Code Generator for SHARC ADSP-2106x

Peter Aronsson Levon Saldamli

Supervisor : Peter Fritzson

February 25, 1999

Abstract

This report describes the analysis, design and implementation of a code generator as a part of a C compiler for the SHARC ADSP-2106x digital signal processor from Analog Devices Inc. . It also describes the architecture of the SHARC processor, the compiler generator system CoSy and the back end generator BEG which is part of CoSy, which was used to generate the code generator. SHARC has a loop instruction to optimize loops, and support for parallel instructions using a VLIW architecture. Support for parallel instructions is implemented by using the instruction scheduler which BEG generates based on a description of the processor. The Loop optimization part consists of a loop analyzer and a hand written loop engine that generates statements for the loop instruction. The resulting compiler is then compared to the commercial compiler g21k from Analog Devices inc. The result is that the compiler is almost as good, and in some cases better than the g21k compiler. The compiler has also been tested regarding C language coverage, using test suits.

Preface

This master thesis is part of the WITAS project, which is an abbreviation for the Wallenberg laboratory for research on Information Technology and Autonomous Systems. The WITAS project has as a main goal to before year 2003, have "an airborne computer system which is able to make rational decisions about the continued operation of the aircraft, based on various sources of knowledge including pre-stored geographical knowledge, knowledge obtained from vision sensors, and knowledge communicated to it by radio" [1].

The SHARC processors will in this project be used to analyze pictures taken by a video camera on the aircraft.

The code generator, which is the result from this thesis will be used for compiling C code to SHARC assembler, using tha ACE ANSI-C frontend. However, there already exists a compiler for the SHARC processor from Analog Devices, called g21k. This compiler is, in the current version, not stable and crashes when optimizing certain programs. One reason for doing this compiler is to have a complement to the g21k compiler, as well as eventually achieving better optimized code.

Another reason, more related to research aspects, is that the back end can be used with the Java Real-time front end already implemented in CoSy, which is the tool used for implementing this code generator.

This master thesis begun with a two day course in CoSy, at ACE corporation in Amsterdam on the 22-23 of June 1998. This was two very intense days, and we learned a lot about CoSy, especially BEG, which is the back end generator in CoSy. Since incremental program development was used, we could start with a subset of the instruction set and then increase this set until we had a complete compiler.

After two months we had a working compiler. Then it was time to implement parallel instructions and loop optimization. This could be done in parallel independent of each other, which was an advantage since we were two.

Testing of the compiler was performed through the whole period, since we used incremental development; however running executable programs on the processor has been performed mostly at the end of the thesis.

We would like to thank the following people for their support (in no particular order):
Martien De Jong, ACE,
Job Ganzevoort, ACE,
Rob E.H. Kurver, ACE,
Marco P. Roodzant, ACE,
ACE Corporation in Amsterdam, Netherlands, for the course given to us, and
Peter Fritzson, our supervisor.

Peter Aronsson, Levon Saldamli.
Linköping, Sweden, February 25, 1999

Contents

List of Figures

Chapter 1

Introduction

This master thesis describes the design and implementation of a code generator for the ADSP 2106x SHARC processor from Analog Devices [2]. The code generator is implemented by using a back end generator, called BEG, which is a part of the compiler construction tool CoSy. A whole compiler has been implemented, by using an existing front end, which is a DSP-extended C-compiler front end. This front end supports specific DSP features in the C language. This is however not a part of the thesis, but only used to test the back end.

Chapter two contains the problem statement of the master thesis.

Chapter three is dedicated to describing the CoSy compiler construction tool. It explains the most important concepts in CoSy and the overall architecture of the tool. A more thorough explanation of BEG is also given, since this is the important part of this thesis.

Chapter four describes the processor. It explains the architecture of the processor, the instruction set, and the runtime environment used by the existing compiler from Analog Devices.

Chapter five gives an analysis on different aspects of the design and implementation.

Chapter six describes the design and the implementation of the compiler.

Chapter seven presents performance results of the compiler. Some comparisons with other compilers are also given.

