would be to achieve this result directly in the instruction patterns, without scanning the code (in new functions). There is a way of doing this, which involves the `combine` pass that combines instructions and the `split` pass that splits instructions.

It would then work like this: a wide `mulhi` would be generated with two HI operands that would have been sign extended earlier. Instead of handling this explicitly at RTL-expansion, the `combine` pass would be used to combine the sign extending and multiplying instructions, if both operands were originally `QImode` operands. If not, then the `split` pass would split up the operation into the series of instructions per `mulhi`. The trick is that the `split` pass is run after the combiner, so if `combine` succeeds, the `split` pass will not execute on the original multiplication instruction, as it has been replaced by the combiner earlier. This works by using a `define_insn_and_split` pattern.

This was however not to turn out successful. The combiner pass can only operate locally within a basic block, and thus fails with the PRE temporaries outside the loop. Unfortunate, as it would have been a swift and effective solution while avoiding the trouble of adding more code.

True widening multiplication was thus not directly supported by GCC, but was handled with a `define_expand` and a simple added function for finding the QI original registers.

### **Predicated instructions**

The DSP supports conditional execution by means of adding a condition to any ISA instruction. This takes extra space from the VLIW-packet, but in exchange one may gain reduction of jumps which equals faster execution.

GCC has a pass called `if-convert` which is designed to replace if-clauses with conditional execution wherever possible. Furthermore, there is in GCC the option of using the abstract `cc0` condition code register, or instead a defined register (in the `.h` file) for the purpose of storing condition codes of compare instructions. The `cc0` register is typically used so that all instructions

are considered to implicitly clobber (potentially update) it, whereas a condition code register on the other hand can provide better scheduling if not all of the machines instructions affect it (by separating compare and branch instructions).

FlexASIC has not such a single condition code register, but instead each accumulator has its own flag register. This makes `cc0` preferred, as any instructions defining an accumulator will affect its flag register. However, it is still desired to make the best possible scheduling, as in many cases the compare instruction can be parallelized if moved up in the block from the branch instruction (in the cases where `do1oop` has replaced these instructions, this is not pertinent).

There is actually the possibility of avoiding the compare instruction entirely, as the flag register may already have been set in an arithmetic operation thus reflecting its result. This may theoretically be used as a basis for rewriting the code and eliminating compares, but in practice the hardware loop instruction makes this of lesser importance.

In order to use the `if-convert` pass, there can not be an implicit update by instructions (at least not by if-blocks bigger than one single instruction).

This task thus possesses a conflict, as FlexASIC has multiple flag registers which cannot be modeled adequately by GCC, which basically only handles a single condition code register.

The DSP instructions can be set to inhibit flag updates, which needs to be done for a series of predicated instructions in case the actual flag register is changed by one of them.

By releasing the implicit update of the condition code register, the `if-convert` pass can do its work.

The first thing to do was to add a simple

```
(define_cond_exec
  [(cond_code (reg:CC CC_REGNUM) (const_int 0))]
  "TARGET_predicates"
  " ,<code>: "
  )
```

GCC will now, with a new “predicable” attribute added in the `.md` file, generate for every instruction in the `.md` file a new instruction which begins with

```
(define_insn ""
  [(cond_exec
  ...
```

with the same pattern as before.

Now the `if-convert` pass will succeed in replacing if-bodies with conditional execution as in:

```
cmp    a1, a2
brr    #L3, .a2:ge
copy   a1, a2
copy   a1, a3
L3:
...
```

which becomes

```
cmp    a1, a2
copy  a1, a2, .a2:lt,inh
copy  a1, a3, .a2:lt,inh
...
```

The copy instructions have a condition added, which is based on the flag register for a2. The inhibit flag is also set, so that the flag of the a2 accumulator will not be updated. In this way, several predicated instructions can follow a single compare as the DSP executes.

As a consequence of adding a condition to a DSP instruction is the cost in issue width: 16 bits are required to hold the condition. A problem with this is that the instruction attribute added for holding the number of issue slots can not be adjusted for the new `cond_exec` instruction, as all attributes are hard-coded into the compiler – they are not variables. What's worse is that there was not found another attribute to distinguish an ordinary instruction from its conditional counterpart. Since the DFA used in the scheduling pass can operate solely on attributes, the VLIW scheduling was now incorrect as all predicated instructions were one size bigger than their attribute value.

