Diploma Thesis

Optimizing the GCC Suite for a VLIW Architecture

Author: Adrian Strätling
Supervisor: Dr. Robert Baumgartl
Date of Submission: 18.11.2004
Adrian Strätling

*Optimizing the GCC Suite for a VLIW Architecture*

Diploma Thesis, Chemnitz University of Technology, 2004
Abstract

A Compiler is the more useful the faster code it produces. Ideally it should allow the programmer to concentrate only on writing correct and readable code. Therefore, it is necessary that it optimizes the program while translating it into a hardware dependent binary.

This work examines the GCC suite with reference to its optimizing capabilities. It extends the existent TMS320C6x GCC back end to benefit from the GCC optimization infrastructure. It explains the changes to the machine description which lead to better performance.

Depending on the code to translate, a speedup of up to 300% is measurable. Due to the limited scope of this work, the paper will not only discuss conceptional limits, but also limitations certainly surpassable by further development.
Declaration of Authorship

I hereby declare that the whole of this diploma thesis is my own work, except where explicitly stated otherwise in the text or in the bibliography.

This work is submitted to Chemnitz University of Technology as a requirement for being awarded a diploma in Computer Science (“Diplom-Informatik”). I declare that it has not been submitted in whole, or in part, for any other degree.
Notational Conventions

In this work, the following notational conventions will be used:

- Text passages that are meant to draw the reader’s attention to them, even when only skimming over the text, will be typeset in **bold series**.

- C, RTL, and Assembler source code will be typeset in *typewriter* family, where:
  - Single variables, statements and function names are expressed by *without*, whole instructions ‘*within*’ single quotation marks.
  - Standard Pattern Names are shown as “*name*”.

- The machine description often uses string values. These are printed as “*value*”.

- Well known names are introduced by ‘*single*’ quotation marks. Further occurrences may not necessarily be emphasized as such.

- Compiler options are displayed in *slanted* shape.

- Filenames will be shown as ‘*somefile.c*’. Unless specified otherwise, all filenames are meant as being relative to the subdirectory ‘*gcc/c6x*’ in the extracted GCC source tree.

- Source Listings and Examples are typeset to emphasize keywords, strings and comments. Listings with more than one line will be displayed inside a box. Line numbers are only used, if the surrounding text uses them for reference.

Apart from the GCC Design Chapter I will use ‘GCC’ and ‘GNU C compiler’ as synonyms. Please note that GCC actually means ‘GNU compiler collection’.
Contents

List of Figures iii

List of Tables v

1 Introduction 1

2 Technical Features of the TI TMS320C6x 2

3 GCC Design 4

3.1 GCC Compiler Driver 4

3.2 GNU C Compiler 4

3.3 Hardware Independent Transformations 6

3.3.1 GENERIC 6

3.3.2 GIMPLE 6

3.3.3 Single Static Assignment (SSA) Form 7

3.3.4 Optimization Passes 8

3.4 Hardware Dependent Transformations 12

3.4.1 RTL Generation 12

3.4.2 Optimization Passes 13

3.5 Planned GCC Optimizations 16

3.5.1 Software Pipelining 16

3.5.2 Program Level Optimization 17

4 Back End Optimizations 19

4.1 Breaking down RTL 19

4.1.1 Splitting Moves 19

4.1.2 Splitting Additions 21

4.1.3 Expanding vs. Splitting 23

4.2 Using Attributes 24

4.3 Conditional Execution 26

4.4 Improving the Scheduler 27

4.4.1 Automaton Description 28

4.4.2 Adjusting Dependencies 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.3 Exploiting Parallelism</td>
<td>31</td>
</tr>
<tr>
<td>4.4.4 Delayed Branch Scheduling</td>
<td>32</td>
</tr>
<tr>
<td>4.4.5 NOP Insertion</td>
<td>34</td>
</tr>
<tr>
<td>4.5 Register Allocation</td>
<td>35</td>
</tr>
<tr>
<td>4.6 Additional Addressing Modes</td>
<td>37</td>
</tr>
<tr>
<td>4.7 Additional Machine Modes</td>
<td>38</td>
</tr>
<tr>
<td>4.7.1 64 bit Instructions</td>
<td>39</td>
</tr>
<tr>
<td>4.7.2 Load/Store Multiple</td>
<td>40</td>
</tr>
<tr>
<td>4.8 Maintainability Improvements</td>
<td>42</td>
</tr>
<tr>
<td>4.9 Low Level Instructions</td>
<td>43</td>
</tr>
<tr>
<td>4.10 High Level Instructions</td>
<td>45</td>
</tr>
<tr>
<td>4.11 Branch Handling</td>
<td>47</td>
</tr>
<tr>
<td>4.11.1 Calls</td>
<td>47</td>
</tr>
<tr>
<td>4.11.2 Conditional Branches</td>
<td>48</td>
</tr>
<tr>
<td>5 Performance</td>
<td>50</td>
</tr>
<tr>
<td>5.1 Exemplary Performance Improvements</td>
<td>50</td>
</tr>
<tr>
<td>5.1.1 Avoiding NOPs</td>
<td>50</td>
</tr>
<tr>
<td>5.1.2 Filling Branch Delay Slots</td>
<td>51</td>
</tr>
<tr>
<td>5.1.3 Conditional Instructions</td>
<td>52</td>
</tr>
<tr>
<td>5.1.4 Parallel Instructions</td>
<td>52</td>
</tr>
<tr>
<td>5.2 Benchmarks</td>
<td>53</td>
</tr>
<tr>
<td>5.2.1 Benchmarking Environment</td>
<td>53</td>
</tr>
<tr>
<td>5.2.2 Results</td>
<td>54</td>
</tr>
<tr>
<td>5.2.3 Conclusions</td>
<td>54</td>
</tr>
<tr>
<td>5.3 Conceptional Limitations</td>
<td>56</td>
</tr>
<tr>
<td>6 Further Improvements</td>
<td>58</td>
</tr>
<tr>
<td>References</td>
<td>63</td>
</tr>
<tr>
<td>Index</td>
<td>65</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Datapath of the TIC6x chipset family</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>GCC Toolchain</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>GCC Compiling Stages</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Data Flow without SSA</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Data Flow with SSA</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Software-Pipelined Loop</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>High-Level Compiler Architecture for Whole-Program Optimization</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>Extract and Zero Extend a Bit Filed</td>
<td>43</td>
</tr>
<tr>
<td>9</td>
<td>Basic Blocks</td>
<td>48</td>
</tr>
<tr>
<td>10</td>
<td>Corrupt Branches</td>
<td>48</td>
</tr>
</tbody>
</table>
List of Tables

1 Attributes .................................................. 25
2 Implemented Compiler Patterns (‘standard names’) ............... 44
3 Comparing Results ............................................ 54
1 Introduction

In recent years Digital Signal Processors (DSP’s) have become more and more widely used for taking over special computational functions of the main microcontrollers. They are now ubiquitous used in audio and video processing. Graphics hardware can be found in virtually any personal computer.

Even more densely packed processing power is needed in upcoming mobile environments. Mobile phones already feature video conferencing and the display of downloaded movies. Highly optimized software is needed to run on these new appliances. Using an optimizing compiler makes this an easier task for the developer. Assembly code is still necessary for some time critical purposes, but it decreases maintainability and portability of the resulting software.

Historically, the GCC suite was not designed to produce code for those platforms. Since recently, the market for those chips becomes more important, the compiler development clearly goes in this direction, too. Code size suddenly becomes important. New hardware features need compiler support.

During my work on the TMS320C6x port for the GNU assembler Jan Parthey developed a GCC back end for the same architecture. Having focused on the fundamental framework and because of the limited time frame of his diploma thesis, the code it produced did not make use of the architectural advantages of the TMS320C6x. His work discusses the principles of developing a functional back end for the GNU compiler collection. For an introduction into the compiler basics I recommend to read his work first.

My aim was to extend the existing GCC port to make it produce faster code. Section 2 will briefly discuss the features of the TMS320C6x architecture which I used to do that. Before implementing new algorithms to the GCC suite I wanted to examine which passes are particularly useful for improving performance on the given hardware. Section 3 describes the architecture of the GNU compiler collection and the contained C compiler itself.

The major part of this work was the upgrade and partial redesign of the existing back end. The most work was done to the machine description. The target macros could be left untouched, except for a few. Those, and the changes to other back end parts are explained in section 4.

Optimizing should make the software faster. This must be measurable. The list of the GCC optimization passes looks quite impressive, but can they compete with the TI compiler which produces remarkably good code? Section 5 will answer these questions.

There is still a lot of room for further improvements. Section 6 will give advice for further development.
2 Technical Features of the TI TMS320C6x

The TMS320C6x is a Digital Signal Processor (DSP) which is designed for a high instruction throughput. It features 8 pipelines to execute instructions fully parallel.

To reduce the complexity of its hardware design it is clustered into two separate register files, each being attached to four functional units.

![Datapath of the TIC6x chipset family](image)

Each functional unit primarily operates only on its attached register file. Within one processor cycle only one unit from each register file may read only one datum from the opposite. Furthermore, the C64x processor is capable of sharing this read datum among all functional units of the register file. The load/store units may read or write one datum each cycle, but not in the same register file. Due to these constraints, not every register can freely be used in any instruction. Meeting these constraints is important for the compiler to produce valid assembly code.

```assembly
sub b15, a15, b15
```

The destination register lies in register file B, the second operand therefore has to be read from the other register file via a ‘cross path’. Unfortunately the sub instruction is only able to read its first operand via a cross path. This instruction is invalid. As subtractions are not commutative, register a15 has to be moved to the other register file first. However, this is often unnecessary as the compiler chooses the registers wisely.

The chips are capable of executing any instruction conditionally. This means that the instruction is only executed if the specified condition register has a desired value, zero or non-zero. Using this, if-then-else statements with only a few conditional instructions can be encoded without using costly branch statements.

```assembly
cmpeq b9, b8, b0
[ b0] add a3, a4, a4
[!b0] sub a4, a3, a4
```
Finally, I wanted to take advantage of its ability of the data units to manipulate the address register before or after using its address value. If the compiler knows that the architecture supports autoincrement/autodecrement addressing, it can combine a load and a subsequent address register modification to a single instruction. The example shows the old and the new implementation of a push instruction.

\[
\begin{align*}
\text{ldw} & \quad *b15, \quad a1 \\
\text{add} & \quad -4, \quad b15, \quad b15 \\
\end{align*}
\rightarrow
\begin{align*}
\text{ldw} & \quad *b15--, \quad a1 \\
\end{align*}
\]

Yet, the most difficult task for the compiler is another one. All desktop processors manage their pipeline stalls in hardware. The TMS320C6x do not. A pipeline stall may occur, if one instruction has to use the result of another instruction, but this result is not yet computed. If there are enough other instructions that can be executed during that time, the pipeline does not have to be stalled. If there are none that are ready to be executed, NOPs have to be inserted in order to delay the execution of subsequent instructions. Luckily, there are only a few instructions that need more than one cycle: multiplications (two cycles), loads (five cycles), and branches (six cycles). Stores do actually need four cycles, but a load following a store to the same memory address will read the stored value. Therefore, stores can be regarded as completing within one cycle. Unluckily these instructions appear quite frequently in every program. There are different solutions to this problem depending on which optimizations are run during compilation. They are discussed in section 5.4.5.

A more exhaustive coverage of this architecture is provided by Texas Instruments. The architecture is described in the technical brief [7] and the complete instruction set can be found in [8].
3 GCC Design

This section is about the internal structure of the compiler itself. It should help to understand which optimizations take place in which part of the compilation process and which intermediate code representation are used.

The previous paper dealing with the TMS320C6x back end was implemented on the most recent version available then: 3.3. I started my work with upgrading the source to version 3.4 and later to 3.5. That version was renamed to GCC 4.0 in September 2004, due to the many substantial changes. It is still in development at the time of this writing.

3.1 GCC Compiler Driver

The ‘GCC tool chain’ is a set of tools that is usually required to translate high level source files into binary code, directly runnable on the target machine. These tools are controlled by the so called ‘compiler driver’. Figure 2 shows the usual control flow.

![Figure 2: GCC Toolchain](image)

3.2 GNU C Compiler

Optimizing the GCC for a particular hardware basically means optimizing the compiler. The other parts also perform optimizations to some extent, but that is usually independent from the target architecture and therefore out of the scope of this paper. Although the assembler normally only translates from assembly into binary it might optimize the code if provided with the necessary information. The so called ‘compiler proper’ of the GCC suite is itself parted into three main parts.
3.2 GNU C Compiler

Figure 3: GCC Compiling Stages

*Front End*

During the parsing of the source file the compiler front end reads the input language and performs lexical and syntactical analysis on the code. The front end represents the code using any desired intermediate language, only bounded in that it must later translate it into a language-independent representation, which the middle end of the compiler understands. Up until version 3.4 the middle end operated on ‘Register Transfer Language’ (RTL), which is an architecture independent, lisp-like assembly language. The new middle end speaks ‘GIMPLE’. I will explain the terms later in this section.

The translation should result in ‘GENERIC’ which is later lowered to GIMPLE by a compiler pass. GIMPLE is a subset of GENERIC which has additional constraints to speed up optimizations.

*Middle End*

The middle end performs different **hardware independent optimizations** on the internal representation. Actually, some passes need at least little information about the underlying architecture to achieve good results.
Back End

The back end performs **hardware dependent optimizations** and finally **generates assembly language**. It has the highest potential to improve code performance, as it can benefit from architectural features, but also has to cope with possible limitations of the actual hardware. As section 2 pointed out, the functional units of the TMS320C6x processors have a somewhat limited access to all registers. Hence, the compiler can not allocate hardware registers at will.

### 3.3 Hardware Independent Transformations

All current GCC releases still perform all optimizations on the RTL level. Most of them optimize the code only at a quite local level (within basic blocks). Version 3.0 was the first to support global RTL optimization passes. Unfortunately, RTL is too close to the machine language to be a suitable and effective language for more powerful optimizations.

The Tree SSA project [4] developed a completely new machine-independent optimization framework based on the ‘Static Single Assignment’ (SSA) form [14] [15]. This is an intermediate representation that makes data flow analysis much simpler as it largely reduces dependencies. This allows efficient implementations of the optimizing algorithms. To operate with SSA two more IR’s were introduced: ‘GENERIC’ and ‘GIMPLE’.

#### 3.3.1 GENERIC

It would be nearly impossible to write optimizing passes for every supported front end if there was no source language independent form of storing the program data. GENERIC is such a representation that was developed to store entire functions in trees. It is created after the front end has finished the parsing of a particular compilation unit. No optimization passes currently operate on it, but this might change in future releases. Hence, the front end is recommended to emit GENERIC, which usually is the less complex case. However, the C front end for instance does not. It emits GIMPLE.