Chapter 2

Problem statement

This chapter gives the problem statement of the thesis. The problem to be solved in this thesis consists of several parts, described in the sections below.

2.1 CoSy

One substantial part of this thesis work is to learn the CoSy system. This is a substantial part since the CoSy system is rather big and complex. In order to use CoSy and be able to write engines, at least three different languages need to be learned. First, the language called fSDL, which describes the internal representation of a program, must be understood. Perhaps it isn't so important to know every part of this language, but the basics is needed. Second, the structure of a compiler is given in another language, called EDL. This is not very complicated, and much of the structure is already given. The third language is the specification language given as input to the back end generator, BEG, which generates an almost complete back end.

2.2 The SHARC Processor

Another part of the problem is to learn the instruction set of the SHARC processor, and the overall architecture of the processor. Important properties of the processor is parallel execution of instructions, and loop instructions. These two topics should be analyzed and an optimized solution should be given.

2.3 Design and Implementation

The largest part of this thesis work will be design and implementation of the back end to the compiler. This will be done by using an incremental software development. This technique has been used previously in developing under CoSy, and has shown to be very useful and rapid. The advantages is that testing can be performed much sooner than compared to for instance the water fall model. This is good, since the earlier an error is detected the better.

The largest part of the implementation will be writing the Code Generator Description.

There are two parts of the thesis that are easy to separate. These are loop optimization and execution of parallel instructions. The parallel execution of instructions strongly connects to learning how to write the specification in BEG, where the architecture of the processor must be described by different resources that different instructions use. The loop instruction, however, requires knowledge in loop optimization and how to write engines in CoSy. These two tasks is separated and developed independently of each other.

One goal of the thesis is to produce a compiler that is better, i.e. produces more optimal code, than the existing compiler provided by Analog Devices.

To summarize, one can say that the problem statement in this thesis is to develop back end for the digital signal processor ADSP 2106x by using CoSy, that delivers optimized code regarding parallel execution and loop instructions.

Chapter 3

CoSy

This chapter describes different aspects of the compiler construction tool CoSy.

3.1 Overview

CoSy [3] [4] is a compiler construction tool for developing compilers for different platforms and architectures. It has been developed by Associated Computer Experts, ACE, in Amsterdam, Netherlands, and by partners in the Esprit project COMPARE. The first version was released in 1994. CoSy is designed for modularity, where a compiler is built of several engines, each one working on the internal representation of the program to compile. This makes the CoSy system very flexible and robust. It also allows reuse of components, which decreases development time substantially.

The internal representation used by the engines is called CCMIR, Common CoSy Medium level Internal Representation. CCMIR is a graph representation of a program and it is defined with a language called fSDL, full Structure Definition Language. The CCMIR is stored within the Common Data Pool, CDP, which all engines work against, see figure 3.1.

The structure of generated compilers is defined by a language called EDL, Engine Definition Language, which is used to describe the different engines in a compiler and in which order these engines should run.

3.2 fSDL and CCMIR

The fSDL language is used to describe the structure of the CCMIR, as well as access rights of engines working on CCMIR. When compiled, the result is a package for manipulating the CDP. This package is called DMCP, Data Manipulation and Control Package, and it makes sure that engines only access the data it specifies in it's *view* [5] [6].

The domain is a notion in fSDL for types, i.e. a type in fSDL is a domain. A Data type is represented by operators and fields. An operator can be compared

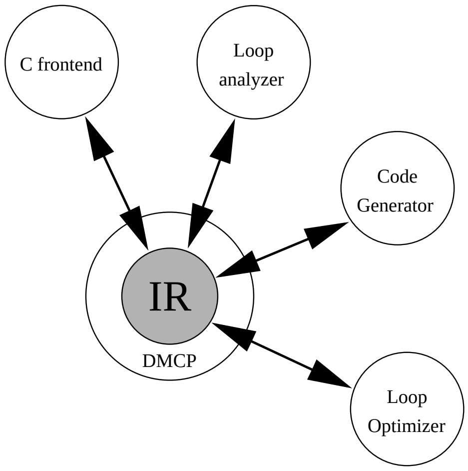


Figure 3.1: The CoSy compilation system. The engines works against the Internal Representation, IR, by accesing it through the Data Manipulation and Control Package, DMCP.