A quick solution was tried, by scanning the code for such instructions and starting a new cycle wherever needed, with respect to the added conditions. This may withdraw somewhat from the benefits of using predicated instructions with GCC. It may be so that smaller modifications to the GCC sources in terms of adjusting the `n_islots` attribute for `cond_exec` insns might pay off, but this is a greater domain than that of this degree project.

There was little benefit in the results of the predicated code with the programs compiled. Only in one case were there actually if-clauses inside a loop body that got converted. This however led to no speedup, it was a perfect tie between the versions of predicated instructions or the jumps.

A problem of not updating the condition code register, is that the compare instruction can be rescheduled freely, as GCC is not aware of the actual flag registers updates, which happen every time an accumulator is set. This would lead to undefined behavior if not handled.

It so happened that this feature could be tried still, as the compare instructions only rarely get misplaced. This has not yet been handled as other points have been of higher priority than predicated instructions for FlexASIC.

### **SIMD memory accesses**

By combining adjacent 16 bit loads/stores into single 32 bit instructions, the two memory accesses per cycle of FlexASIC can be used with doubled efficiency. Whenever there is an access like `a[0]`, `a[1]`, the data can be loaded with one instruction by putting the 16 bit values in the low/high parts of the 32 bit accumulator.

It would have been nice to express this in MD-RTL with a simple `insn-pattern`, but unfortunately this cannot be done. The code is first RTL-expanded into separate load/store insns. The problem with combining them later is that the data registers defined/used by these two instructions are two separate QI register, and there is unfortunately no GCC pass that handles this.

The only way to implement this point then is to add a pass that will replace the original QI-reg

with subregs of a new wide register, and then replace/remove load/store insns as needed.

First off, a pass was inserted after auto-inc-dec which then would handle the somewhat simpler cases of insns representing both memory access and address pseudo update. It worked fairly well, but it was found that auto-inc-dec would not work with offsets in addressing as was planned.

Because auto-inc-dec only would handle the cases of an access followed by an address register modification, a second effort was conceived to work independently of auto-inc-dec, rewriting offsets as well. Whenever possible, two adjacent accesses are combined, and if possible, a post-inc/dec instruction is emitted, otherwise just a regular 32 bit load/store. The increment/decrement will then be halved, as the DSP interprets it according to the word size used. In case an argument (dependent) register is used for the modification, it is not combined to a post-inc/dec, as it is of unknown value.

This pass was based on the RTL data flow functionality of GCC. After having located a load instruction `insn` in the loop body, the following code

```
use_rec      = DF_INSN_USES (insn);
link         = DF_REF_CHAIN(*use_rec);
addr_def_insn = DF_INSN_UID_GET(DF_REF_INSN_UID(link->ref))->insn;
def_rec      = DF_INSN_DEFS (addr_def_insn);
link         = DF_REF_CHAIN(*def_rec);
```

finds the used address register, its definition, and then `addr_def_insn`'s uses. The uses of the address register definition are then scanned for adjacent loads (with an offset of one):

```
for(;link;link = link->next){ //uses
    use_insn = DF_INSN_UID_GET(DF_REF_INSN_UID(link->ref))->insn;

    if (INSN_CODE(use_insn)==CODE_FOR_mem_to_reg_w_pos_offset_qi &&
        INTVAL(XEXP(XEXP(XEXP(PATTERN(use_insn),1),0),1))==1){
    ...
```

## Mode switching

The DSP has instructions that can take advantage of different mode settings. If the `cuc` register is set a certain way, the operands of multiply (and `mac/mas`) instructions are interpreted as either signed or unsigned. If the `cuc` mode is not touched, it is the default to have one of the operands of each kind for such instructions. This was an important implementation detail, as many inner loops contain `mac/mas` instructions, which use SIMD loaded data which appears in a register pair. Mode switching rids the code of any unnecessary copying in this context, as the operands can be used directly, in either the high or low part of an accumulator.

GCC supports mode switching of entities by means of a few macros and hooks:

```
#define OPTIMIZE_MODE_SWITCHING(ENTITY) TARGET_mul_modes

#define NUM_MODES_FOR_MODE_SWITCHING {3}

#define CUC_SIGNED 0
```

```

#define CUC_UNSIGNED 1
#define CUC_S_US 2
#define MODE_NEEDED(ENTITY, I) \
creamxl_mode_needed((ENTITY), (I))

#define EMIT_MODE_SET(ENTITY, MODE, HARD_REGS_LIVE) \
creamxl_emit_mode_set((ENTITY), (MODE), (HARD_REGS_LIVE))

```

The ENTITY is in this case the cuc register, and I is any RTL-insn.