#### 3.3.2 GIMPLE

All new platform independent optimization passes run on GIMPLE, which is a stricter form of GENERIC with less statements, highly influenced by SIMPLE IL [6]. The name is in fact a mixture of the names GENERIC and SIMPLE. Some other words like ‘gimplify’ and ‘gimplification’, which are used within the GCC community, derive from that. They all describe the transition from GENERIC to GIMPLE.
This transition is done automatically and nearly language independent. The front end only has to provide a definition of `LANG_HOOKS_GIMPLIFY_EXPR` to handle any language-dependent tree codes besides `GENERIC`. The example shows the conversion of a simple C function. This SSA code is a snapshot of the internal representation after the common subexpression elimination pass had removed an additional temporary.

```
int times (int a, int b)
{
    return a? a*2 : b*3;
}
```

GIMPLE statements have the following properties:

- GIMPLE expressions may contain no more than three operands with the exception of function calls. Complex expressions are broken down with the help of temporaries. Load and store operations are separated.
- There are no control flow structures. All loops, branches etc. appear as simple gotos.
- Expressions with side effects are allowed only as rvalue. A `*p++ = i` would be split into the assignment `*p = i` and the increment `p = p + 1`.
- Any function is completely represented as a tree.

### 3.3.3 Single Static Assignment (SSA) Form

The SSA form is even more strict than GIMPLE. It additionally assures that any program variable is assigned a value exactly once. Thus, for any use of a variable there is precisely one definition. Usually programs are not written that way. Accordingly, this form has to be created inside the compiler. It does so by creating a new version of a variable each time it is assigned a new value. The different versions are distinguished by subscription (a version number). The figures 4 and 5 show the conversion and illustrate the significant decrease of dependency complexity.

The optimization passes that run on this representation can now quickly determine the place
of the definition for any variable that is read in an expression and do a better job. They only have
to assure that if one pass invalidates the SSA form it has to be recreated before the next pass
starts.

During the creation of SSA it often happens that it is unable to tell which incarnation of
the variable is used in a specific expression. This occurs every time a variable is set within a
conditional block with a non-constant condition.

\[
\text{switch (c1)} \\
\{ \\
\text{case 1: } x = 1; \\
\text{case 2: } x = 2; \\
\text{case 3: } x = 3; \\
\} \\
\]

\[
\text{switch (c2)} \\
\{ \\
\text{case 1: } y[1] = x; \\
\text{case 2: } y[2] = x; \\
\text{case 3: } y[3] = x; \\
\text{case 4: } y[4] = x; \\
\}
\]

Figure 4: Data Flow without SSA

Figure 5: Data Flow with SSA

The value of \( x \) depends on the conditions of the switch statement. The ‘phi expression’ in the
SSA version states that the value of \( x_4 \) is either \( x_1, x_2 \) or \( x_3 \). The exact value is unknown
to the compiler unless for instance some optimization pass would recognize \( c1 \) to be constant.
All \( \phi \) expressions will be resolved when converting SSA back to GENERIC.

### 3.3.4 Optimization Passes

This is a list of all platform independent passes which are involved in optimization. I have left
out some passes that have other purposes. A summary of all passes is available in [3 Section 7].
I will start with all passes that work directly on the GIMPLE form. The second list contains the
optimizations that **run on the SSA form**.

**Remove Useless Statements**

Unreachable code is removed. If-statements with a constant condition expression, or com-
mands following a **return** are common examples.
Lower Control Flow
If-then-else statements are rewritten into a form in which both the if and the else part of the condition only consist of a single goto.

Lower Exception Handling Control Flow
Exception handling The exception handling statements (try, catch, finally) are transformed into less abstract statements representing the true control flow.

Build CFG
Builds the Control Flow Graph. Functions are decomposed into ‘basic blocks’ and connecting edges. Various later passes rely on this information.

Find Referenced Variables
This pass finds all referenced variables in every function and stores this data for later use by the SSA rewriting routines.

Enter SSA
The code is rewritten into SSA form (section 3.3.3).

The next passes all work with the SSA form. Some of them only run once, some of them more often during the entire optimization process. These usually simplify the code before other passes continue.

Dead Code Elimination
DCE scans the code for statements that have no side effects and whose result is not used. These statements are deleted, as they have no purpose. Any value that is stored in memory is considered used, alias analysis will deal with those.

Dominator Optimization
Node A dominates Node B if every control flow to B passes A. Using this knowledge it is possible to propagate constants and copies from B to A. They can then be used by other nodes as well. Page 11 shows a the result of a similar optimization.

Redundant Phi Elimination
This pass simplifies SSA. Every occurrence of a $\phi(a_1, a_1, ..., a_1)$ is removed.

Phi Optimizations
If it is possible to represent a $\phi$ expression as straight line code, this pass does so.

Forward Propagation of Single-Use Variables
Variables which are only used once are merged with the expression they are used in. Then, the algorithm attempts to simplify that expression.

Copy Renaming
This pass tries to remove copy operations by renaming temporaries.
May-Alias Optimization

This pass performs flow sensitive ‘alias analysis’ to detect which memory locations can be reached by more than one pointer. These areas are said to alias. Pointers that point to aliasing structures have to be handled with care because assigning a value to one pointer might change the value of the other.

The pass then tries to rewrite addressable objects into non-aliased variables that can be renamed into SSA form. It also updates internal data.

Scalar Replacement of Aggregates

Suitable non-aliasing aggregate variables are translated into scalar SSA variables. This improves subsequent optimization passes.

Dead Store Removal

If two stores save a value to the same memory location and no load reads that value, the first store can safely be deleted.

Profiling

The code is rewritten so as to collect profiling data. This provides optimization feedback for a later run. The execution frequencies of branches and calls for instance can be used to improve loop optimization and inlining.

Lower Complex Arithmetic

Complex arithmetic expressions are broken down to scalar arithmetic operations. $y=x^2$ to $y=x*x$ and the like.

Tail Recursion Elimination

Transforms tail recursion into loops. Tail recursion is a type of recursion where a function calls itself only from its end. This behavior can always be converted into a loop.

Partial Redundancy Elimination

Detects expressions that are computed multiple times, stores them into variables and reuses these for every subsequent occurrence of that expression. It also moves loads out of loops.

If-Conversion for Vectorizer

Tries to rewrite the code into a form that suits present vector operations. ‘Loop Peeling’ is a technique that reduces dependencies by extracting early iterations from a loop, as to allow parallelization of the remaining loop.
This example is taken from [17] which discusses the circumstances of when and how often to peel loops.

**Loop Optimization**
Moves loop invariant statements and conditions out of the loop. An optional strength-reduction rewrites expressions to use results of previous iterations which is more efficient.

```
j = 0; k = 0;
for (i = 0; i < 10; i++)
{
    j = i * 4 + 11;
    k = i * 6 + 9;
    a[j] = k;
}
```

```
j = 0; k = 0;
for (i = 0; i < 10; i++)
{
    j += 4;
    k += 6;
    a[j] = k;
}
```

It furthermore performs induction variable optimizations. It eliminates additional induction variables or transforms an index variable into a pointer.

It may also ‘unroll’, or ‘unswitch’ a loop. Unrolling substitutes the loop by the sequence of its iterations, to allow optimizations across cycles. Its also possible to only unroll the loop two or four times and divide the cycle count by that value. Unswitching will move conditional statements out of the loop, duplicating the loops body.

**Conditional Constant Propagation**
Finds variables that are constant, even in the presence of conditional branches and substitute them by their constant value.

**Folding Built-In Functions**
Simplifies built-in functions, where possible. String lengths from constant strings can be computed in the compiler, constant arguments can be used to calculate certain expressions in advance.

**Control Dependent Dead Code Elimination**
Eliminates unnecessary control flow statements, such as jumps to an immediately following label, jumps to jumps, jumps across jumps, etc.

**Tail Call Elimination**
Identifies function calls that may be rewritten into jumps. It does not actually transform the code, it just indicates the possibility. The target has to provide `fixup_tail_calls` to support that pass.
3.4 Hardware Dependent Transformations

All hardware dependent passes operate on the ‘Register Transfer Language’. RTL is generated after the ‘pass manager’ has run all target independent optimization passes on GIMPLE. Currently, there are still many platform independent RTL optimization passes present in the compiler. This will change in the future when more and more passes will be converted to SSA form and inserted into the new middle end architecture. Nevertheless, RTL will still remain the appropriate language for all back end optimizations.

3.4.1 RTL Generation

The ‘machine description’, consists of several definitions that deal with RTL patterns. Interestingly, they are RTL patterns themselves. These definitions fulfill the following tasks:

- **Instruction Definitions** with a ‘standard name’ tell the compiler which operations the hardware supports (move, add etc.) and for which ‘machine modes’ they are available. The machine mode describes how many bits are used for the instruction. A list of all standard names (all operations the compiler is using) and of the machine modes is available at [3, Chapter 12.9 and 10.6]. These patterns emit the initial RTL to represent the corresponding operation (the move or the addition).

```
(set (reg:SI 70) (mem:SI (reg:SI 71)))
```

This would be a suitable RTL for a load operation. The source is a memory location of which register 71 holds the address. The value is stored into register 70.

- **Expander Definitions** always have a standard name and can be used to emit multiple RTL patterns for one compiler operation. They are usually used to force some operands into registers before emitting an operation which does not support another type of operand in that position. They can either emit static RTL or dynamically generate RTL using emit_insn. Invoking DONE thereafter, the defined static RTL will not be emitted. Invoking FAIL will make the compiler consider this pattern as non-existent.

- **Split definitions** make the compiler split one RTL pattern into multiple others. In contrast to expander definitions, which are only used for initial RTL generation, splits can occur at several later stages.

- **Peephole definitions** do the inverse. They allow the replacing of several instructions by some (usually less) others.

- As the last action, **Instruction Definitions** emit assembly code for their corresponding RTL pattern. This applies also to definitions without a name. The compiler ignores any name not contained in the ‘standard names’.
3.4.2 Optimization Passes

The list below only contains RTL passes dealing with optimization. The list in the GCC internals [Chapter 7.5] contains some more. Passes which I consider redundant due to above SSA passes are not listed. However, they might as well be contained in upcoming releases.

Exception Handling Landing Pads
Generates code to communicate with the exception handling library routines. The ‘landing pads’ are the entry points in the function that are invoked by the library.

Common Subexpression Elimination (CSE)
Changes addressing modes to reduce cost and removes redundant computation within basic blocks.

Global Common Subexpression Elimination (GCSE)
Performs CSE across basic blocks, load/store motion and global constant/copy propagation. Code hoisting (unification) eliminates expressions that are computed in multiple code paths of the CFG to reduce code size but not speed. This is entirely different from CSE which removes multiple identical computations within the same code path.

Jump Bypassing
This aggressive form of GCSE propagates constants into conditional branches.

If-Conversion
Tries to convert if-then-else statements with conditional branches into comparisons producing boolean values and conditional move instructions. Finally, it will generate predicated operations if the hardware supports them. This also enlarges the basic blocks which allows for further optimizations.

Web Construction
Splits independent uses of pseudo registers for better register allocation.

Loop Optimization
Performs the same tasks as the platform independent pass, but on RTL level. This is necessary, because the RTL generation may have created new operation patterns.

```
loop:
...
mvkl 0x1234, a4
add   a3, a4, a3
...
```

Large immediates need to be moved to registers to circumvent the limitations of the `add` instructions. These constants should be moved out of the loop.
Life Analysis
Computes data dependencies and deletes computation of results that are never used. It further combines memory references and address modifications to create autoincrement or autodecrement addressing.

Register Movement
If an operation needs one of its operands to be reloaded into a register, a register-to-register move would be necessary. This pass tries to avoid these moves by renaming the affected registers.

Optimize Mode Switching
Some operations may need the processor to be in a specific mode. This pass finds those and minimizes the number of necessary mode changes.

Instruction Scheduling
Reorganizes RTL patterns within basic blocks to minimize pipeline stalls. It does so by separating the use of a value from its definition, but respecting all dependencies. The second scheduling pass can optionally perform the packing of parallel instructions. This is the major improvement done by this work. I will discuss it in detail in section 4.4.

Register Allocation
Up until now, every register was a pseudo register. To reduce complexity the amount of these pseudo registers is not limited. Finally, all used register must end up in one of the hardware registers of the target platform. This pass reduces the number of used registers to meet the requirements of the architecture. Several sub passes are executed to perform this task.

Register Class Preferencing
A chip may have different register classes. The TMS320C6x all have two separate register files, each of which should be pertained in a separate class. The current back end implementation uses even more classes to reflect certain other constraints. This pass finds the preferred register class for each pseudo register. The preferred class is defined by the instruction definitions.

Register Allocation
There are two different algorithms available. The original register allocator of early GCC versions and a relatively new one, which is only available in recent releases.

Two Phase Register Allocation
The ‘Local Register Allocation’ assigns hard registers within basic blocks.
The ‘Global Register Allocation’ allocates registers for pseudo registers with a life spans larger than a basic block.
Graph Coloring Register Allocation

This register allocator creates a graph representation with a node for each pseudo register and an edge between any two pseudo registers which carry a value at the same time. After the initial setup the algorithm tries to color the graph with as many colors as there are hard registers available. This NP hard problem is solved by a polynomial time heuristic. A comprehensive discussion of graph coloring is contained in [2] and [1, Chapter 9.8]. The implemented algorithm itself is documented in [13].

Reloading

Any pseudo register that could not be assigned to a hard register is put into stack slots. Some operations may have been invalidated because an operand ended up in the wrong register class or the stack. The reloading pass repairs those by loading the corresponding values temporarily to registers, or by prepending copy instructions. Further, it can optionally eliminate the frame pointer and insert statements which save and restore call-clobbered registers around function calls.

Basic Block Reordering

Reorders basic blocks to avoid branch instructions. It can use statistical data about execution frequencies or branching probability, but it may also make profit of accessible profiling information.

Variable Tracking

Computes places of variables and stores it in RTL code so the debugger later knows which variable is stored in which register at a given time.

Branch Shortening

This is a tribute to RISC machines with a limited range for branching. The compiler uses information about the code length to select the correct statements. The branch instruction of the TMS320C6x has a range of $2^{21}$ bytes. If this is not enough, the compiler moves the target label into a register first and emits a jump relative to that register.

Delayed Branch Scheduling

Finds statements to put in delay slots of branches. Although it could theoretically do this for any instruction, it is usually only confronted with branches. This reordering would be quite beneficial for the TMS320C6x family as its branch instruction has five delay slots. Paired wit parallelism up to 40 instruction could be placed there. Unfortunately, its usage had to be constrained for complexity reasons, described in section 4.4.4.

Register-to-Stack Conversion

Makes use of a register stack. This is currently only supported for the 80387 MPU.
Assembler Output (Final)

Outputs assembly code for the compilation unit. The RTL patterns are matched against the define_insn patterns of the machine description. All operands are printed via the PRINT_OPERAND and PRINT_OPERAND_ADDRESS target macros.

Debugging Output

This last pass outputs any debugging information.

3.5 Planned GCC Optimizations

This section will discuss some advanced optimization techniques which the compiling tools of Texas Instruments use. The Development of the GCC suite never stands still. Two extremely powerful optimization should be discussed here. Software Pipelining is already part of the GCC ‘mainline’. However, I was not able to make it produce valuable results. The ‘File Level Optimization’ was proposed at the GCC Summit in 2003, but does not seem to be implemented by now.