with a struct in C, and a field with the fields of a C struct. For example, to define the domain **mirOP2** having three *shared fields* namely `Left`, `Right` and `Strict`, and two operators **mirPlus** and **mirDiff**, one could write:

```
domain mirOP2 :
        {
                mirPlus,
                mirDiff
        } + <
                Left [primary]:         mirEXPR,
                Right [primary]:        mirEXPR,
                Strict :                BOOL
        >;
```

where the '+' operator means union. The fields `Left` and `Right` have the type **mirEXPR**, which in this case is a domain, meaning that these fields construct edges in the represented graph. The attribute on the fields, the primary attribute, state exactly that.

The **BOOL** type is an example of a special domain called an *opaque*. This means that it is user defined, and can be considered a domain without a known structure. Since probably all domains belonging to the **mirOP2** domain has the fields `Left`, `Right` and `Strict`, these are declared as shared.

The fSDL is as mentioned used to describe the CCMIR, and the *view*s of engines. A *view* is a domain with access rights added to it. Each engine can also extend the CCMIR with it's own domains. This makes the CoSy system very flexible, since new functionality easily can be added. For example, in order to handle loops in the SHARC processor, two new statements are created. These statements should belong to the domain **mirSimpleSTMT**. The fSDL code for this extension looks like this:

```
domain  mirSimpleSTMT:
        mirSTMTFlds + {
                xirLoopBegin <
                        IterCount [primary]:    mirEXPR,
                        Label           :       NAME
                >,
                xirLoopEnd <
                        Label           : NAME
                >
        };
```

The operator **xirLoopBegin** has two fields, **IterCount** and **Label**, holding the value of the number of iterations of the loop and a label for the last instruction. The operator **xirLoopEnd** has only the field **Label**. The two operators also has the shared fields stated in the domain **mirSTMTFlds** [5].

3.3 Engines

An engine is a module working on and updating the CCMIR tree. It can be all from a front end, that actually creates the tree, to a back end, that traverses the CCMIR tree and emits code for the output language. The fact that a compiler in CoSy is built of several engines makes the CoSy system very flexible. For instance, if a back end for a certain processor exists, compilers for different languages can easily be implemented by substituting the front end engine in the existing compiler. The same goes for developing compilers for different platforms, where only a different back end needs to be developed to use with the same front end. There are several engines delivered with the CoSy system, most of them optimizers of different kinds.

Engines can be composed by other engines, building a hierarchy of engines. There is also support for running engines in parallel, or in a loop. This can be useful when doing optimization, where several different engines perform optimization on the same code, and another engine chooses the best result. All this is done using the Engine Definition Language, EDL.

To construct an engine, here called myengine, three functions must be implemented:

```
state_p myengineCallInit(optionType);
myengineCallWork(state_p, ...);
myengineCallCleanup(state_p);
```

- `myengineCallInit` is called to initialize the engine. It includes creating a state struct, defining the state of the engine. This is necessary because engines cannot have any global variables. If global variables were allowed the system would be less robust. The solution is to put such variables in the state struct. Different options to the engines is passed to this function and placed in the state struct. For instance, for debugging purposes you can give a verbose option to the engine, stating how much verbose output you want from the engine.

- `myengineCallWork` performs the actual work. It takes a state pointer as an argument which is a pointer to the state struct.
- `myengineCallCleanup` does some cleanups after the engine has run. It typically frees the state struct memory from the CDP.

It is also necessary to state what the engine operates on, i.e. which arguments it takes. This is done in EDL. For example to write an engine called myengine that operates on the domain **myengineUnit**, which is a view of **mirUnit**, and uses the target description file, one could write:

```
ENGINE CLASS myengine(in myengineUnit, in myengineTarDes)
```

which declares a class from which several engines can be instantiated. This is another reason why global variables are not allowed. Since different engine instances share the same code, global variables cannot be used.