Mode switching was also on the agenda for the case of loading accumulators with automatic sign-extension. It would save one instruction to let the DSP load 16 bits and automatically sign/zero extend to the higher part, which it can do if the mode for the particular accumulator is set up correctly. This however was, just as it was for accumulator flag registers, a more complicated task left undone. Triviality is lost when mode switching depends on a specific hard register, as it is done on pseudos, prior to IRA. One would basically have to write a machine dependent mode switching pass after IRA, placing out mode switching instructions where such loading is used.

### The reorg pass

Immediately before the assembler output stage a pass for machine dependent fix ups (reorganization) is run, which is where certain left-over issues are dealt with as needed.

The first thing done for FlexASIC is to handle loop-invariants caused by reload. There are special registers that must be used for modifying an address register which reload will load with an immediate if necessary. Unfortunately, this constant-to-register insn will not be handled at the later stages of compilation and must therefore be moved out of the loop here.

The `doloop_begin` insn is ensured to lie in the proper place. There may have been emitted another instruction after `doloop`, in which case the hardware loop instruction will not be outputted correctly.

The conditions for the hardware loop instructions (see previous chapter) are checked, and a single block repetition might take the place of the normal looping instructions if possible. The re-inserting of compare/branch instructions in the case of any condition breach here has not been implemented, as it is yet to happen.

After this, VLIW scheduling is done again without rearranging instructions, to assure correct VLIW-packaging, after any changes to the list. The DFA is queried for instruction issue on each successive cycle, as are the data flow functions.

Pipeline conflicts can occur in FlexASIC, for instance when modifying an address register and using it with an offset in the next cycle. In such cases a `nop` must be inserted, which is also taking place here.

## 5 Results

### The dot product example

This thesis began in chapter two with an example code which has been used as a basis during the project.

The source code is:

```
long dot_product(int *a, int *b)
{
    long s = 0;
    int i;
#ifdef UNROLL_FAC
    __builtin_unroll(UNROLL_FAC);
#endif
    for (i = 0; i < N; i++)
        s += (long)a[i]*b[i];
    return s;
}
```

The `#ifdef` clause means that the code can be compiled with something like `-D UNROLL_FAC=2`, to unroll the loop two times, for example.

The loop has been improved in steps, implementing the relevant DSP features discussed. Here is a walk through of these progressive improvements, with a measurement of the effect on execution these optimizations give at the end.

<i>Compilation</i>	<i>.s code of inner loop body</i>	<i>comments</i>
1	<pre> L2:  mv    a4h, a2h       mv    a0h, r0       addh  a0h, a2       mv    *r0++       mv    a2h, r3       mv    a5h, a2h       addh  a0h, a2       mv    *r3, a3h       mv    r0, a0h       mv    a2h, r2       mv    *r2, a2h       mpy   a2h, a3h, a2       add   a1, a2, a1       cmp   256, a0h       brr   #L2, .a0:ne </pre>	<p>Computing addresses and incrementing the loop index without DSP optimizations.</p> <p>Brr costs three cycles (taken branches).</p> <p>15+4 instructions per loop iteration</p>
2	<pre> L2:  mv    a4h, a2h       mv    a0h, r0       addh  a0h, a2       mv    *r0++       mv    a2h, r3       mv    a5h, a2h       addh  a0h, a2       mv    *r3, a3h       mv    r0, a0h       mv    a2h, r2       mv    *r2, a2h       <b>mac</b>  <b>a3h, a2h, a1</b>       cmp   256, a0h       brr   #L2, .a0:ne </pre>	<p>Using the mac instruction performs the mpy and add on one cycle.</p> <p>14+4 cycles per iteration</p>