3.5.1 Software Pipelining

Loops are a very integral part of programs: “80% of execution time is spent in 20% of the code.” These are usual the loop bodies. Speeding up loops is one the most efficient optimization strategies. Many algorithms (Invariable Code Motion, Loop Unrolling, etc.) work in this area.

Software Pipelining is a powerful approach to speed up loops by parallelizing them. Only independent code can be put into parallel blocks which is the key hit point in most parts of the program. They are too dependent to parallelize them efficiently. In loops usually only the data within one iteration depend on each other. Software Pipelining schedules a loop in a way as to execute multiple iterations in parallel. It does so by overlapping instructions from different iterations.

Figure 6 illustrates such a software pipelined loop. The letters A, B, C, and D represent the stages of the loop. The ‘loop kernel’ (the shaded area) consists of the loop stages that can be run at one time. The sequence before the kernel is known as ‘pipelined loop prologue’ and the area below it as the ‘pipelined loop epilogue’. In this figure, a maximum of four instructions can be executed in parallel.

‘Modulo Scheduling’ is a technique to construct software pipelined loops which aims at minimizing the loop cycle count to increase performance. It first estimates the ‘Initiation Interval’ (II), which is the length of the loop kernel. This length depends on hardware constraints and code dependencies. It then tries to schedule the loop in II cycles.
A1
B1 A2
C1 B2 A3
D1 C2 B3 A4
D2 C3 B4
D3 C4
D4

Loop Prologue
Kernel
Loop Epilogue

Figure 6: Software-Pipelined Loop

the algorithm tries with larger II until it succeeds.

The loop prologue and epilogue are generated to keep the code correct. They initialize the data to secure that all different instances of the loop can operate properly. In some cases it is feasible to let the instructions inside the loop initialize themselves. This does not improve the speed, but reduces the code size. In case the loop bound (the number of iterations) is not known, a copy of the original loop is kept to deal with loops and parts of loops which do not fit the pipelined version.

This Optimization technique is present in both compilers. Its use in the TI Compiler and the Linear Assembler is documented in the TI manuals [7, p. 76], [9, 5-29]. The GCC implementation of the algorithm in the GCC is still undergoing development and testing. It uses the DFA based automaton description to check for hardware resource constraints. The documentation of its principles can be found in the proceedings of the GCC summit 2004 [5]. According to this paper the algorithm should already be working (to a limited extent). However, my tests showed no detectable change in the schedule with this optimization enabled.

3.5.2 Program Level Optimization

A lot of optimizations operate on a local level. The instruction scheduler, for example, only reorders statements, if they appear in the same basic block. The TI compiler knows file level and even program level optimization. Only since version 3.4 the GCC supports ‘unit-at-a-time’ optimization. Additionally, there are considerations about ‘Whole-Program Optimization’.

‘Interprocedural Optimizations’ allows for better optimizations because more information about the different functions is available, if the complete compilation unit is present in memory. Local functions which are never called can be deleted. Knowledge of how often and in
which parts of the program a function will be called makes ‘Function Inlining’ much more effective. Moreover, it is possible to rearrange the parameters for local functions to be passed in registers instead of memory locations. It can be guaranteed that all calls use the same convention.

Program Level Optimization is even more powerful because it can access all vital parts of a program and could theoretically optimize anything. The TI optimizer, for instance, uses the alias information of the function calling code to better parallelize load/store instructions. It can make certain that the memory segments do not interfere.

There is, of course, a drawback with global optimization: compile time. The larger the program gets, and modern programs tend to become large and complex, the more time the algorithms need to perform all optimizations. Additionally, the complete program has to be parsed for each compilation run. This not only implies that the complete source code has to be present, the repetitive parsing takes time and consumes memory.

The ‘Architecture for a Next-Generation GCC’ \cite{[11][12]} therefore proposes not to optimize on the source language but on a ‘language independent, low-level, SSA-based, strongly typed’ assembly language. Figure 7 illustrates the compilation data flow.

![Figure 7: High-Level Compiler Architecture for Whole-Program Optimization](image-url)
4 Back End Optimizations

The last section showed what optimizations GCC is capable of. Before trying to invent new algorithms to better support VLIW architectures I wanted the back end to benefit from these passes as much as possible. The development and implementation of further algorithms should be delayed up to the point when addressing the GCC optimizations will need more effort, given an equivalent performance increase.

In my effort to enhance the existing optimizations, I first targeted the instruction scheduler. It should parallelize the instructions as much as the data dependencies allow. To let it do the best possible job, any RTL instruction had to emit exactly one assembly instruction. In any other case the resulting schedule would be far from perfect. It might even be invalid.

4.1 Breaking down RTL

When I started my work the generated RTL was far from atomic. For simplicity reasons several patterns emitted more than one assembly instruction. The “movsi” pattern is printed here as an example. If you are not familiar to RTL, please read the [16, Sections 2.3.9 and 3.2] for an introduction. The RTL reference can be obtained from the GCC website [3, Section 10]

4.1.1 Splitting Moves

```
1  (define_insn "movsi"
    [(set (match_operand:SI 0 "general_operand" "=i,r,m,r,r")
        (match_operand:SI 1 "general_operand" "r,m,r,M,i"))]
    ""
5   "@"
   mv $1, %0
  ldw $1, %0;nop 4
  stw $1, %0
 mvkl $1, %0
10  mvkl $1, %0;mvkh $1, %0"
[])
```

The problematic statements here are in line 7 and 10. They both define two separate assembly instructions to be emitted. At first I deleted the nop 4 instruction. NOP instructions are handled separately now. Section 4.4.5 will show how. Furthermore, I instructed the compiler to split the last instruction into two distinct RTL statements. This is done with the insertion of a defineInsn_and_split to handle the case of moving a label or a large immediate into a register.
This split produces two instructions: The `set_lo_sum` sets the lower part of the register and `set_high` writes to the upper half. Of course, these patterns have to be provided to output the correct assembly statements later.

```
(define_insn "set_high"
  [(set (match_operand:SI 0 "register_operand" "+a,b")
      (high:SI (match_operand:SI 1 "s17to32bit_operand" ")))
   (use (match_dup 0))]

"mvkh\t%$\t%1,\t%0%?"
  [(set_attr "type" "mvk")
   (set_attr "unitside" "a,b")])

(define_insn "set_lo_sum"
  [(set (match_operand:SI 0 "register_operand" "+a,b")
      (lo_sum:SI (match_operand:SI 1 "s17to32bit_operand" ")))
   ""
  "mvkl\t%$\t%1,\t%0%?"
  [(set_attr "type" "mvk")
   (set_attr "unitside" "a,b")])
```

The ‘`(use (match_dup 0))’’ in line 4 is absolutely necessary for correctness. If it was not present, the `set_lo_sum` would be optimized away. The compiler assumes that both instructions write to the same register and therefore the first assignment will be overwritten by the second. Thus, it concludes that the first instruction can safely be deleted. The ‘`(use (match_dup 0))’’ tells the compiler that the destination register is also read by this instruction. Originally I thought the ‘+’ in front of the constraints of the destination operand (line 2) would have that effect. It apparently had not. The first statement can now stay in the instruction stream, as it provides a source operand for the second.

The `set_attr` statements can be left aside for the moment. I will discuss them later in section 4.4 they are used to pass certain information to the scheduler.
4.1.2 Splitting Additions

The same problem arose with the "addsi3" pattern. However, the solution to it is somewhat different. A pattern existed to handle additions with the second operand being an immediate that would not fit into the 5 bit which the add instruction provides for an operand. This pattern was placed before the actual "addsi3", which could only handle the trivial cases. It emitted the necessary copy sequences and the actual addition:

```
(define_insn "*addsi3_prologue"
  [(set (match_operand:SI 0 "general_operand" "+r,r,r,r")
       (plus:SI (match_operand:SI 1 "general_operand" "+r,0,r,r")
               (match_operand:SI 2 "const_int_operand" "+L,M,M,i")))]
   ""
   "@
   add $2, $1, $0
   addk $2, $0
   mvkl $2, a14;add $1, a14, $0
   mvkl $2, a14;mvkh $2, a14;add $1, a14, $0"
  []
)
```

I have replaced it by a define_expand which provides more flexibility:

```
(define_expand "addsi3"
  [(set (match_operand:SI 0 "register_operand" "")
       (plus:SI (match_operand:SI 1 "register_operand" "")
               (match_operand:SI 2 "reg_or_ci_operand" "")))]
   ""
   
   { if (c6x_expand_addoperands(SImode))
     DONE
   }
)
```

Its sole purpose is to provide a frame for the C function c6x_expand_add. This function checks whether the operands are valid for a single add instruction and emits a copy statement if necessary. It uses the GCC internal force_reg function to force an operand into a register. c6x_expand_add always returns 0 to signal that the defined plus pattern should be emitted. This pattern would be ignored by invoking the DONE macro if the function returned nonzero. This might be adequate if the function itself would emit a plus pattern, but it currently does not.

Please notice that we do not have to care of how the value is moved to the register, because the move pattern will cope with that. If necessary the instruction might even be split again into a mvkl, mvkh sequence.
Last but not least there is `c6x_emit_add_const`. I have omitted the implementation details to show what the function does. Its definition can be found in ‘c6x.c’ among the other functions.

```c
void
c6x_emit_add_const(enum machine_mode mode, rtx dest, rtx src1,
                   int imm, bool fr)
{
  if (rtx_equal_p(dest, src1) && CONST_OK_FOR_M(imm))
    /* emit addk pattern */
  else if (CONST_OK_FOR_L(imm))
    /* emit normal add pattern */
  else {
    if (no_new_pseudos)
      /* use fixed scratch register because we cannot allocate a new one */
    else
      /* allocate new register */
      /* move immediate to register */
      /* add two registers */
  }
}
```

The function emits the appropriate RTL instructions to perform an addition of register `src1` with the immediate `imm` storing its result in register `dest`.

Even though the expander definition seemingly takes care of exactly the same issue, this function is still necessary as it serves another purpose. When the function prologue and epilogue need to adjust the stack and the frame pointer they might emit an addition that cannot be handled by a single `add` instruction. The compiler does assume that any expander definition, such as the prologue and epilogue patterns, only emits correct RTL instructions. The “addsi3” expander definition will not be called at that time. To circumvent that, the prologue and epilogue patterns use the above function to modify the stack pointers. `c6x_emit_add_const` cares about the actual output.

It will only emit one instruction, unless the immediate value is too big. The other cases need a temporary register. Unfortunately it may happen that there are no new pseudo registers available (no_new_pseudos is nonzero). After the reload pass mapped the pseudo registers to the actual hard registers no new pseudos can be allocated any more. Instead, it uses a14 or a14 as temporary storage depending on the register file of the destination operand. This is essential because the other source operand may lie in the other register file and an ‘add Areg, Areg, Breg’ or vice versa is not allowed on the TMS320C6x architecture.
4.1.3 Expanding vs. Splitting

The `define_insn_and_split` appears to be the easiest way to handle moves of large immediates. The split is automatically carried out during the optimization process. However, it proved to be the less profitable way. Expander definitions early emit all necessary statements and therefore allow better scheduling.

The first run of the scheduler tries to reorder instructions to make the life spans of temporary values (as needed in the case of “addsi3”) overlap each other. This is done, by moving away definitions of temporaries from their usage. If they overlap, different registers will be allocated to them and they can probably be scheduled in parallel, later. Having many instructions available at an early point allows for a higher degree of parallelism.

I have rewritten the previously defined “movsi” instruction definition to an expander pattern that performs all necessary transformations at the time of initial RTL generation. This little work had already significant positive effects. Branch slots are used more effectively (with activated ‘delayed branch scheduling’, section 4.4.4) and the degree of parallelism is remarkably higher.

\[
\begin{array}{ll}
| & \\
stw & a15, *--b15 \\
mvk & 100, b3 \\
mv & b15, a15 \\
add & -8, b15, b15 \\
sw & b3, *b15 \\
mvkl & 231072, b3 \\
mvkh & 231072, b3 \\
stw & b3, *+b15[1] \\
mvkl & L4, b3 \\
mvkh & L4, b3 \\
stw & b3, *--b15 \\
mvkl & fadd, b3 \\
mvkh & fadd, b3 \\
b & b3 \\
nop & 5 \\
\end{array}
\]

\[
\begin{array}{ll}
| & \\
stw & a15, *--b15 \\
mvkl & 231072, b3 \\
mvkh & 231072, b3 \\
add & -8, b15, b15 \\
mvkl & L4, b3 \\
mvkh & L4, b3 \\
stw & a3, *b15 \\
mvkl & fadd, b3 \\
mvkh & fadd, b3 \\
b & b3 \\
nop & 5 \\
\end{array}
\]  

9 + 5 = 14 cycles

7 + 5 = 12 cycles

In contrary to the left example the mvk, mvkl and mvkh in the right do operate on different registers. They appear earlier and do not appear to be grouped anymore. If register b8 would have been an A register instead, the schedule would even be another two cycles shorter. Consequently rewriting the back end to use expander patterns should make that possible. It is further necessary to eliminate all unnecessary split definitions (as the above movsi_big in the long run. Otherwise, the combiner pass will merge correctly expanded instructions to these patterns which will later have to be split again.
However, splitting patterns are necessary, if the instructions have to know which hard registers they use. These `define_insn_and_split` must define `reload_completed` as their splitting condition and must not allocate new pseudo registers. If scratch registers are necessary for storing temporaries, a `match_scratch` must be used in the corresponding expander definition. The fixed registers `SCRATCH_REGNO_A` and `SCRATCH_REGNO_B` can also serve this purpose. A necessary split is used to handle “movdi” later in section 4.7.1.

### 4.2 Using Attributes

Attributes are a major step towards a good scheduler. They are useful for attaching information to RTL instructions, which is later used to program the instruction scheduler. Furthermore, they allow the modularization of the design process. I will now give a short introduction into this topic and discuss how I used attributes.

There are two possible attribute types: strings and integers.

```lisp
(define_attr "datapath" "no,a,b" (const_string "no"))
```

This defines a new attribute `datapath` which may have three possible values: “no”, “a” or “b”. I use this attribute in the back end to memorize that the corresponding instruction (a load or a store) uses data path “a” or “b”. The default value “no” which is set with the last expression inside the `define_attr` is used for any instructions that does not set this attribute.

```lisp
(define_attr "needs_nops" ""
  (cond [(eq_attr "type" "bimm,bint,breg") (const_int 5)]
       [(eq_attr "type" "load") (const_int 4)]
       [(eq_attr "type" "mpy") (const_int 1)]
       (const_int 0)))
```

The second definition is a bit more complex, but it shows how powerful attributes are. The attribute `needs_nops` holds the number of delay slots for each particular instruction. Instead of having to write an `set_attr` for any instruction that does not have the default number (zero), I can simply define the value depending on other attributes which are already set appropriately.