This engine can now be used by another engine. Putting it in a simple compiler can be done as follows:

```
ENGINE CLASS mycompiler (IN IR) [TOP]
        REGION  u: mirUnit;
        REGION  t: tarDes;
{
        frontend(u,t)
        myengine(u,t)
        backend(u,t)
}
```

A compiler is, as shown above, an engine composed of several other engines. The `mycompiler` has three engines, frontend, myengine and backend. All of them working on the top of the CCMIR tree, which belongs to the domain **mirUnit**, and the target description file, which contains target dependent information.

The target description file is a text file, stating the target architecture, for instance the instruction length, sizes of different types and memory structure.

3.4 BEG

BEG [7] [8] [3] is an abbreviation for Back End Generator. It takes a code generator description file, and generates a set of engines constituting parts of the back end of a compiler, see figure 3.2.

A complete back end consists of the engines in figure 3.2 and some additional engines, i.e. an engine for generating global data, an engine that rewrites the

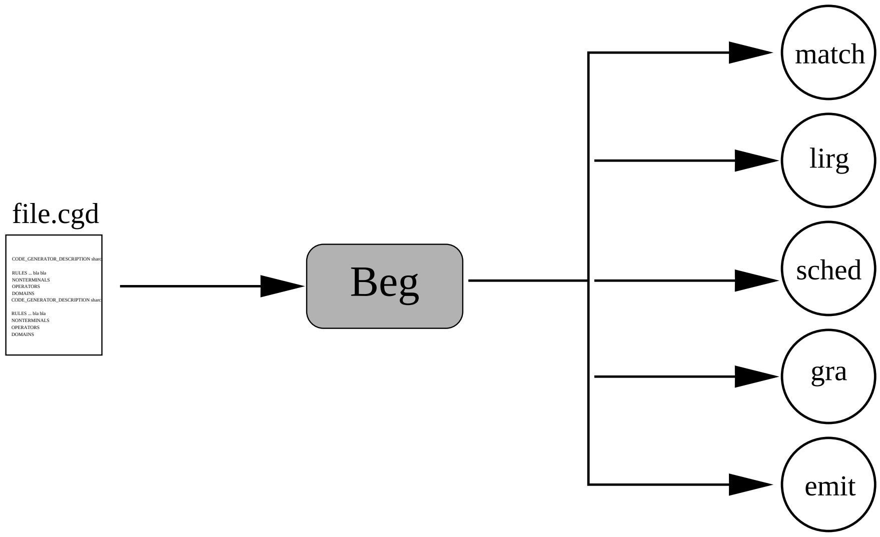


Figure 3.2: The Back End Generator. The match engine is responsible for instruction selection. Lirg is also part of this. The sched engine performs instruction scheduling. The gra engine is a register allocator and the emit engine is responsible for emitting assembler code.

CCMIR so that BEG can handle it, and engines for other back end specific tasks. The different engines in the figure are explained later.

The generation of code from the CCMIR is done in several steps, where each step corresponds to an engine. For example, the emit engine is responsible for emitting assembler-code to the output file.

The code generator description, the CGD-file, consists of several parts. First the *view* is stated in the IR-part. Then comes a register part, stating the registers of the target architecture and which of these are available to the register allocator. After that all non-terminals and their attributes are stated. Then comes some information to the scheduler and finally the most essential part, the rules.

3.4.1 Non-terminals

A non-terminal can correspond to, for instance a value stored in a register, but it can also be an addressing mode, or some other construct. How to choose the non-terminals is probably the most difficult task in writing a back end for a compiler. Choosing the right non-terminals makes the rules more simple to write and easier to understand. For instance, if the target processor has a set of registers for integer numbers, it probably would be logical to have a non-terminal called *reg*. This non-terminal then means: “a value or expression evaluated in a register".

There are four different kinds of non-terminals:

- MEMORY. This means that the rule does not need a register to put the result in. It is usually used for rules reducing to nothing, i.e. rules for an entire statement.