<i>Compilation</i>	<i>.s code of inner loop body</i>	<i>comments</i>
3	<pre> L2:      mv      a4h, a2h                mv      a0h, r0                addh   a0h, a2                mv      *r0++                mv      a2h, r3                mv      a5h, a2h                addh   a0h, a2                mv      *r3, a3h                mv      r0, a0h                mv      a2h, r2                mv      *r2, a2h                mac    a3h, a2h, a1                cmp    256, a0h                brr   #L2, .a0:ne </pre>	<p>VLIW scheduling gives great benefit, as all instructions preceded by '   ' are executed on the same cycle as the previous one.</p> <p>8+4 cycles per iteration</p>
4	<pre> <b>mv      255, brcl</b> <b>bkrep  #L6</b>  .align 4 L2:      mv      r2, a1l                mv      a3h, a1h                addh   a1l, a1                mv      *r2++                mv      a1h, r4                mv      a4h, a1h                addh   a1l, a1                mv      *r4, a2h                mv      a1h, r3                mv      *r3, a1h                          .align 4 L6:      mac    a2h, a1h, a0 </pre>	<p>The hardware loop instruction saves much overhead for each iteration.</p> <p>7 cycles per iteration</p>

<i>Compilation</i>	<i>.s code of inner loop body</i>	<i>comments</i>
5	<pre> mv    0, cuc   mv   2, m0  mv    127, brcl bkrep #L6lign 4 L2:   mv   *r1, a3h   mv   *r0, a3l mv    *r1(1), a2h   mv   *r0(1), a2l   mac  a3h, a3l, a1 .align 4 L6:   mac  a2h, a2l, a0   mv   *r0++m   mv   *r1++m </pre>	<p>mode-switching instruction</p> <p>127 iterations, loop unrolled twice</p> <p>mac operands in high and low parts</p> <p>3 cycles for 2 iterations</p>
6	<pre> mv    0, cuc   mv   127, brcl .align 4 L2:   mv   *r1++, a3   mv   *r0++, a2 .align 4 L6:   mac  a3l, a2l, a0   mac  a3h, a2h, a1 </pre>	<p>SIMD load with post increment</p> <p>2 cycles for 2 iterations</p>

<i>Compilation</i>	<i>.s code of inner loop body</i>	<i>comments</i>
7	<pre> mv    0, cuc   mv   63, brcl     bkrep #L6     .align 4 L2:   mv   *r1++, a7       mv   *r0++, a6     mv   *r1++, a5       mv   *r0++, a4       mac  a7l, a6l, a1       mac  a7h, a6h, a2     .align 4 L6:   mac  a5l, a4l, a0       mac  a5h, a4h, a3     ...     add   a3, a2, a2       add  a2, a1, a1       add  a1, a0, a0       mv   8, cuc </pre>	<p>63 iterations, loop unrolled 4 times</p> <p>3 cycles for 4 iterations</p> <p>Summing up the expanded variables when loop is done</p> <p>resetting mode for CU's</p>

The codes above were run on the simulator and all found to give a correct return value. Here are the results in clock cycles as reported by the simulator:

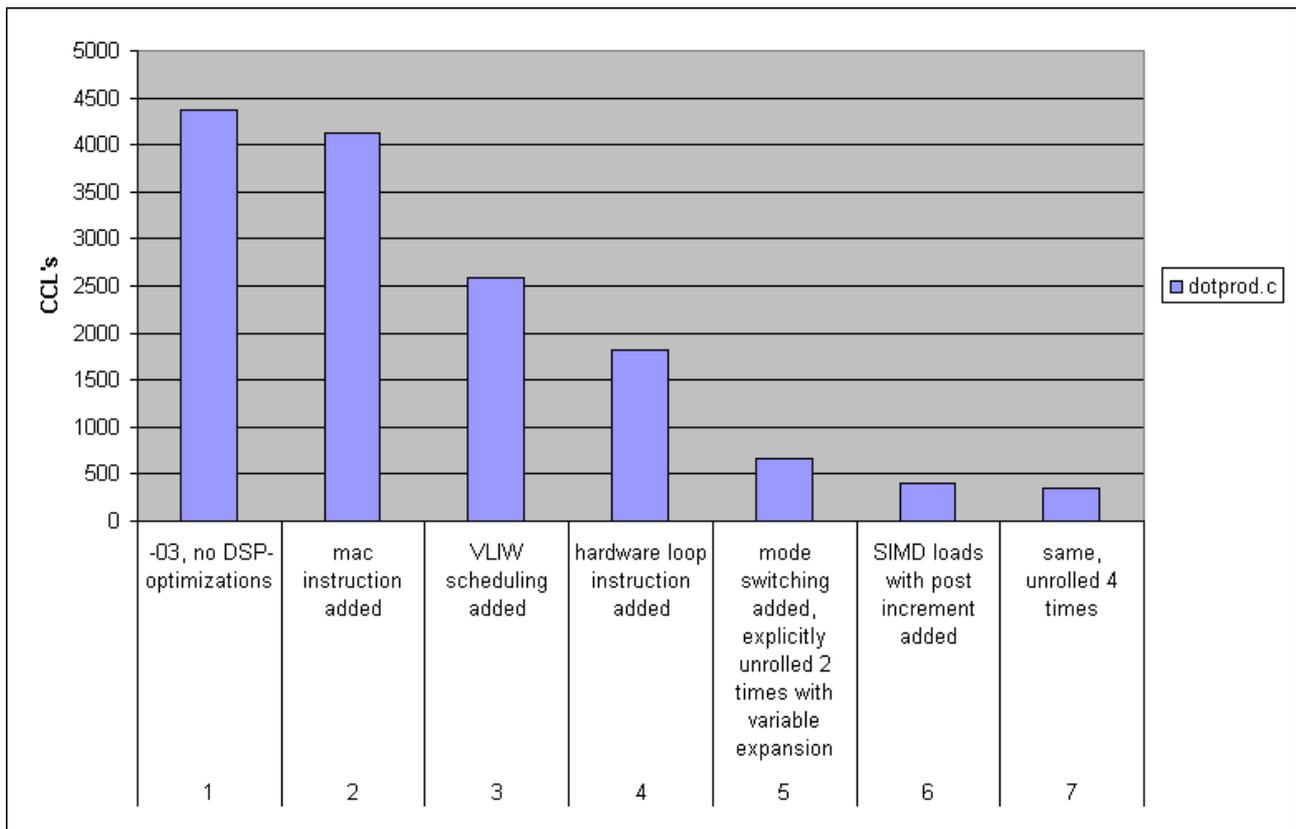


Figure 2: Compilation results of dotprod.c

This program is not very typical of the benchmarks for FlexASIC, however. It was used as a test program for which most of the various DSP features are useful. The DSP can do about ten times better with these optimizations it seems, even though the first version is actually not of a satisfactory standard. The reason for this is that while working on the compiler certain deficiencies vanished as other areas were improved, so this chart serves mainly as a display of step-wise improvements.

Looking at the .s code, there is still the opportunity for modulo-scheduling, which was not done due to lack of time. Software pipelining would achieve the ultimate version of two cycles in the inner loop of version seven above. There is a modulo scheduling pass in GCC, but it does unfortunately not work with `post-inc` insns or similar.

### Compilation of benchmarks

The existing compiler has been tested with a benchmark suite to measure improvements between versions. The source codes for these were provided by Ericsson in Kista, and a handful were chosen (randomly), to measure the success of the GCC adaption during the project. These programs are more complex than the simple dot product loop, and include for example triply nested loops, widening multiplication operations and store operations.

The results of compiling these benchmarks have been measured with cycle accuracy on the simulator for the target, and have as well all been tested closely in regards to the output matching exactly that of the existing compiler. Here is a display of the GCC results for the programs that

were compiled and most centered on while working on the project, that shows that the FlexASIC GCC port is capable of producing competitive code for these functions as of today. Alongside each measurement for GCC , a measurement for the existing compiler with highest optimization possible is presented.