In this case I made use of the `type` attribute. This attribute later tells the scheduler the type of the instruction it wants to schedule. Depending on this instruction type it will select the correct functional unit (FU) to use and makes sure that every FU is only used once in one processor cycle. The conditional expression defines branch instructions to have five, load instructions to have four, multiplications to have one and all other instructions to have zero delay slots. The type attribute has to be set for each instruction anyway, so the number of delay slots is also known. This attribute will become important later when I discuss the attempts to insert NOPs into the...
### 4.2 Using Attributes

A list of the all attributes I used, along with their possible values and their meaning can be found in table 1. The values of the type attribute are not completely listed. This attribute has almost as many values as there are instructions available on the TMS320C6x chips. Therefore, I have tried to group instructions with respect to their scheduling behavior. Other instructions had to be split into multiple attributes, because they need to be scheduled differently. They start with the name of the appropriate assembly instruction and end with the functional units they can be scheduled on. A `sub_ls` would refer to a subtraction that can be scheduled on an L or an S unit. Values of instructions identical to the assembly opcode are not listed here. A complete list can be obtained from [8, Chapter 3-24].

<table>
<thead>
<tr>
<th>attribute</th>
<th>values</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td></td>
<td>This is the instruction type. It is usually equal to the name of the assembly instruction. Its default value is “nop”, but that is only used by the “nop” instruction itself.</td>
</tr>
<tr>
<td>load</td>
<td></td>
<td>An instruction which loads a value from memory (<code>ldb(u)</code>, <code>ldh(u)</code>, <code>ldw</code>, <code>lddw</code>).</td>
</tr>
<tr>
<td>load2</td>
<td></td>
<td>An instruction which loads two subsequent memory locations into two adjacent registers.</td>
</tr>
<tr>
<td>store,store2</td>
<td></td>
<td>As above, but for store instructions.</td>
</tr>
<tr>
<td>mvk</td>
<td></td>
<td>This instruction moves an immediate into a register (<code>mvk</code>, <code>mvkh</code> or <code>mvkl</code>).</td>
</tr>
<tr>
<td>shift</td>
<td></td>
<td>A logical or arithmetical shift instruction (<code>shr</code>, <code>shru</code>, <code>shl</code> or <code>sshl</code>).</td>
</tr>
<tr>
<td>logic</td>
<td></td>
<td>A logical operation (<code>and</code>, <code>or</code>, <code>xor</code>, <code>not</code>).</td>
</tr>
<tr>
<td>breg,bimm,bint</td>
<td></td>
<td>A branch to register, label or interrupt register.</td>
</tr>
<tr>
<td>predicable</td>
<td>no, yes</td>
<td>This instruction can be executed conditionally.</td>
</tr>
<tr>
<td>datapath</td>
<td>no, a, b</td>
<td>The name of the datapath this instruction uses. Only loads and stores use a datapath.</td>
</tr>
<tr>
<td>cross path</td>
<td>no, x</td>
<td>This instruction uses a cross path. Which cross path is used depends on <code>unitside</code>.</td>
</tr>
<tr>
<td>unitside</td>
<td>a, b</td>
<td>The name of the cluster this instruction must be scheduled on. This is determined by its destination register, or by the base address register in case of a load or store.</td>
</tr>
<tr>
<td>needs_nops</td>
<td>0, 1, 4, 5</td>
<td>This amount of nops would have to be inserted after this instruction (also known as its delay slots).</td>
</tr>
</tbody>
</table>

Table 1: Attributes
4.3 Conditional Execution

The TMS320C6x support conditional execution for every instruction. This is special as the most common architectures only allow conditional jumps. Luckily the GCC already supports conditional execution, so I just had to enable the back end to use it. All the work is done by the following lines:

```
(define_attr "predicable" "no,yes" (const_string "yes"))

(define_cond_exec
  [(ne (match_operand:SI 0 "register_operand" "=r")
       (const_int 0))]
  ""
  "[ $0]")

(define_cond_exec
  [(eq (match_operand:SI 0 "register_operand" "=r")
       (const_int 0))]
  ""
  "![%0]"
)
```

The first line defines the attribute `predicable` and sets its default value to “yes”. Because all instructions on the TMS320C6x are conditionally executable, the default is valid for every instruction and needs not to be changed.

The `define_cond_exec` patterns define the two possible predicates. One for true (line 3) and one for false conditions (line 9). The pattern in line 3 is to be read as: ‘If register (operand 0) is not equal to zero execute the the corresponding instruction, otherwise execute a NOP.’

These predicates are now used by the compiler in the following way: For each instruction which has the attribute `predicate` set to “yes” and for each `define_cond_exec` one additional pattern will be generated. The following example will illustrate this behavior. The attributes are intentionally left out to improve clarity.

```
(define_insn "negsi2"
  [(set (match_operand:SI 0 "register_operand" "+a, a,b, b")
         (neg:SI (match_operand:SI 1 "register_operand" "+a,?b,b,?a")))
   ""
   "neg@if$a",\t%$?a"
  []]
```

The condition expression is prepended to the RTL expression of the pattern and the predicate string is added in front of each assembly output variant. With its `predicable` attribute set to “yes” by default, the above pattern will additionally generate:
4.4 Improving the Scheduler

This behavior is particularly useful with branch instructions. In fact I could not have split the branch pattern without using conditional execution. However, the compiler uses predicates for other purposes as well. They can help to avoid branches in if-then-else statements by executing the instructions with the condition result as their predicate. Transferred into C statements, the changes would look like these:

```
if (c) goto 11:
    y = x;
    goto 12:
11:
    x = y;
12:
...
```

As the avoidance of branches also results in larger basic blocks, there are more opportunities to parallelize instructions which in turn increases performance as well.

4.4 Improving the Scheduler

The back end has had no processor description. To be true, the compiler used a generic description, which, of course, made no use of the chips features (section 2). To use these special abilities I have provided a pipeline description.
4.4.1 Automaton Description

The GCC up to version 3.4 supported two different versions of processor descriptions. The older one was a description of the functional units, but had its limitations. A new ‘Automaton Description’ was developed to reduce the amount of additional code needed to work around those. The new description is strongly recommended\(^1\) – it was the more flexible choice anyway.

From that description a ‘Deterministic Finite Automaton’ (DFA) is generated during the compilation of the compiler back end, and this automaton incorporates the scheduler. The new pipeline description deals better with modern processors. Its ‘hazard recognizer’ detects pipeline hazards (stalls) based on the automaton description and allows to quickly test various schedules (of a limited number of instructions) to find the best one.

The C62x automaton description is a list of functional units, including the two data and cross paths. For each chip there is a separate automaton described in ‘c62.md’, ‘c64.md’ and ‘c67.md’ respectively. The back end does not yet use features of the C64x or the C67x that would require a specific pipeline description. Accordingly, ‘c64.md’ and ‘c67.md’ are merely copies of ‘c62.md’, only adapted to the appropriate namespace. Thus I will only handle the common behavior.

The next statements define all necessary functional units:

```plaintext
1  ;; actual reservation units
(define_cpu_unit "c62_D1,c62_L1,c62_M1,c62_S1, c62_D2,c62_L2,c62_M2,c62_S2" "tic62")
2  ;; two cross paths
5  (define_cpu_unit "c62_arb,c62_bra" "tic62")
3  ;; two data paths
(define_cpu_unit "c62_da,c62_db" "tic62")
```

The real ALUs are defined in line 1 and named in TI fashion\(^8\). The next two definitions create additional units to take care of the following constraints:

**Data Path**

Within one cycle each register file can only be accessed once with a load/store instruction. Any load/store instruction occupies one data path. The scheduler can only issue load/store instructions in parallel, if their respective source and target operands are registers from different register files.

**Cross Path**

Within one cycle only two values can be read via a cross path. One value from the A register file and one from the other. Any such instruction occupies the corresponding path and no other instruction can use it.

\(^1\)The old description is not documented in the GCC internals any longer.
4.4.1 Improving the Scheduler — Automaton Description

These are the hardware constraints that put the most pressure on the scheduler. However, \[8\] defines some more of them. There is for instance the maximum count of five parallel read accesses to one register. At the recent stage of optimizing ability the scheduler will quite unlikely produce execution packets that violate this condition, but in principle that may happen. This problem has to be resolved by further refining the automaton descriptions.

Yet another constraint exists for the C62x and the C67x. They cannot handle execute packets that cross fetch packet boundaries. The fetch stage of the pipeline always fetches eight instructions per cycle. It may happen that the first instruction of the new packet should be executed in parallel to the last of the previous packet. The left listing shows such a situation.

```
7:  mov a1, b1
8:  mov b1, a1
1:  add 4, a2, a3
```

Only the C64x can handle that properly. To eliminate this problem for the other chips I have adapted the assembler to fill the slots of the old fetch packet with NOPs. In the right listing the slots seven and eight of the first execute packet have been padded with NOPs to move the second execute packet to the next fetch packet. Since the assembler runs after the compiler, this prevents invalid code. On the other hand, if the compiler knew about this issue, the assembler would not be forced to delay instructions, which in turn would lead to faster code.

The relation between the instructions to schedule and the functional units is described with several `define_insn_reservation_unit` statements.

```
1 (define_insn_reservation "c62_load_aa" 5
   (and (eq_attr "cpu" "c62")
     (and (eq_attr "type" "load")
      (and (eq_attr "unitside" "a")
       (eq_attr "datapath" "a")))
   "c62_D1+c62_da")
```

The attribute `cpu` selects the chip CPU according to the `–mcpu` command line option. Other attributes control the assignment of functional units to the instructions to be scheduled. The `type` of an instruction and the `unitside` are criteria for the range of functional units which are able to execute the instruction. The `datapath` and `crosspath` attributes watch over the corresponding constraints.
The statement defines a reservation unit for a load instruction. It is only valid on the C62.
If the type of the instruction is “load” which is executed on cluster A and whose destination
register is accessed via data path A, this instruction occupies “D1” as well as the data path “da”
and can be scheduled in this cycle if both units are available.

4.4.2 Adjusting Dependencies

The define_insn_reservation defines an execution delay of five cycles. This value
describes after how many cycles the result of the instruction is available to other instructions.
This is the time it takes to get the memory value and copy it to the destination register. But the
load instruction may also modify the base register:

```
ldw *b15++, a2 ; pop a2
```

The increment is already completed within the same cycle. Subsequent instruction do not have to
wait five cycles. They can access the new value of b15 already in the next one. The automaton
description does not provide a mechanism to define such a behavior. Instead, there is a target
hook to handle these situations.

```c
#define TARGET_SCHED_ADJUST_COST c6x_adjust_cost

int c6x_adjust_cost (rtx insn, rtx link, rtx dep_insn, int cost)
{
    ...
}
```

The function receives the current instruction (insn), the kind of dependency (link), the
instruction the dependency occurs with (dep_insn) and the costs (cost) that have been deter-
mined so far. The default behavior of this function is to simply return cost.

Three possible kinds of dependencies can occur:

*data dependency*

One of the source operands of insn is set by dep_insn. The instruction has to wait until
its source operands are available. In this case it is checked whether this applies to the address
or the destination operand of dep_insn and the cost is set appropriately.

*anti dependency*

One of the source operands of dep_insn is overwritten by insn. The instruction has to wait until dep_insn has read its operands. The pipeline description defines all instructions
to be fully pipelined, meaning that every source operand has been read before any destination
operand is written. We do not have to change the rules for anti dependencies as the default
rules are correct.

\[2\] However, the other automaton descriptions define it in the same way.
output dependency

The destination operand of `dep_insn` is overwritten by `insn`. The instruction has to wait until `dep_insn` has finished writing. The calculation of output dependency costs is correct, too.

4.4.3 Exploiting Parallelism

To achieve the issue of multiple instructions in one cycle the scheduler must be informed that the architecture is capable of multiple issue. This is done by the following target hooks:

```c
#define TARGET_SCHED_ISSUE_RATE c6x_issue_rate
#define TARGET_SCHED_FIRST_CYCLE_MULTIPASS_SCHEDULING hook_int_void_1
#define TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD c6x_use_sched_lookahead
```

The first macro tells the scheduler how many instructions it can emit per clock cycle. The function `c6x_issue_rate` returns eight, if the target option `–mparallel` is set, and 1 else. This option can be set on the command line, but is activated, too, if `–O2` or larger is specified.

The second activates multi pass scheduling by returning a pointer to a function that will always return 1. Multi pass scheduling allows the scheduler to try several possible instructions for the next slot to find one which will result in the maximal possible number of issued instructions in that cycle.

The last controls how many instructions are considered for speculation. The current value is 8. Higher values will probably increase the compile time and not result in better schedules. However, it may seem useful to make that option available on the command line to allow some fine tuning.

The scheduler will now schedule instructions for parallel execution if `–mparallel` is set, but this does not yet effect the assembly output. The scheduler will only mark the execute packets by assigning machine mode TI (Tetra Integer) to each first instruction.

How can we turn this information into the parallel operator ‘||’? The assembly language specification only allows ‘||’ as the first characters on a new line. The regular target hooks for assembly output (`PRINT_OPCODE` and `PRINT_OPERAND`) are called after an initial tabulator has already been inserted. Hence, they cannot fulfill this task. `FINAL_PRESCAN_INSN` should intentionally allow the back end to check all operands before they are being output. It is called before the tabulator is output and is therefore perfectly suitable for printing the parallel operator in front of any instructions that does not have machine mode TI. These are exactly the instructions that are parallel to the first of the execute packet, as the example illustrates.
The first fetch packet contains both `add` and the `mv` instruction, the second consists of the branch and the `mvk`.

The other purpose of the function invoked by the macro is to delay the output of NOPs, if they follow one of the parallel instructions. The NOPs are inserted after all instructions of that execution packet were output. The NOP in line 7 belongs to the branch in line 5, but it is output after the `mvk` instruction.

### 4.4.4 Delayed Branch Scheduling

The ‘Delayed Branch Scheduler’ (DBS) searches for instructions to put in the delay slots of branch instructions. It could as well be used to fill other delay slots, but the instruction scheduler is usually the better choice to do that. Quite like the instruction scheduler, the DBS is instructed using attributes:

```
(define_attr "in_branch_slot1" "no,yes"
(if_then Else (eq_attr "type" "bimm,bint,breg")
(const_string "no")
(const_string "yes")))
```

This pattern defines the attribute `in_branch_slot1`. The machine description defines some further branch delay attributes:

<table>
<thead>
<tr>
<th>attribute</th>
<th>cycle</th>
<th>available for ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_branch_slot1</td>
<td>1</td>
<td>all instructions but branches</td>
</tr>
<tr>
<td>in_branch_slot2</td>
<td>2</td>
<td>all instructions but branches and loads</td>
</tr>
<tr>
<td>in_branch_slotn</td>
<td>3-5</td>
<td>all instructions but branches, loads and stores</td>
</tr>
</tbody>
</table>

The intention of this differentiation is not to allow branches in a delay slot of a branch, and to constrain load instructions (four delay cycles) to the first slot. This ensures that all instructions have been completed when the actual branch occurs six cycles after its issue. Furthermore, I am not convinced whether the GCC scheduler could handle the dependencies properly. This needs to be checked before changing this behavior.
The following lines define the actual delay slots using the above attributes to represent the constraints:

```
(define_delay (eq_attr "type" "bimm,bint,breg")
  [(eq_attr "in_branch_slot1" "yes") (nil) (nil)
   (eq_attr "in_branch_slot2" "yes") (nil) (nil)
   (eq_attr "in_branch_slotn" "yes") (nil) (nil)
   (eq_attr "in_branch_slotn" "yes") (nil) (nil))
```

All instructions which the type attribute classifies as a branch now have five delay slots, one for each expression. All instructions that match the condition of a particular delay slot can be scheduled in it. The two (nil) columns could be used to annul instructions that were scheduled due to false branch prediction. The TMS320C6x processors do not annul any instructions, so this is not necessary.