- REGISTERS. The most common kind,used to cover expressions returning results. This non-terminal puts the result from the rule in a register. For each application of this rule a register will be allocated, by the register allocator.

- ADRMODE. This kind of non-terminal is used when the non terminal doesn't describe an entire instruction, but just a part of it, namley the addressing mode. For example, if an address consists of an address and an offset, a non-terminal of type ADRMODE could describe this addressing mode.

- UNIQUE. This kind of non-terminals produce their result in some special location. Used, for instance, for instructions that modify condition flags.

A non-terminal can also have attributes. These are used to hold attributes from the underlying CCMIR tree. For instance, a non-terminal *reg* could hold an attribute `signed`, stating if the value in the register is signed or not. Attributes can be of two kinds, *condition attributes* and *attributes*. The latter kind is evaluated in the emit phase, i.e. when the emit engine is running. This means that they can't be used when doing the actual matching by the match engine. Therefore *condition attributes* are available. These attributes can be accessed in the CONDITION part of a rule, see section 3.4.2.

For instance, a non-terminal *reg*, containing an integer value, can have a condition attribute `size`, stating the size of the register, which could be byte or word. This must be a condition attribute since the rules probably look different for byte and word. If there is a rule for **mirPlus** that looks like:

```
RULE o:mirPlus(r1:reg, r2:reg) ->reg;
CONDITION {
IS_BYTE(r1) && IS_BYTE(r2)
}
EMIT {
/* Emit byte add */
}
```

If this approach is used, two or more rules has to be written for **mirPlus** taking *reg* non-terminals as operators. This can be done without condition attributes if different non-terminals for different sizes is used. For instance, *reg8* for byte and *reg16* for word. Then no condition attributes are needed. Again, the most important design issue is to choose the right non-terminals.

3.4.2 Rules

The rules is the central part of the CGD. Each rule specify a pattern which the match engine tries to apply to the CCMIR-tree. The goal is to completely cover the CCMIR-tree with rules. If a rule isn't a top level operator, i.e. it matches a pattern of a statement, it should produce a non-terminal. In other words, therule reduces a part of the tree to a non-terminal. If a non-terminal in enclosed in square brackets, the specified non-terminal contains an address that performs a memory access.

A rule can look like this:

```
RULE [bi_plus] o:mirPlus(r1:reg, r2:reg) -> r:reg;
COND {
        IS_INTEGER_OR_FIXED(o.Type)
}
TEMPLATE alu2;
COST 3;
EMIT {
        emit(myst, ADD, r, r1, r2);
}
```

This rule, with the name "bi_plus" reduces the **mirPlus**-operator to a *reg* non-terminal. The rule takes two *operands*, the *reg* non-terminals named `r1` and `r2`. The rule has a condition clause, stating a condition that must be fulfilled for the rule to match. In the example above, the type of the **mirPlus** node must be an integer or a fixed point. The example also shows the use of costs on rules. Since every rule can have a cost, the optimal code can be chosen by the matcher. If the rule reduces to a non-terminal of type register, there should also be an emit clause. The emit statement within the emit clause is a C-statement, emitting code to the assembler file. The file descriptor for the file is in the state structure, so it can be easily reached in all parts of the emit engine. All rule clauses are discussed more thorough in the Rule clauses section below.

There are some special rules for changing the CCMIR. This can be useful for transforming a node to another, closer to the target architecture. For instance, **mirDiv** can be rewritten to **mirShiftRight** if **mirDiv**'s right side is a constant with a value $x = 2^n$.

There are two different kinds of rules for rewriting:

- Trafo rules. The trafo rules is an integrated part of the matching process. This means that trafo rules can rewrite some node to another node. For example **mirPlus**(`r`: *reg*, **mirIntConst**), where **mirIntConst** have the value -1, can be rewritten using a trafo rule to **mirNeg**(`r`: *reg*, **mirIntConst**). Trafo rules are applied after matching.

- Rewrite rules. Rewrite rules are totally deterministic, i.e. if a rewrite rule is applicable to some pattern, it will be applied unless another rewrite rule with a lower cost is applicable. Rewrite rules are applied before matching.