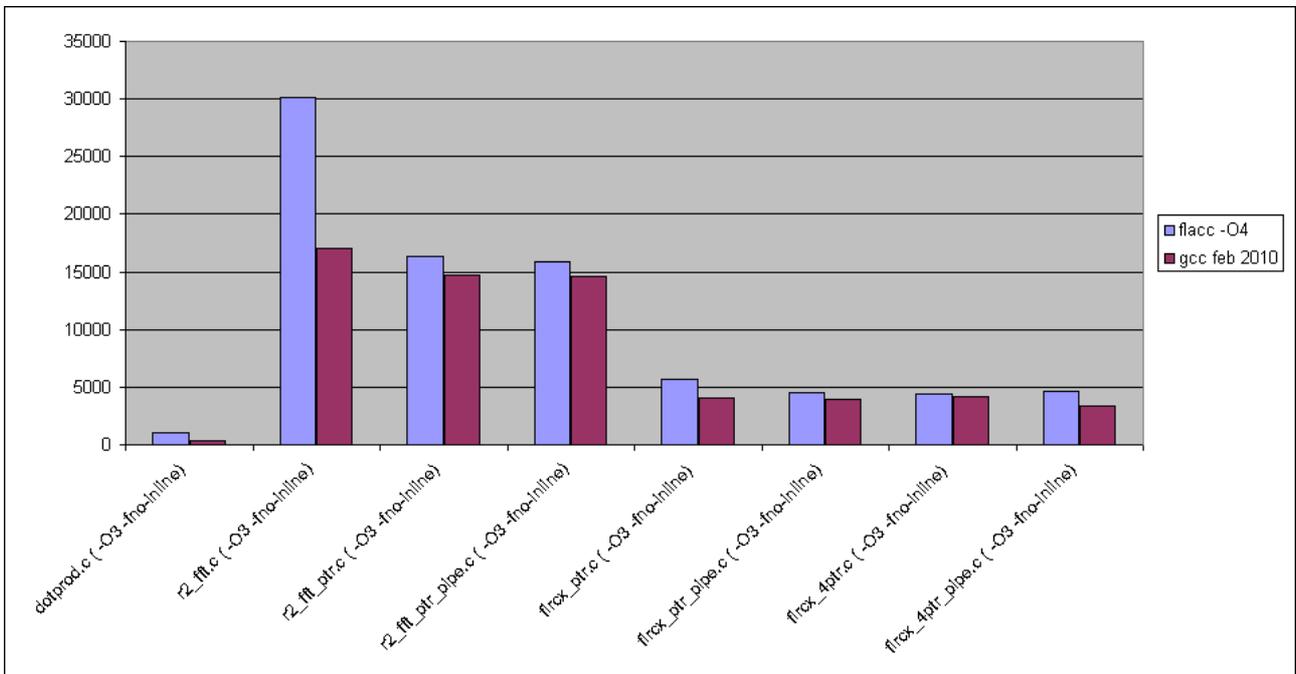


Figure 3: Comparison between GCC and existing compiler on dotprod.c and seven benchmark programs.

To the left the seventh (best) version of dotprod.c is compared. The next example shows that the existing compiler is not doing so well with indexed addressing, whereas gcc is managing nearly as well as with pointer arithmetic, seen next to the right. It is very promising to see GCC produce faster code for so many of the benchmarks, even within the time frame of a degree project.

From the twenty or so benchmarks in the suite, thirteen have been worked on (without any special preference) and seven are displayed here. The six others to which GCC was not quite attuned at the moment of this writing, are quite similar in structure, and therefore not considered necessary for the sake of demonstrative results. It is estimated that they as well can be brought to better ends, even though GCC has not produced code of similar quality yet.

The unroll and variables expansion passes were not found beneficial for these benchmarks yet. VLIW scheduling, hardware loops, mac/mas instructions, SIMD memory loads with mode switching were key to improved performance.

## 6 Conclusions

### GCC as a DSP compiler

GCC features powerful optimization algorithms, both on SSA form and on the machine dependent RTL code. FlexASIC requires as well specialized optimizations as laid out in this thesis.

Some of these features are fully supported by GCC. The VLIW scheduling task is solved well, except for the `cond_exec` patterns which have the wrong `n_slots` value. The substitution for `mac/mas` instructions are handled very well by the `combine` pass. Mode switching is taken care of by a GCC pass as well.

Other features are handled well, but not to the extent wished for. The `doloop` pass substitutes the inner loop most of the time, but it may be over conservative and avoid the optimization. It will not arrange the CFG in a fashionable form so as to allow nested hardware looping instructions. `Auto-inc-dec` would not work on offsets in addressing, which is unfortunate as this seems to be the preferred addressing mode.

Yet other features required addition of optimization passes. The SIMD memory instructions had to be implemented by hand for the DSP. This was a rather fair task, utilizing the data flow framework and more.

There was no software pipelining available, as the modulo scheduling / selective scheduling passes are not compatible with auto-increment instructions, which were considered more basic and prioritized during the project. This scheduling optimization is the single most important optimization left undone.

The irregular register file of FlexASIC has been handled as described without much further exploration. There seems to be more opportunity here for improvement. The IRA use of cover classes and its effect on reloading have not been explored further, but it is clear that the amount of reloaded registers are a bit higher than necessary.

GCC is providing a nice interface for IR manipulation which make it quite straight forward to implement optimization passes, both on SSA and RTL. This means one could do practically anything one wished for, at a very precise level for the DSP.