The delay branch scheduler would normally run as the last optimization pass. However, I became aware of some subtle problems concerning the coexistence of the instruction and the delayed branch scheduler.

The instruction scheduler gathers dependency information, schedules the instructions and groups them into packets for parallel execution. During the schedule is created, the back end records in which processor cycle each instruction is scheduled. This allows it to insert only as much NOPs as needed when outputting the final assembly code. The DBS reorganizes this schedule, partly destroying the gathered information.

For instance, it may move one instruction which was the only one scheduled for that cycle, the cycle information of the other instructions will stay untouched. Because of the moved instruction there is now a gap in the schedule that will later be filled by a NOP. This will not invalidate the schedule, but it will do no good either. The next listing illustrates this behavior. The numbers in front are the recorded cycle count. The `add` instruction is moved behind the branch leaving its cycle empty. A NOP fills that place in the resulting code (line 2).

```
1 : ldw  *++b15, a4
2 : add  a1, a2, a3
    nop  3
6 : b   a4
    nop  5
12: ...

1 : ldw  *++b15, a4
    nop  4
6 : b   a4
    add  a1, a2, a3
    nop  4
12: ...
```

The situation gets even worse, if the DBS moves an instruction out of an execute packet of several instructions. This may change the program’s semantics in some cases as the next example demonstrates. If the delay branch scheduler takes out one of the the two move instructions to put
it behind the branch, the code does not swap $a3$ and $a5$ anymore. It actually sets both registers to the value of $a3$.

\[
\begin{align*}
1: & \quad \text{\texttt{mv} } a3, a5 \\
| & \quad \text{\texttt{mv} } a5, a3 \\
2: & \quad \text{\texttt{b} } a4 \\
& \quad \text{\texttt{nop} } 5 \\
8: & \quad \ldots
\end{align*}
\]

Because of these severe problems, I have deactivated the delay branch scheduler in combination with the second pass of the instruction scheduler which does the VLIW packing. I used the target macro `OPTIMIZE_OPTIONS` to set `flag_delayed_branch` to zero.

### 4.4.5 NOP Insertion

The processors of the TMS320C6x family does not feature hardware pipeline stalls to satisfy dependencies. To ensure that the results of an instruction are available at the time of their use the programmer (here the compiler) has to insert useful instructions or NOPs if no suitable others are available.

To let the back end insert NOPs appropriately it has to know which instructions need them, and how many NOPs are to be inserted after them. To easily handle the different cases, the machine description defines an attribute to store the number of necessary NOPs for each instruction:

```c
#define_attr "needs_nops" "
  (cond [(eq_attr "type" "bimm,bint,breg") (const_int 5)]
        [(eq_attr "type" "load") (const_int 4)]
        [(eq_attr "type" "mpy") (const_int 1)]
        (const_int 0))
```

Branch instructions have five delay slots, loads four, and multiply instructions have a delay of one cycle. The others can be considered to complete in the cycle they were issued. The pipeline here somewhat hides the real complexity.

The NOPs must be inserted after an instruction which needs more than one cycle, or, in terms of the machine description, for which `get_attr_needs_nops` returns non-zero. Because every assembly output template (e.g. “neg\t%$\t%1,\t%0%?”) ends with a ‘%?’, `c6x_print_operand` is called after the assembly statement is. This function cares for the output of the necessary NOPs, depending on the selected optimizations. Some optimization passes provide special information that can be used to insert NOPs more cleverly. For completeness, I have listed the case of an activated delayed branch scheduling here, too. Of course, it will currently never occur, because DBS is deactivated due to the problems discussed earlier.
NOP Insertion Without Optimization

The amount of NOPs is read from the the *needs_nops* attribute and output directly after the instruction.

NOP Insertion and Delayed Branch Scheduling

At first it is checked if `dbr_sequence_length()` returns non-zero. If this is not the case, the actual instruction is not part of a delayed branch sequence and the NOPs are output directly after the instruction. In the other case the output of the NOPs is delayed. Their output will be handled later by `c6x_seqend` which is called by the target_hook `DBR_OUTPUT_SEQEND` after all instructions that were placed into the sequence have been output. `c6x_seqend` will, of course, only emit as many NOPs as necessary.

NOP Insertion and Parallel Scheduling

If the scheduler is run, it checks dependencies and tries to schedule the instructions as to minimize the processing cycles. The hooks `TARGET_SCHED_REORDER(2)` are considered to reorder the instructions in the ready cue in case the back end wants to override the default rules. I used the hooks to memorize the program cycle the instruction is scheduled. This value is stored in a hash table I created for that purpose. From the cycle information I can later determine how many NOPs have to be inserted between two instructions. This value then overrides the attribute value.

4.5 Register Allocation

Due to the clustering of the design (figure 1 on page 2) the functional units primarily operate on their attached register file. However, many instructions can access a source operand from the opposite register file. Such an access via a cross path is limited to two accesses per clock cycle and should therefore be avoided where possible. This would unnecessarily reduce the number of instructions that can be emitted in the same cycle. These considerations result in the following code changes:

All register operand constraints that used “r” now use “a” or “b” (in any combination that is possible for the individual instruction). These letters also represent register operands, but differentiate between the A and B register file. The constraint “r” is only used in the rare case, where the register file side does not matter. The letters “A” and “B” only match for registers that can be used as a predicate. The letters “Q” and “R” specify memory operands whose base register lies in the A or B register file respectively.

If the use of current constraint would result in the use of a cross path, the compiler is discouraged to use it, by prepending ‘?’ to the constraint letter. When calculating the register
preferences, the compiler regards such a constraint as more costly and will use it only, if the cheaper ones fail. I did not apply ‘?’ to load/store instructions as they always use one data path, not regarding any of the register files as more costly.

```c
(define_insn "negsi2"
  [(set (match_operand:SI 0 "register_operand"
         "=a, a,b, b")
      (neg:SI (match_operand:SI 1 "register_operand"
               "a, ?b, b, ?a")))]
  
  "neg\%\t\%1, \%0?"
  [(set_attr "type" "neg")
   (set_attr "crosspath" "no, x, no, x")
   (set_attr "unitside" "a, a, b, b")])
```

This pattern defines the negation of a register. The source like the destination register can be on either register file. The possible combinations of read accesses are:

- A reads A executed on cluster A
- A reads B executed on cluster A, but reads source operand from register file B
- B reads B executed on cluster B
- B reads A executed on cluster B, but reads source operand from register file A

The register allocator further needs to know in which order the registers should be assigned. The structure `REG_ALLOC_ORDER` in `c6x.h` defines that order. At first, it lists all registers that serve no special purpose. They are followed by the registers that support ‘circular addressing’, even though that is not used yet. At the end there are the condition registers and the fixed registers. A back end can mark a register as ‘fixed’. If so, the compiler will not use it for register allocation. Usually the stack pointer, the frame pointer and some scratch registers are marked as fixed. The scratch registers can then be used freely to work around certain limitations. Though the fixed registers will never be used by the compiler, the structure has to contain all registers.

Additionally to the above preferences, the registers from both are noted in an alternating way. This ensures that there are always registers from both files available. If the list started with the A registers, followed by the B registers, their would soon occur a situation in which all A registers are in use. Surely, the next instruction would need another A register, because certain constraints demand that. The compiler would stop at that point with an internal error. The register mixture should prevent that.

In the rare case that only one more hard register is available, it may still occur that it cannot be used because it belongs to the wrong register file. The corresponding `define_insn` will fail. I know of no practical solution to that. It is possible to use split definitions with the predicates
4.6 Additional Addressing Modes

Load and Store instructions each support the modifying of the base register which was used to calculate the memory address. This means that they can change the value of the base register before or after retrieving its value. This allows the compiler to group a memory access and the corresponding pointer changing instruction. The combination of these operations is especially common for push and pop operations. They can now be performed with only one instruction.

\[
\begin{align*}
\text{ldw} & \quad \text{a2} \\
\text{add} & \quad 4, \quad \text{b15, b15} \\
\end{align*}
\]
\[
\rightarrow
\begin{align*}
\text{ldw} & \quad \text{b15++, a2} \\
\end{align*}
\]

To achieve this I had to modify two functions which handle all addressing modes. They now do allow pre/postincrement, pre/postdecrement and pre/postmodify, too.

\textit{c6x_legitimate_address_p}

This function checks the validity of an address and returns nonzero if the passed parameter is a valid address for the TMS320C6x. An address is valid if one of the following conditions apply:

- The address is a signed immediate or a label.
- The address is stored in a register.
- The address is computed of a base register and a 5 bit signed immediate.
- The address is stored in a register and incremented or decremented by one before or after the instruction.
- The address is computed of a register and an offset and incremented or decremented by the offset before or after the instruction.

I have not allowed addressing with a base and an index register, because I am not sure how to tell the compiler that the processor will use the index value as a relative address. It must not be scaled corresponding to the machine type. It might as well be the case that the compiler already does that correctly, but this would require some intense testing. However, this is implemented, too. It only has to be activated.

\textit{c6x_print_operand_address}

This function is responsible for printing an address operand. I changed it to support the new addressing modes, too. Furthermore, it will now convert braces from absolute ‘( )’
BACK END OPTIMIZATIONS

offsets to relative offsets ‘[ ]’ where applicable. Unfortunately, it is not possible to use larger offsets. Theoretically the 5 bit number could as well be double words which would increase the ‘offsettable’ range. However, the function cannot access the mode of the instruction to differentiate the accesses. The mode of the operand is always SI because it is a memory location.

4.7 Additional Machine Modes

The TMS320C6x architecture features support for data with more than 32 bit. The type long in the TI manual [8] refers to 64 bit on the C64x and 40 bit on the other processors. Except for the C64x there are no specific load or store instructions to access long data, nevertheless a couple of instructions can operate on it. The values that are larger than 32 bit span two subsequent registers with the lower part in a register with an even number, a3:a2 being a valid example. The chips that only support 40 bit data use the lower register and the lowest 8 bits of the upper one, the other bits are ignored.

So far I implemented only a few sample instructions to support this functionally. These are explained below. The correct machine mode for the 64 bit versions is DImode. Thus, the patterns would carry names as “adddi3” and the like. They should look almost identical to the “<pattern>si” versions but must additionally take into account that they have to work with up to five registers and have to deal with the fact that the hardware does not fully support 64 bit operations in the GCC sense. At least one operand always uses only 32 bit. The correct usage of registers for 64 bit values is already checked by the macros HARD_REGNO_NREGS and HARD_REGNO_MODE_OK which Jan Parthey had already provided.

No 40 bit support has been implemented so far. I do not know for certain how this has to be done, though I guess the proper machine mode would be PDImode (Partial Double Integer), which is suitable for values that have more than 32, but do not use the all 64 bits.

Instructions with machine modes of less than 32 bit are supported to a certain degree. The current implementation allows loading, storing and expanding them to SImode.

Furthermore, there is a pattern that allows two 16 bit values to be multiplied with each other. The 32 bit result is stored to a SImode register. However, the compiler interprets all expressions in their result’s machine mode. Hence, this pattern will only be called for short expressions. The addition of other patterns supporting HImode or even QImode is questionable, since the TMS320C6x executes all instructions in 32 bit mode.

3 The appropriate C type would be long long on most architectures.
4.7.1 64 bit Instructions

I will explain “movdi” as an example of how the DI operations have to be managed. The pattern will provide moves, loads and stores, each depending on the respective abilities of the individual architectures.

```
((define_expand "movdi"
  ([set (match_operand:DI 0 "" "")
    (match_operand:DI 1 "" "")])
  ""

  
if ( REG_P(operands[0]) && CONSTANT_P(operands[1]) ) {
    emit_move_insn(gen_rtx_SUBREG(SImode, operands[0],0),
                   GEN_INT(INTVAL(operands[1])%(1<<GET_MODE_BITSIZE(SImode))));
    emit_move_insn(gen_rtx_SUBREG(SImode, operands[0],4),
                   GEN_INT(INTVAL(operands[1])>>GET_MODE_BITSIZE(SImode)));
DONE
}
if ( !REG_P(operands[0]) && !MEM_P(operands[0]) ) FAIL;
if ( !REG_P(operands[1]) && !MEM_P(operands[1]) ) FAIL;
if (TARGET_C64) {
  if ( !( REG_P(operands[0]) || REG_P(operands[1]) ) )
    FAIL;
} else {
  if ( !(REG_P(operands[0]) && REG_P(operands[1]) ) )
    FAIL;
}
})
```

The expander definition handles all copy operations of two 64 bit operands (DImode). If the instruction moves an immediate value into a DImode register, the expander pattern will split the number into a high and a low part. The high part is then moved to upper, the low part to the lower sub register (each having SImode). Apart from that, the pattern only supports register-register-moves and loads/stores in case of the C64x. Other possible operations are rejected by invoking FAIL. The compiler will then try to work around the missing instruction.

In general the register-register-moves are restricted to copy values within one register file. Moving 64 bit data from one file to the other can only be done one register at a time. Unluckily it is impossible for the expander definition to check which register file a register will later belong to, it operates only on pseudo registers. Register allocation will be done at a far later stage, so the decision how to move the registers has to be delayed.

Copy operations within one register file are managed by a single define_insn. Two define_insn_and_split handle all moves between the two register files. They split the 64 bit move into two 32 bit moves, copying each sub register separately.
The operand predicates are defined to match only registers from the specified register file. The split is executed after completion of the reload pass, when hard register numbers are already assigned. The other split definition matches a copy from the B to the A register file and looks analogously. Furthermore, there is a \texttt{define_insn} handling the valid 64 bit moves and one for the 64 bit loads and stores.

\begin{verbatim}
(define_insn_and_split "*movdiab"
 [(set (match_operand:DI 0 "registerB_operand" ""))
  (match_operand:DI 1 "registerA_operand" ""))]
 "reload_completed"
 ""
 ""
 [(set (subreg:SI (match_dup 0) 0) (subreg:SI (match_dup 1) 0))
  (set (subreg:SI (match_dup 0) 4) (subreg:SI (match_dup 0) 4))]
 ""
\end{verbatim}

This instruction definition solely outputs the assembly statement, but before doing so, it has to determine two missing register numbers. The assembler expects something like:

\begin{verbatim}
mv a3:a2 a9:a8\end{verbatim}

but the compiler internally handles these \texttt{DImode} registers as \texttt{a2} and \texttt{a8}. To add the additional registers, the internal number of the operand registers is retrieved and the next higher index number is retranslated into a register name.