To tie the non-terminals together, chain rules are used. These are rules which take a non-terminal and reduces it to another. They are used to, for instance, move values between different registers.

Rule clauses

A rule is constructed of a rule head, and some optional rule clauses. See figure 3.3 for a complete description of all rule clauses.

In rewrite and trafo rules only condition, cost, calc and eval clauses are allowed.

3.4.3 Pattern Matching

The code generator description file, CGD-file, describes the mapping of the CCMIR to assembler code using rules and non-terminals. The CCMIR tree is covered using rules, reducing subtrees to non-terminals, for which different assembler instructions are emitted. Figure 3.4 shows how the CCMIR-tree looks like for the statement:

```
x = y + 1;
```

where the variable x is global and the variable y is local. The task of covering the graph with rules is done by the match engine. Every part of the graph must be covered by a rule, reducing it to a non-terminal. In this example the **mirObjectAddr** node in figure 3.4 is covered by a rule reducing the subtree to a non-terminal called *dadrreg*. Then the subtree with top node **mirContent** is reduced to a **BegUsePsr**, using a *rewrite rule*. This non-terminal can then be transformed to a *reg* non-terminal using a *chain rule*. Next, the subtree starting with **mirPlus** can be covered using a rule that takes the *reg* non-terminal and an **mirIntConst** with value 1. This rule has an emit clause, not shown in the example, for the instruction:

```
Rx = Rn + 1;
```

This emit clause is handled by the emit engine. Finally, the entire statement can be covered using a rule taking the *daddr* non-terminal and a *reg* non-terminal. Since the **mirObjectAddr** was reduced to a *dadrreg* a chain rule must be used to transform it to the non-terminal *daddr*. The rule then emits the code:

```
DM(Mx,Iy) = Rn;
```

This code simply means, store the register `Rn` in data memory with base address `Iy` and offset `Mx`.

Note that since the top node of the tree corresponds to a statement, the last rule does not reduce to a non-terminal. This is obvious, since all code for the tree has already been emitted.

A strength with the matching algorithm used by the matcher engine is that each rule is given a cost. That way, optimizing can be performed in the code generation, and better code can be produced.

Clause	*Description*
CONDITION <text>	The condition to be fulfilled for the rule to match. The <text> must be valid C-expression because it will be a condition expression in an `if` statement.
COST <text>	The cost of applying the rule. The <text> can be any C expression.
CALC <text>	Specifies the calculation of attributes on the right hand side of a rewrite or trafo rule.
EVAL <text>	The code in <text> is executed when matching. Is used for evaluating conditional attributes.
SCRATCH <list>	When a rule needs an extra register, for instance to save some temporary result, this clause can be used. It tells the register allocator that it needs some register or registers of types specified in <list>.
CHANGE <list>	This clause is used to inform the register allocator that this rule changes the registers in <list>. For instance, a call to a procedure might change some registers. Then the rule reducing a **mirFuncCall** to some non-terminal should have this clause.
KILL <name>	<name> specifies a non terminal that defines memory, i.e. the non-terminal corresponds to a memory access.
BARRIER	The barrier clause can be one of PREBARRIER, BARRIER or POSTBARRIER. It informs the scheduler not to move code over this barrier.
PRODUCER	Specifies the producer class of the rule. See section 3.4.4.
CONSUMER	Specifies the consumer class of the rule. See section 3.4.4.
TEMPLATE	Specifies a template for the rule. See section 3.4.4.
LOCK	Tells the register allocator that the rule locks its registers.
UNIQUE	Tells the register allocator that the register of the resulting non-terminal should be different than the rules operand registers or scratch registers.
TARGET <name>	The <name> specifies the target register of an instruction, i.e. where the result is stored.
RESULT <name>	The same as TARGET, except that the rule doesn't change the result register.
EMIT <text>	The C code in <text> emits the assembler code to the output file.

Figure 3.3: The rule clauses.