There may be a gap between the machine independent passes of GCC, and the results attained by any specific machine, in terms of optimal code. In the same way, there is a level of FlexASIC specific coding lost using the GCC DSP supporting passes, such as `doloop`. This makes the framework for adding passes an important factor of GCC DSP suitability.

It has been rewarding to find that GCC could give comparable results to that of the existing compiler, as it is true that the current results can be improved even further.

### Todo

Here is a summary of what seems most missing as of today.

### Register reloading

IRA/reload should be made to work better, as there often is reloading done when it is in fact unnecessary. This involves possible editing of the register classes, constraints and cover classes.

## **Modulo scheduling**

Many inner loops begin an iteration with a load instruction, and it is therefore quite clear that further ILP can be achieved by software pipelining.

An alternative might be to use the modulo scheduling pass of GCC, with `auto-inc-dec` turned off, and after it handle SIMD memory accesses and incrementing instructions.

There is also the option of implementing a modulo scheduling pass from scratch<sup>[7]</sup>, which would give the opportunity for trimming it to suit FlexASIC as best possible.

It might be worth trying instead an implementation of a simpler pass that focuses on two iterations by finding the tightest VLIW-packed loop available from them. This would handle most of the obviously missing ILP, it is estimated.

## **Nested hardware loops.**

There is the possibility of rearranging nested loops at the end of compilation to try to make them reside in a suitable layout for hardware loop instructions. As mentioned earlier, the loop passes will optimize the CFG, but as well then loose the code layout needed by FlexASIC.

This might be done as a separate pass, which could as well take over the role from `doloop` entirely, as the existing pass is over conservative at times. If such a pass is inserted late in the pass list, it would be possible for it to intelligently check the FlexASIC conditions on the code for hardware looping. This benefit would be an improvement, as it is actually unfeasible to undo the work of `doloop` after IRA and reload, which may alter the code.

## **Rescheduling of compare instructions**

This is needed whenever hardware loops are not used. Part of such a function has been written for hiding the compare within the available VLIW space.

As predicated instructions are not currently considered and therefore the implicit updates of the condition code registers are present, this issue can rest until such updates are found counter productive.

## **Instructions coverage, code acceptance**

Only basic and obviously needed instructions have been entered into the machine description. There are many alternate instructions not used.

Not too many programs have passed through the compiler as of now. Some further work on this would give more stability, as a missing insn pattern might crash a compilation.

## **Passing of arguments on the stack**

Has not been needed yet.

## **Modulo addressing**

Has not been implemented and has not been called for on any of the benchmarks tested so far.

## **Unroll / variables expansion passes**

These passes have not been found profitable yet on the benchmarks, because the IRA/reload part of

the compiler is not used well enough to avoid unnecessary register copying. This may change if reloading is reduced.

The simple variables expansion pass only works with mac instructions right now, which needs to be extended. It would be interesting to compare the results with a modulo scheduling algorithm.

### **Flag registers and configuration registers**

Mode switching each single accumulator configuration register would give automatic sign/zero extension into the higher part. Similarly, each flag register may be used for conditional execution, including branches.

These features are not directly supported by GCC, but can be handled with added passes after register allocation.

For the flag registers, one could try to eliminate unnecessary compares. `ivopts` could be tuned to choose a code form where this is more often possible, perhaps. The hardware loops are however the dominant issue here.

In order to mode switch the accumulator configuration registers, a pass after IRA could insert mode setting instructions in proper places. If done after the last scheduling pass, it might be possible to try if this option is desirable or not, by means of testing the VLIW packaging results.

These two points would not give much benefit as to what has so far been compiled.

## References

1. Optimizing an Architectural Simulator for the Flexible ASIC Architecture (thesis), Tomas Östlund.
2. The Conceptual Structure of GCC, Abhijat Vichare, IITB, Bombay, <http://www.cse.iitb.ac.in/grc/docs.html>.
3. Computer Architecture, A quantitative approach, 4th ed, Hennessy&Patterson.
4. Datorsystem, program och maskinvara, Brorsson.
5. An Introduction to the Theory of Optimising Compilers with Performance Measurements on Power Processors, Jonas Skeppstedt LTH.
6. GCC Internals, <http://gcc.gnu.org/onlinedocs/gcc-4.4.1/gccint/index.html>.
7. Modern compiler implementation in Java, second edition, Appel.