\section*{4.7.2 Load/Store Multiple}

The compiler has special support for architectures that can load contiguous memory into subsequent registers. The analog applies for stores. To use this ability I have taken the implementation of "load_multiple" from the RS6000 port and adapted it to the needs of the C64x. The pattern consists of two parts:
4.7.2 Additional Machine Modes — Load/Store Multiple

```
(define_expand "load_multiple"
  [(match_par_dup 3 [(set (match_operand:SI 0 "" "")
                      (match_operand:SI 1 "" "))
                      (use (match_operand:SI 2 "" ")))])
 "TARGET_C64"
{
  ... /* check validity of operands before dereferencing pointers */
  if (GET_CODE(operands[2]) != CONST_INT
      || INTVAL(operands[2]) != 2
      || GET_CODE(operands[1]) != MEM
      || GET_CODE(operands[0]) != REG
      || REGNO (operands[0])%2 != 0)
    FAIL;
  ... })
```

The expander definition checks if the operation matches. The TARGET_C64 indicates that only the C64x is capable of such an operation. The expander pattern will be ignored if compiling for another target. Operand 2 is the number of registers that are set with this instruction. For the C64x two registers can be set simultaneously with the `lddw` instruction. After checking, the pattern emits two loads contained within one parallel statement. The pattern itself would allow more parallelism. The RS6000 for example can read up to eight registers with one instruction.

```
(define_insn "*load2"
  [(match_parallel 0 "load_multiple_operation"
    [(set (match_operand:SI 2 "register_operand" "=a,b,a,b")
      (match_operand:SI 1 "memory_operand" "Q,Q,R,R")
      (set (match_operand:SI 3 "register_operand" "=a,b,a,b")
      (match_operand:SI 4 "memory_operand" "Q,Q,R,R"))))
  "TARGET_C64"
  "lddw\t%$\t%1,\t%2:%3%?"
  [(set_attr "type" "load")
    (set_attr "datapath" "a,b,a,b")
    (set_attr "unitside" "a,a,b,b")])
```

The *load2 is the appropriate instruction definition which will turn the parallel load into the final assembly instruction. It is, again, restricted to the C64x. In theory, the corresponding "store_multiple" should look analogously. It is not implemented yet, as I could not verify that the compiler uses the load pattern correctly.

I consider this instruction particularly valuable for saving and restoring registers in the function prologues and epilogues (section 6). Assuming the loads/stores can happen in parallel, up to four registers can be saved/restored within one processor cycle.
4.8 Maintainability Improvements

During my work the developers of the GCC suite made improvements to their code and some of them affected the machine description, allowing to achieve a better maintainability of the code. Macros were introduced to avoid duplicity of code if two or more `define_insn` pattern nearly match each other. The following definition shows the usage of these macros.

```c
1 (define_mode_macro HQI [HI QI])
   (define_mode_attr ext [(HI "16") (QI "24")])
...
(define_insn "extend<mode>si2"
5 [(set (match_operand:SI 0 "register_operand" ";a,b")
   (sign_extend:SI (match_operand:HQI 1
   "register_operand" ";a,b")))]
   "ext\t\$\t%1,\t%2,\t<HQI:ext>,\t%0%?"
10 [(set_attr "type" "ext")
   (set_attr "unitside" ";a,b")])
```

In line 1 the macro `HQI` is defined containing the modes half integer (HI) and quarter integer (QI). Line 2 defines a string assigned to each one of them for later use. The `define_insn` uses this macro in line 6 which means that operand 1 may have both modes. The machine description will now cause the generation of two versions of this pattern.

```c
(define_insn "extendhisi2"
[(set (match_operand:SI 0 "register_operand" ";a,b")
   (sign_extend:SI (match_operand:HI 1
   "register_operand" ";a,b")))]
   "ext\t\$\t%1,\t%0,\t16,\t%0%?"
10 [(set_attr "type" "ext")
   (set_attr "unitside" ";a,b")])
```

```c
(define_insn "extendqisi2"
[(set (match_operand:SI 0 "register_operand" ";a,b")
   (sign_extend:SI (match_operand:QI 1
   "register_operand" ";a,b")))]
   "ext\t\$\t%1,\t%0,\t24,\t%0%?"
10 [(set_attr "type" "ext")
   (set_attr "unitside" ";a,b")])
```

Every occurrence of `<code>` will be replaced with `hi (qi)`, `<CODE> results in HI (QI) and the placeholder `<HQI:ext>` will become “16” (“24”). I used such macros throughout the back end where it seemed to improve code quality.
4.9 Low Level Instructions

During the code generation process the compiler will break down its own intermediate representation (RTL) into instructions supported by the underlying hardware. The back end should provide rules on how to translate compiler operations into assembly instructions. I have provided several new patterns which have an equivalent hardwired instruction on the TMS320C6x. More complex operations need several assembly instructions. These are dealt with in the next section.

A complete list of all implemented ‘standard names’ is shown in table 2.

```c
(define_insn "extzv"
  [(set (match_operand:SI 0 "register_operand" ";=a,b")
        (zero_extract:SI
          (match_operand:SI 1 "register_operand" "a,b")
          (match_operand:SI 2 "s5bit_operand" ";")
          (match_operand:SI 3 "s5bit_operand" ";")))]

  ""

    return ";,extu\%$%t%1,\%2,\%4,\%0%?";
  }

  [(set_attr "type" "ext")
   (set_attr "unitside" "a,b")])
```

The example definition is somewhat special. The operation represented by this pattern performs the extraction and zero extension of a bit field of operand 1. The compiler uses operand 2 for storing the width of the field and operand 3 holds the starting bit. The assembler uses other parameters as can be seen in figure 8.

```
Figure 8: Extract and Zero Extend a Bit Filed
```

The `define_insn` has to compute the missing value in order to emit the correct assembly code. This cannot be done with a static output template, C statements are used to prepare it. The ‘%?’ in the string stands for a predicate register that may be inserted in front of the instruction if that should be executed conditionally. Function `c6x_print_operand` will handle this.
<table>
<thead>
<tr>
<th>name</th>
<th>new function</th>
</tr>
</thead>
<tbody>
<tr>
<td>“abssi2”</td>
<td>X compute absolute value of a register</td>
</tr>
<tr>
<td>“addsi3”</td>
<td>add two registers</td>
</tr>
<tr>
<td>“andsi3”, “iorsi3”, “xorsi3”</td>
<td>perform logical operation</td>
</tr>
<tr>
<td>“ashlsi3”, “ashrsi3”, “lshrsi3”</td>
<td>perform shift operations</td>
</tr>
<tr>
<td>“b&lt;code&gt;”</td>
<td>branches if condition is true</td>
</tr>
<tr>
<td>“call” / “call_value”</td>
<td>call a function returning void/non-void</td>
</tr>
<tr>
<td>“clzsi2”</td>
<td>X detect leftmost 1 bit</td>
</tr>
<tr>
<td>“cmpsi”</td>
<td>compare two register and internally store condition code</td>
</tr>
<tr>
<td>“extend&lt;mode&gt;si2”</td>
<td>X sign extend register value</td>
</tr>
<tr>
<td>“extv”</td>
<td>X extract a bit field and sign extend it</td>
</tr>
<tr>
<td>“extzv”</td>
<td>X extract a bit field and zero extend it</td>
</tr>
<tr>
<td>“indirect_jump”</td>
<td>unconditional branch to an address taken of a register</td>
</tr>
<tr>
<td>“jump”</td>
<td>unconditional branch to a label</td>
</tr>
<tr>
<td>“load_multiple”</td>
<td>X load subsequent memory locations into registers</td>
</tr>
<tr>
<td>“mov&lt;mode&gt;”</td>
<td>X copy operation</td>
</tr>
<tr>
<td>“mulhi3”</td>
<td>X multiply the lower halves of two registers</td>
</tr>
<tr>
<td>“mulsi3”</td>
<td>X multiply two registers</td>
</tr>
<tr>
<td>“negsi2”</td>
<td>X negate register value</td>
</tr>
<tr>
<td>“nop”</td>
<td>do nothing</td>
</tr>
<tr>
<td>“set_high”</td>
<td>X set high part of register</td>
</tr>
<tr>
<td>“set_lo_sum”</td>
<td>X set low part of register</td>
</tr>
<tr>
<td>“one_cmplsi2”</td>
<td>X one complement</td>
</tr>
<tr>
<td>“popsi”</td>
<td>X pop register from stack</td>
</tr>
<tr>
<td>“epilogue”</td>
<td>emit function epilogue</td>
</tr>
<tr>
<td>“prologue”</td>
<td>emit function prologue</td>
</tr>
<tr>
<td>“pushsi”</td>
<td>X push register onto stack</td>
</tr>
<tr>
<td>“s&lt;code&gt;”</td>
<td>X set register to one (zero) if condition is true (false)</td>
</tr>
<tr>
<td>“subsi3”</td>
<td>subtraction of two registers</td>
</tr>
<tr>
<td>“tablejump”</td>
<td>jump to a variable address</td>
</tr>
<tr>
<td>“zeroExtend&lt;mode&gt;si2”</td>
<td>X zero extend register value</td>
</tr>
</tbody>
</table>

Table 2: Implemented Compiler Patterns (‘standard names’)

*a<code> is one of {eq, ne, gt, ge, lt, le, gtu, geu, ltu, leu}.
*b<mode> is one of {qi, hi}.
*c<mode> is one of {qi, hi, si}.
*d<code> is one of {eq, gt, lt, ltu}.
*e The subtraction of an immediate is already handled by “addsi3”.
*f This is used to implement a dispatch table if no “casesi” pattern exists.
*g<mode> is one of {qi, hi}.
4.10 High Level Instructions

For almost anything the back end does not provide, the compiler substitutes the missing part. Multiplications by small integers, for example, can be efficiently rewritten with shifts and adds. If the missing pattern is too complex to be rewritten in the assembly output, the compiler uses one of library functions provided by the ‘gcclib’.

Directly after the compiler is built, it is immediately used to compile this library. Being a cross compiler for the new target platform, the ‘gcclib’ will consist of linkable code for the TMS320C6x processors.

The ‘gcclib’ contains algorithms that break down complex instructions such as a `div` using only what is supported by the target processor. These algorithms are very helpful, as they allow to develop the machine description step by step. It is not necessary to support the complete compiler instruction set from the beginning. Functions provided by ‘gcclib’ use commonly available instructions to build more complex operations. Using specific machine instructions may largely improve their performance. This can be achieved by two different approaches. Expander definitions may emit a sequence of instructions to imitate one complex operation. Secondly, new built-in functions can substitute the generic library calls by machine optimized algorithms.

Exemplary for the first approach, I provided support for a multiplication that does not fit in the ALUs of the TMS320C6x chips. Expander definition imitate 32 bit multiplication by emitting the six necessary instructions. Before explaining the implementation details I will briefly discuss the theoretical background.

The TMS320C6x have several multiplication instructions. Though only providing 16 bit, every possible case of a multiplication is supported (including signed and unsigned). For the algorithm I will make use of these instructions:

- **mpyu**: multiply the 16 least significant bits (lsb) of operand 1 with the 16 lsb of operand 2. Handle both operands as unsigned values.
- **mpyh**: multiply the 16 most significant bits (msb) of operand 1 with the 16 msb of operand 2. Handle both operands as signed values.
- **mpyhslu**: multiply the 16 msb of operand 1 with the 16 lsb of operand 2. Handle the msbs signed and the lsbs unsigned.
- **mpyluhs**: multiply the 16 lsb of operand 1 with the 16 msb of operand 2. Handle the lsbs signed and the msbs unsigned.

According to this scheme several other instructions exist.

---

4 In fact, gcclib is currently not built due to problems with the assembler.
The mathematical background is a simple transformation:

\[(h_1 \cdot 2^{16} + l_1) \cdot (h_2 \cdot 2^{16} + l_2) = (h_1 \cdot h_2) \cdot 2^{32} + (h_1 \cdot l_2 + l_1 \cdot h_2) \cdot 2^{16} + (l_1 \cdot l_2)\]

As the compiler only uses the lower 32 bit for the result and discards the higher bits, the `mpyh` is not needed here. The result of the `mpyu` is stored to the destination register. The results of `mpyl` and `mpyh` are first accumulated, shifted to the right position and finally added to the destination register. The pseudo assembly code to represent ‘\(O = M \times N\)’ is:

```
mpyu M, N, O ; o := l_1 \cdot l_2
mpyhslu M, N, T1 ; t_1 := h_1 \cdot l_2
mpyluhs M, N, T2 ; t_2 := l_1 \cdot h_2
add T1, T2, T3 ; t_3 := h_1 \cdot l_2 + l_1 \cdot h_2
shl T3, 16, T4 ; t_4 := lsb_{16}(h_1 \cdot l_2 + l_1 \cdot h_2) \cdot 2^{16}
add T4, O, O ; o := lsb_{16}(h_1 \cdot l_2 + l_1 \cdot h_2) \cdot 2^{16} + (l_1 \cdot l_2)
```

This instruction sequence uses four temporary registers. Two would theoretically suffice, but the compiler will optimize the registers count anyway. We do not have to do that here.

To implement this algorithm I have added the expander definitions for “`mulsi3`”, “`umulsi3`” and several instruction definitions to provide the additional multiply patterns. On of these is `mpyhslu`

```c
#define_insn "mpyhslu"
{(set (match_operand:SI 0 "nonimmediate_operand"
"a, b, a, b, a, b, b")
(unspec:SI [(match_operand:SI 1 "register_operand"
"a, b, a, b, a, b, b, a")
(match_operand:SI 2 "reg_or_8bit_operand"
"a, a, b, a, b, L, L, L, L")]
UNSPEC_MPYHSLU))}

"mpyhslu\[t\%t\%t\%t\%t\%t\%t\%t\]
{(set_attr "type" "mpy")
(set_attr "crosspath" "no,x, no,x, no,x, no,x, no,x")
(set_attr "unitside" "a, a, b, a, a, b, b")}}
```

Please note the `unspec` in line 4. This is a machine specific instruction the compiler recognizes as ‘unspecified’, meaning that it does not know how to change these patterns. It will just pass them all along the optimization stages. It will not alter them, but they may be deleted if they are useless. These instructions are differentiated by an index. In the above example this index is represented by the constant UNSPEC_MPLYHSLU These constants are defined in a `define_constant` construct to provide a better readability.
4.11 Branch Handling

This is not actually an optimization, but the old implementation of calls and branches did not work with the GCC version 4.0. According to the GCC mailing list, it should not even have worked with the prior ones. Calls could not return as the return label was non-existent and conditional branches were scheduled far too early. This would have applied to the 3.3 version too, but the back end then was not intended to support optimization.

4.11.1 Calls

The compiler assumes that there is an instruction which implements the call-return semantics on the architecture to allow function calls. On the TMS320C6x there is no such instruction. Multiple instructions have to be emitted in order to mimic call-return functionality.

```
call:
   mvkl ret_label, a1
   mvkh ret_label, a1
   stw a1, *--b15
   mvkl f_label, a1
   b a1
   nop 5
ret_label:
```

The lines 1 - 3 of the call sequence store the return label address. If follows the branch sequence to jump to the function address. This is where the function prologue begins.

The function epilogue ends with the return sequence. Its first instruction loads the return address from inside the stack. I used `*+a15[1]` here, because that is equivalent to using `*b15++` after popping the frame pointer in line 2. This way, the `ldw` can be scheduled earlier, as to avoid the NOP instructions (line 4) needed otherwise.

When I first upgraded the back end to support GCC version 3.5, the assembly code was not generated as shown above. The return label was completely missing, though the code in `c6x_call_expand` did clearly emit it. The compiler optimized it away because it never appeared in a branch statement, only its address was taken. After corresponding with the GCC community it became evident that the call pattern had to ‘memorize’ the label and print it to the assembly file itself. The optimization passes are not designed to deal with those labels.

At first, I tried to attach the label information to the call instruction with the help of instruction notes. These notes are stored in a list present in every instruction. The different passes store various information in it. Results of branch prediction, flow analysis data, and information about
the data a register of this function actually contains. I thought I could use this to transport the label to the output stage. However, this REG_LABEL note could not be stored in the list, since at the time of call expanding the structure is not yet initialized. Trying to circumvent that I introduced a new information type which I named MD1, after a convention the GNU assembler uses to distinguish machine dependent data types. This did not work either. The instructions are copied, splitted, merged and restructured during the entire compilation. Appropriately, there exist functions that have to decide which notes have to be copied or merged as well, and which do not. I do not want to change vital code. Therefore, I designed the back end to deal with this labels itself.

The GCC already uses hash table handling for internal use. Actually, I even found different implementations. The file 'c6x.x' now contains several functions wrapping the hash functionality. They handle the creation of the table, the insertion, finding and removal of entries, and the table deletion. The hash table is created by the target macro ASM_FILE_START which is called when the compiler starts outputting a new assembly file. The deletion is done by the corresponding ASM_FILE_END. The entries are not deleted until then, because the processing cycle information may be used multiple times.

The call expander pattern will store the label into the hash table, so the output functions can extract it and put it behind the branch statement, as necessary.

### 4.11.2 Conditional Branches

The problem with conditional branches, especially those appearing in loops, is tightly coupled with the handling of basic blocks. I will therefore give a short introduction into the topic.

![Figure 9: Basic Blocks](image1)

![Figure 10: Corrupt Branches](image2)

Figure 9 shows a loop enclosed by two other basic blocks. A basic block is a sequence of instructions that can only be entered at the beginning and only be left at its end. It always starts
4.11.2 Branch Handling — Conditional Branches

with a label and ends with a branch, but contains no other branches nor labels. If it ends with an unconditional branch there is one edge connecting it to the next basic block. If it ends with a conditional branch it has two edges: one for the `then` and one for the `else` part of the condition. If the condition contains no else part, the CFG will contain an edge leading to the block with the statements following the `if` block.

Figure 10 shows what used to happen to optimized loops. Normally the final branch depends on all instruction within the basic block. This ensures that all these operations are completed before the branch occurs. Somehow, the compiler loses track of the label when the branch is split in multiple instructions. I tried to add a `'(use ...)'` statement to the branch. Though this has worked before, it fails here. For now, I have simply reverted the branch statements to a simple `'b label'`. This is not correct for all possible cases, but branches exceeding a length of $2^{21}$ bytes should rarely occur on embedded systems. This is the maximum distance which can be encoded in a `b label` instruction.
5 Performance

There are a lot of independent optimization algorithms at work. Though every one of them operates only within a limited area, they have to be seen as a unity. Each one for itself would achieve only a minimal performance increase.

Logically, there are bundles of algorithms that are useful to combine. These bundles are used when –Ox is specified on the command line. The different bundles are used to achieve different optimization goals.

–O0 (default): no optimization
Optimize compile time. No optimizations take place, as they would slow down compilation time. Can be used for debugging.

–O1 (-O): default optimization
Optimize program execution time by only moderately increasing compilation time. Delayed Branch scheduling is activated.

–O2: heavy optimization
Optimize program execution time, but try to avoid code size increase. Function inlining for instance will only be used in rare cases. This significantly increases compilation time.

–O3: optimal execution time
Optimize for optimal execution time in the sense of best effort. This may increase code size, and will probably make debugging difficult. It may happen that function calls are substituted by their result. Debugging this function is virtually impossible then.

–Os: optimal code size
Optimize for best code size. This may slow down execution time compared to –O3

5.1 Exemplary Performance Improvements
At any time during the entire development process I used several test cases to test the correctness and performance of the improved compiler. This section will use these files to show some impressive performance improvements.

5.1.1 Avoiding NOPs
The prior TMS320C6x back end inserted NOPs after every instruction that may have needed them. As there are quite a lot of load an branch instructions in a usual program, the code contains

---

5 Compilers can not compute the best possible schedule, unless the halting problem is solvable.
5.1.2 Exemplary Performance Improvements — Filling Branch Delay Slots

a lot of (unnecessary) NOPs. Every function call needs at least two branches. One for the call and one for the return. Additionally, each function has to load its parameters from the stack. This is usually done by a sequence of load instructions.

```
...  
add    a4,   a1, a4  
ldw  *+a15(24), a0  
nop     4  
add    a4,   a0, a4  
ldw  *+a15(28), a1  
nop     4  
add    a4,   a1, a4  
mvkl   32,   a14  
add    a15,   a14, a0  
ldw  *a0,   a0  
nop     4  
add    a4,   a0, a4  
...  
```

9 + 12 = 21 cycles

The scheduler now arranges the loads one after another as they are independent. Only one line here contains a NOP. It ensures that the memory value has been loaded to register \( a_8 \) before being used in the \( \text{add} \) instruction. Using the other indexing mode (relative instead of absolute offsets) has the benefit that larger offsets can still be reached, without having to be moved to a register first.

```
...  
ldw  *+a15[7], a9  
ldw  *+a15[8], a4  
ldw  *+a15[1], b14  
add    a3,   a8, a3  || ldw  *+a15[5], a8  
add    b3,   a3, b3  || ldw  *+a15[6], a3  
nop    3  
add    a8,   b3, a8  
```

6 + 3 = 9 cycles

5.1.2 Filling Branch Delay Slots

With an activated delayed branch scheduler the instructions are reordered to avoid the annoying ‘nop 5’ following each branch. Unfortunately, it can not yet be combined with parallel scheduling due to the difficulties, discussed in section 4.4.4. Together with the other optimization passes, enabled with the option \(-O\), the delayed branch scheduling yet improves the performance a little.

```
...  
add    a14  
nop     5  
```

1 + 5 = 6 cycles

```
...  
b     b14  
mpyluhs a3,  a3,  a3  
add    a4,  a3,  a4  
shl     a4,  16, a4  
add    a8,  a4,  a4  
add     4,  b15, b15...  
```

6 + 0 = 6 cycles
5.1.3 Conditional Instructions

The last example showed how the delay slots of a branch can be used efficiently. An even better choice is to avoid branches at all. This can be achieved with the help of conditionally execution. It cannot eliminate function calls and its corresponding return sequence. Function inlining already does that, trading speed against space, by replacing the call statement with the function code. It can, however, replace branches that the compiler emits for if-then-else constructs. The left example shows the resulting code for such a case, the right code uses a more elegant version.

```
...  
  cmplt b1, b0, b0
  [[b0]]  
  b  L4
  ...  
  ; then part
L3:   
  ...  
  ; common part
  ; return
L4:   
  ...  
  ; else part
  b  L3
  ...  

  ...  
  ; mixed then & else part
  cmplt b0, b8, b0
  [[b0]]  
  add -2, b3, a4
  [[b0]]  
  mv b3, a3
  ...  
  ; return
```

Instead of branching to the then part and back, if the condition is false, the compiler here mixes the instructions of the then and the else part. The condition only controls which of the results will be used. Mixing both parts has the benefit that both can share expressions and that the instructions can be parallelized. Branches will only be used, if extensive calculations have to be done in one of the parts.

5.1.4 Parallel Instructions

The most impressive speedup can be achieved with `–mparallel` which is activated by default if `–O2` is specified. Firstly, because instructions can now be scheduled in parallel. Secondly, because the scheduler provides information to reduce the number of emitted NOPs significantly. They will only be used to retain true dependencies. As the first example already showed a noticeable decrease in execution cycles, the next will focus on the parallelism.
If not considering the NOP instructions, this is a speedup of about 2.4. Of course this value largely depends on the actual code, but also on some limitations which have yet to be overcome.

5.2 Benchmarks

This section should provide an overview of the achieved improvements. The results should also allow a comparison with the TI compiler, whose results can be regarded as close to an optimal schedule.

5.2.1 Benchmarking Environment

To show the improvements, I activated all –O2 optimizations for all tests. To show the impact of the delayed branch scheduling, I deactivated the parallel scheduling and specified –fdelayed-branch instead. Unfortunately, this also deactivates the dependency aware NOP insertion. Therefore,
these tests are more comparable to the old version than to the new parallel one.

I ran the tests on the TI compiler with three different setups. All tests were executed once with option –O3 to let this compiler run all its optimizations. The loop test was also done with deactivated Modulo Scheduling to see its impact and also to have comparable results, as the TMS320C6x GCC back end is not yet capable of it.

5.2.2 Results

Table 3 shows the cycle counts of the produced schedules. The first number is the count of actual instructions, the second counts the NOP cycles.

<table>
<thead>
<tr>
<th></th>
<th>3.3 -O2</th>
<th>4.0 -O2(dbs)</th>
<th>4.0 -O2(p)</th>
<th>4.0 -O3(p)</th>
<th>TI -O2</th>
<th>TI -O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>10+13</td>
<td>9+11</td>
<td>7+6</td>
<td>6+6</td>
<td>2+4</td>
<td>2+4</td>
</tr>
<tr>
<td>call</td>
<td>61+63</td>
<td>53+60</td>
<td>34+22</td>
<td>6+7</td>
<td>15+4</td>
<td>7+11</td>
</tr>
<tr>
<td>condexec</td>
<td>30+34</td>
<td>25+24</td>
<td>18+13</td>
<td>18+13</td>
<td>10+8</td>
<td>10+8</td>
</tr>
<tr>
<td>loop100</td>
<td>3462+2731</td>
<td>1350+1463</td>
<td>824+826</td>
<td>1010+910</td>
<td>712+108</td>
<td>221+3</td>
</tr>
</tbody>
</table>

Table 3: Comparing Results

5.2.3 Conclusions

*Delayed branch scheduling uses less instructions than version 3.3.*

This cannot be explained by the delayed branch optimization pass itself. That decreases only NOPs, but has no effect on the other instructions. The improvement shows that the other passes have become better as well. New assembler instructions now allow a better code generation. The support of address modification reduces the number of necessary addi instructions. Last but definitely not least, the other parts of the compiler evolved as well. The significant drop in instruction cycles, here, results from saved library calls which the old version needed multiply registers.

*Delayed branch scheduling decreases less NOPs than parallel scheduling.*

The solely purpose of the delayed branch scheduling is to save NOPs, but the parallel scheduling does this a lot better. Curiously, the answer has nothing to do with the optimizations itself.

The activated parallel scheduler does not only group instructions. It also provides information about the processor cycle the instruction is scheduled in, actually the cycle within the
basic block. The back end uses this information to emit only as many NOPs as there are necessary for the next instruction to have their source operands available. This scheduler has to be deactivated, because its poor interaction with delayed branch scheduling.

Having to act pessimistically, every delay slot must be filled with NOPs. The delayed branch scheduler only reduces the amount of NOPs after branches. These are relatively rare compared to these of loads, for instance. The parallel scheduler almost eliminates all dependency delays, leaving only branches untouched. In the end, less NOPs are left in the schedule.

Parallel scheduling uses 20%-40% less instructions than the old back end.

This is the usual rate of performance increase that can be observed. Apart from the conceptional limitations which are discussed below, the parallel scheduler groups as many instructions as the data dependencies allow. Considering only the ability to parallelize instructions, the TI compiler is not much better.

The TI compiler is almost ever faster than GCC.

Almost! The ‘call’ test is faster on GCC. The GCC inlines functions, the TI compiler somehow does not. It only optimizes the function itself. As this test only consists of a single call to a function that returns a constant value, due to constant input parameters, the GCC simply does not call the function. Hence, it produces the faster program.

However, in most cases the TI compiler produces the faster program. It does profit from several interacting algorithms:

Software Pipelining (modulo scheduling)

This technique, paired with piped branches allow loops with as less as one cycle. The shortest loop the GCC is currently capable of is as many as six cycles. With deactivated software pipelining (–mu) in test ‘loop100’ the TI loop is only one cycle ahead (8 → 7).

Delayed Branches

The TI compiler can handle delayed branches and parallel scheduling at once. Hence, it can put up to 40 instructions in the delay of a branch. The GCC currently only waits during these five cycles. This explains the many NOP instructions in test ‘loop100’

Parameter Passing in Registers

The GCC handles all function parameters by passing them on the stack. The function than has to load these values from stack into the working registers, which introduces delays.

The TI compiler eliminates these loads by passing all arguments in registers. Furthermore, this reduces stack managing costs to zero.
The listed limitations are not inherent to the GCC compiler. Parameter passing can be implemented at will, delayed branch scheduling and parallel scheduling can work together, though the implementation may become somewhat difficult. The software pipelining is already being developed for the GCC. It has yet to be seen, if it can prove as powerful as the TI version which was implemented for the specific architecture.

5.3 Conceptional Limitations

The new back end did improve the performance of the resulting program code in contrast to the old one. Still, the TI compiler is far ahead. Not only considering performance, but also in the effort of its development. I will now deal with some issues that will probably prohibit the GCC from closing this gap, unless certain compiler internal parts are rewritten or at least reorganized. Such large changes cannot be done without affecting other platforms and would therefore have to done with great care and in cooperation with the GCC community.

The GNU C compiler is a so called ‘phase decoupled’ compiler which basically means that it performs its optimizations step by step and not all together. This is necessary, because GCC supports many targets and it is dependent on each target which optimizations are useful and should therefore be executed. Additionally the enormous complexity of an optimizing algorithm which integrates all promising optimizations can not be handled for such a large compiler. For smaller compilers with a narrow range of supported platforms such an approach can be nevertheless be successive, as has been shown in [10].

For highest performance, several different optimizations would have to be united. I will now show some examples of areas where one optimization pass cannot be as much effective as possible, because it has to live with the results of an earlier one.

**Instruction Selection → Scheduling**

During instruction selection and earlier optimization passes it is tried to find the cheapest instruction with the desired effect. As these passes do know nothing of the architecture, the resulting code may be less efficient, though. A multiplication with a value that is a power of two is very likely to be transformed into a cheaper shift operation.

```
1      mvk .S1  4,  a3
2      mvk .S1  5,  a4
3      add .L1 a3, a4, a4
4      shl .S1 a5,  4,  a5
5      shl .S1 a6,  4,  a6
```

5 cycles

```
1      mvk .S1  4,  a3
||      mpy .M1 a5, 16,  a5
2      mvk .S1  5,  a4
||      mpy .M1 a6, 16,  a6
3      add .L1 a3, a4, a4
```

3 cycles
5.3 Conceptional Limitations

Effectively the shift code is two cycles longer. The add instruction in this case completely hides the delay cycle of the multiply, and executing this in parallel to the add instruction reduces its cost to zero. As the instruction selection is not coordinated with the scheduling process, there is another way of telling it which instructions are best to select. Several macros exist that define relative costs of instructions. Currently the default assumptions control the compilers behavior. To improve the scheduling further, this has to be changed.

Register Allocation → Scheduling

The register allocator assigns the hardware registers the individual instructions operate on. Normally this has no influence on the scheduler, as in common architectures any functional unit can be used with any register. On the TMS320C6x the register assigned almost completely determines the functional unit that will later execute the instruction. Unfortunately, the register allocator is not designed for scheduling. Assigning \( b4 \) and \( b6 \) instead of the equivalent \( A \) registers would turn the above sequence into an even shorter one.

\[
\begin{align*}
1 & \text{mvk} . S1 & 4, & a3 \\
1 & | \text{mpy} . M1 & a5, & 16, & a5 \\
1 & | \text{mvk} . S2 & 5, & b4 \\
1 & | \text{mpy} . M2 & b4 & 16, & b6 \\
5 & \text{add} . L1 & a3, & a4, & a4
\end{align*}
\]

To a certain degree it is possible to influence the register allocator as well. It allocates the registers in the order of `REG_ALLOC_ORDER` in ‘c6x.h’. I have defined this macro to list the registers in an alternating way. So after \( a3 \) follows \( b3 \), then \( a4 \) and so on. I had to do that anyway to avoid that the allocation process runs low on registers of one side. This would probably result in a compiler error. Mixing the registers has also positive impact on the scheduling, as the allocated registers are almost equally divided between both register files. The result is not perfect, but I currently see no way of how to deal with the remains of this problem.
6 Further Improvements

A complex software such as a C Compiler cannot be completed within a year, even though we only have developed a new back end for it. Accordingly, there is still a lot to improve.

Function Prologues and Epilogues

It would be valuable to parallelize the push/pop series that save and later restore the registers a function uses. This could be done with the help of a second base register apart from the stack pointer. Originally, I chose a14 for this purpose, because it is already used as a scratch register in other parts of the back end. Any other free register would serve as well, even the ones previously saved. The push and the pop parts are analogous, so I will only discuss the pushes here.

<table>
<thead>
<tr>
<th>TIC6x:</th>
<th>TIC64:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>1:</td>
</tr>
<tr>
<td>stw a2, *--b15</td>
<td>mv b15, a14</td>
</tr>
<tr>
<td>2:</td>
<td>2:</td>
</tr>
<tr>
<td>stw a3, *--b15</td>
<td>stdw a3:a2, *-a14[1]</td>
</tr>
<tr>
<td>3:</td>
<td>3:</td>
</tr>
<tr>
<td>stw a2, *--b15</td>
<td>stdw a3:b2, *-b15[3]</td>
</tr>
<tr>
<td>4:</td>
<td>4:</td>
</tr>
<tr>
<td>stw a3, *--b15</td>
<td>addk 16, b15</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The current version needs one cycle for each register. An updated version could push up to four registers in the same cycle in case of the C64x processor. The overhead is minimal. The necessary adjustment of the pointers can normally be executed in parallel to the last push. If the offsets became larger, several addk instructions may have to be inserted. Even that would require no extra cycle. If there are two or more register to be saved, the parallel version will at least be as effective as its serial counterpart. Due to the register allocation order (section 4.5) the registers are usually almost equally parted between the two register files. Thus, a high degree of parallelism should be possible, if the registers are ordered in a clever way.

Unluckily, I encountered a problem with a vital instruction: ‘mv b15, a14’ will evoke an error. The compiler tries to delete the move instruction by aliasing a14 with b15 which is normally quite clever. The compiler assumes that reducing the number of used registers is always a good idea, but in the case of the TMS320C6x processors this sometimes prohibits parallelism.
I then tried to use the frame pointer as the second base register. The amount of space between
the stack and the frame pointer is known. However, this results in heavily adjusting offsets
and might introduce new errors, so I did not complete that version.
Recently, I became aware of the fact that the machine description may use machine specific
RTL operations called `unspec` instead of any other RTL operator (e.g. `plus`). The next
section will use these to implement 32 bit multiplications. With their help the `mv b15, a14`‘
might be possible. As the compiler does not know that an appropriate `unspec` is in
fact only a move, it should not try to eliminate this instruction.
However, the most performance improving implementation should not use the stack if any-
how possible. Each value which is passed in an register saves a push and a pop. It may also
reduce the necessity to use stack and frame pointers.
Even if parameters are passed via stack slots. The frame pointer may not be necessary. If the
function does not call another and has only simple local variables (no `alloca`) the frame
pointer can be omitted as well. The function prologue and epilogue of the ‘tic4x’ could serve
as a reference implementation.

**Defining Costs**

Analyze the effect of implementing functions for the target macros in ‘c6x.h’ that can define
relative costs of constants or instructions. They tell the compiler which instructions it should
prefer. It may be useful to use multiplication instead of shifts, as the multiplication units
will probably not be used extensively. This cost should depend on how many functional
units may execute the instruction. The more units can execute the instruction, the more
likely the scheduler will find a free one to schedule this instruction, the lower the actual
costs will be.

**Branch shortening**

Make use of the branch shortening pass. This will use `B label` instead of `B reg` where
possible. The label only version can be used in the usual case when the branch distance is
below $2^{21}$ byte, but the barrier has to be checked to guarantee correct code. For this feature
the `length` attribute must be activated. It counts for any instruction how many bytes it
occupies. On the TMS320C6x that is always 4 byte, but the compiler has to be made aware
of NOP the instructions inserted at a later stage. The target hook `ADJUST_INSN_LENGTH`
should be good starting point for this. It can update the length of an instruction if NOPs have
to be inserted after it. If the compiler then knows the correct distance between the branch
instruction and the label, it can insert the instructions as needed.

**Delayed Branch Scheduling**

The delayed branch scheduling is essential for good performance. Any branch currently
FURTHER IMPROVEMENTS

needs six cycles. Making DBS and parallel scheduling work together will certainly save a lot of NOPs. The problem is described in detail in section 4.4.4.

A possible solution to this might be using the target hook `TARGET_SCHED_FINISH` to group any parallel instructions into explicit parallel RTL statements. The delay branch scheduler then had to pick complete blocks and could no longer separate parallel instructions. The drawback of this solution is that these parallel instructions had to be isolated again before attempting their output. The function `final_scan_insn` will not split the parallel statement to search for matching RTL patterns. It will try to match the parallel block and will fail. Providing all of these blocks is not really an option. It would mean to declare every possible execute packet.

Currently, it is also not possible to schedule a loop with less than six cycles, because the delay of the branch cannot be hidden. Using branches in delay slots of other branches can produce one cycle loops[9, p. 5-42]. Please notice the problems that arise with dependencies. They are discussed in section 4.4.4 too.

Instruction Scheduling

Though scheduling has become a lot better, it can still be improved significantly.

- Adjust the automata descriptions of C64x and C67x to better reflect their architecture and constraints.
- The target hooks `TARGET_SCHED_REORDER(2)` could exchange whole instructions by others. They must perform the same action, but may use a different functional unit. If this unit is free in the current processor cycle, it could be emitted in parallel to the other instructions in the ready cue. Consider the initialization of a variable. This could be done very differently:

```plaintext
mvk 0, reg ; uses S
sub reg, reg, reg ; uses L|S|D
xor reg, reg, reg ; uses L|S
```

However, the substitution has to be done very carefully, as this target hook is called often and dealing with complete instructions tends to be complex.
- Swing Modulo Scheduling claims to work with the PowerPC architecture. This machine description should be consulted for substantial differences. This should be the counterpart to the Software Pipelining which TI uses.

Linear Assembly

Within the the TI tools [7] the so called ‘Assembly Optimizer’ takes linear assembly as its input. This is assembly code that has not been register allocated and not been scheduled. It does not contain information about parallel instructions or instruction latencies (no NOPs).
It is meant to serve as a mediator between the compiler that has to produce good code for general purpose and the writer of assembly code that aims at optimal code for a single purpose. It can achieve quite stunning optimizations if it works in team with a skilled programmer. The idea behind this is to aid the programmer in coding, still, a lot of information has to be provided, which the assembler/compiler cannot derive itself.

It might be useful to instruct the back end to emit linear assembly code instead of normal assembly. That could be refined using the TI Assembly Optimizer. Even the TI compiler does not provide such a feature. It is questionable, however, if the compiler could emit enough information to achieve highest performance. The TI tools describe very extensively how the programmer can provide information the TI compiler can not provide [9, Section 5]

40/64 Bit Support

Extend support for 64 bit for the C64x and implement 40 bit support for the other chips. Test the load_multiple and provide an equivalent store_multiple.

Implementing Complex Instruction Pattern

Implement further patterns to enable the compiler to better translate RTL into assembly. New expander definitions (section 4.10) could do that. It may seem feasible to provide own implementations that better suit the architecture than the generic implementation.

Floating Point Support

Implement floating point instructions for the C67x.

SIMD Support

The hardware supports several kinds of Single Instruction Multiple Data (SIMD) instructions, add2 and add4 given as examples. This should be used as another way of parallelism.

Use Profiling Information

Provide and use profiling information.

Debugging Support

So far, no effort has been made to implement debugging support. As soon as useful applications should be developed this will be essential.

User Interface

Test available optimization passes, hardware dependent and independent on their impact on the resulting schedule. Provide suitable passes for the different optimization levels.

Saving Memory

It may be possible to save some memory, by carefully deleting hash table entries s soon as they are no longer needed.
References


## Index

NOP ............................................. 34, 50, 54
abssi2 ........................................... 44
adddi3 ........................................... 38
addressing modes ......................... 37
addsi3 ........................................ 21 ff., 44
aggregate replacement .................... 10
alias analysis ................................. 10
andsi3 ........................................ 44
ashlsi3 ......................................... 44
ashrsi3 ......................................... 44
assembly output .............................. 16
attributes ..................................... 24 ff.
automaton description ................... 28
b<code> ........................................ 44
basic blocks ................................ 9, 15
benchmark .................................... 58
branch shortening ......................... 15, 59
call ............................................ 44
call_value .................................... 44
casesi ......................................... 44
clzsi2 ......................................... 44
cmpsi .......................................... 44
code macros .................................. 42
common subexpression elimination ...... 13
global ......................................... 13
compiler proper ............................. 4
conditional execution .................... 26, 52
constant propagation ..................... 11
control flow graph ......................... 9
copy renaming ................................ 9
cost ........................................... 20
cross path .................................... 28
dead code elimination ..................... 8 ff., 11
dead store removal ......................... 10
debugging .................................. 16, 61
dominator optimization .................. 9
DONE ........................................ 12, 21
DSP ............................................ 11
epilogue ...................................... 55, 58
epilogue ........................................ 44
exception handling ....................... 13
extend<mode>sì2 .............................. 44
extv .......................................... 44
extzv .......................................... 44
FAIL ........................................ 12, 39
FINAL_PRESCAN_INSN ..................... 31
GCC ........................................... 4
GENERIC ..................................... 6
GIMPLE ........................................ 6 ff., 12
gimplification ............................... 6
gimplify ....................................... 6
if-conversion ................................ 10, 13
indirect_jump ................................ 10, 13
Initiation Interval ......................... 44
instruction dependencies ................. 30
Interprocedural Optimizations .......... 17
iorsi3 ......................................... 44
jump ........................................... 44
jump optimization ........................ 13
<table>
<thead>
<tr>
<th>Term</th>
<th>Page1</th>
<th>Page2</th>
</tr>
</thead>
<tbody>
<tr>
<td>life analysis</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>limitations</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>linear assembly</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>load_multiple</td>
<td>40, 44</td>
<td></td>
</tr>
<tr>
<td>loop optimization</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>unrolling</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>unswitching</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>lower arithmetic</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>lower control flow</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>lshrsi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>machine modes</td>
<td>12, 38, 61</td>
<td></td>
</tr>
<tr>
<td>mode macros</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>modulo switching</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>modulo scheduling</td>
<td>16, 55</td>
<td></td>
</tr>
<tr>
<td>mov&lt;mode&gt;</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>movdi</td>
<td>24, 39</td>
<td></td>
</tr>
<tr>
<td>movsi</td>
<td>19, 23</td>
<td></td>
</tr>
<tr>
<td>mulhi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>mulsi3</td>
<td>44, 46</td>
<td></td>
</tr>
<tr>
<td>negsi2</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>25, 44</td>
<td></td>
</tr>
<tr>
<td>one_cmplsi2</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>parallelism</td>
<td>31, 35, 52</td>
<td></td>
</tr>
<tr>
<td>partial redundancy elimination</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>pass manager</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>phase decoupled</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>phi expression</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>phi optimization</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>popsI</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>PRINT_OPCODE</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>PRINT_OPERAND</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>profiling</td>
<td>10, 61</td>
<td></td>
</tr>
<tr>
<td>Program Level Optimization</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>prologue</td>
<td>55, 58</td>
<td></td>
</tr>
<tr>
<td>prologue</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>pushsi</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>REG_ALLOC_ORDER</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>register allocation</td>
<td>14, 35, 57</td>
<td></td>
</tr>
<tr>
<td>graph coloring</td>
<td>14, 44</td>
<td></td>
</tr>
<tr>
<td>register movement</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>reloading</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>RTL expansion</td>
<td>23, 45, 61</td>
<td></td>
</tr>
<tr>
<td>RTL Generation</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>RTL splitting</td>
<td>19, 21, 23</td>
<td></td>
</tr>
<tr>
<td>s&lt;code&gt;</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>scheduling</td>
<td>14, 15, 32, 35, 37, 52, 54, 60</td>
<td></td>
</tr>
<tr>
<td>delayed branch</td>
<td>15, 32, 35, 51, 54, 59</td>
<td></td>
</tr>
<tr>
<td>instruction selection</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>register allocation</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>set_high</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>set_lo_sum</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>Single Static Assignment form</td>
<td>see SSA</td>
<td></td>
</tr>
<tr>
<td>software pipelining</td>
<td>16, 55</td>
<td></td>
</tr>
<tr>
<td>SSA</td>
<td>7, 11</td>
<td></td>
</tr>
<tr>
<td>standard names</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abssi2</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>adddi3</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>addsi3</td>
<td>24, 44</td>
<td></td>
</tr>
<tr>
<td>andsi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>ashlsi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>ashrsi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>b&lt;code&gt;</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>call_value</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>casesi</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>clzsi2</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>cmpsi</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>epilogue</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>extend&lt;mode&gt;si2</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>extv</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>extzv</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>indirect_jump</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>iorsi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>jump</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>load_multiple</td>
<td>40, 44</td>
<td></td>
</tr>
<tr>
<td>lshrsi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>mov&lt;mode&gt;</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>movdi</td>
<td>24, 19</td>
<td></td>
</tr>
<tr>
<td>movsi</td>
<td>19, 23</td>
<td></td>
</tr>
<tr>
<td>mulhi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>mulsi3</td>
<td>44, 46</td>
<td></td>
</tr>
<tr>
<td>negsi2</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>25, 44</td>
<td></td>
</tr>
<tr>
<td>one_cmplsi2</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>pops</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>prologue</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>pushsi</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>s&lt;code&gt;</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>set_high</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>set_lo_sum</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>store_multiple</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>subsi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>tablejump</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>umulsi3</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>variable tracking</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>web construction</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>xorsi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>zero_extend&lt;mode&gt;si2</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>store_multiple</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>subsi3</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>tablejump</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>tail call elimination</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>tool chain</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>