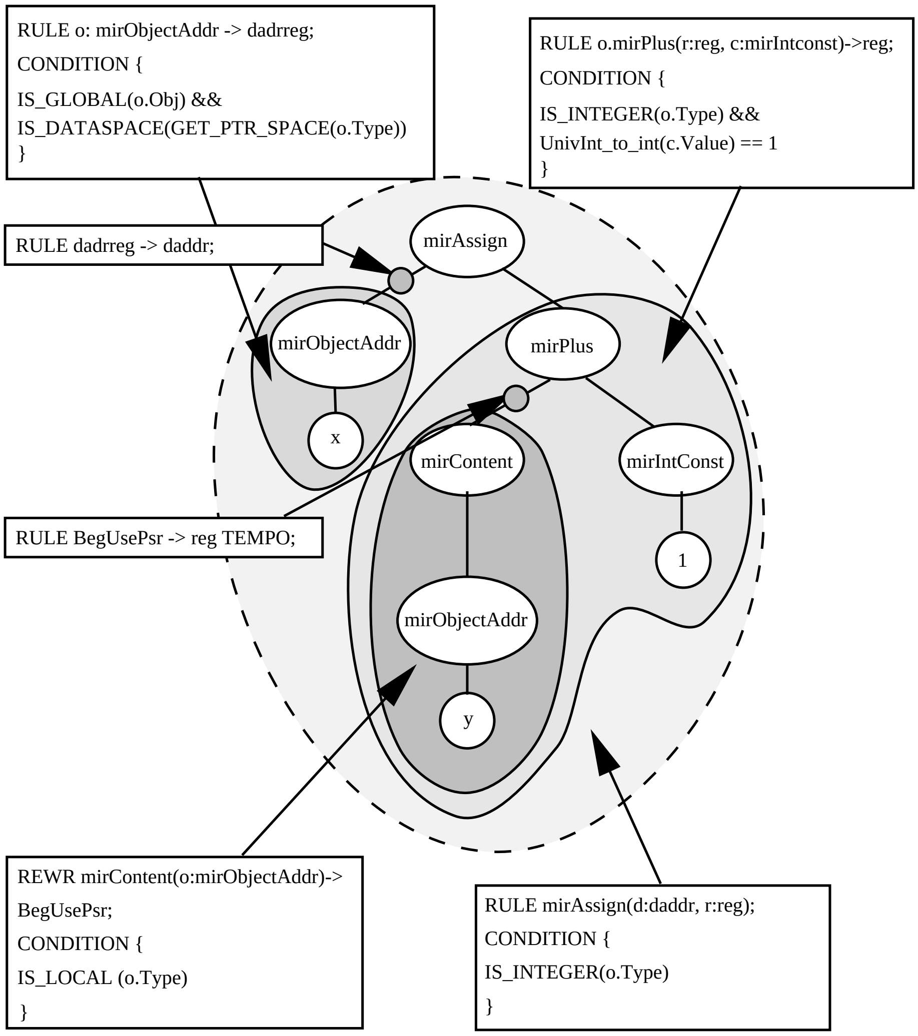


Figure 3.4: The Pattern Matching of rules on the CCMIR tree.

3.4.4 The Scheduler

The scheduler reorders the instructions to get an optimal instruction schedule. It operates on basic blocks and checks implicit dependencies in the cover tree and other dependencies, i.e. through memory access, to ensure that the resulting schedule is still correct.

Scheduler tries to select an order of the given instructions which executes in least possible time. The architecture of the processor is described in the CGD files to BEG, where different resources of the processor are declared. Where possible the scheduler will allow execution of several instructions during a single cycle. Section 5.1 on page 33 describes the operation of the scheduler in more detail.

The specification to the scheduler consists of a latency matrix, resources and templates:

- A Latency matrix is used to describe latencies between different consumer and producer classes, so that instructions can be scheduled before an instruction which will wait for a result.

- Resources and instances of resources are described to the scheduler so that instructions using different resources can be packed together.

- Templates are sets of resources used in different cycles of the execution. Each rule has a TEMPLATE clause which tells the scheduler what template it uses. The Scheduler then checks which resources are used on each cycle and packs instructions where possible.

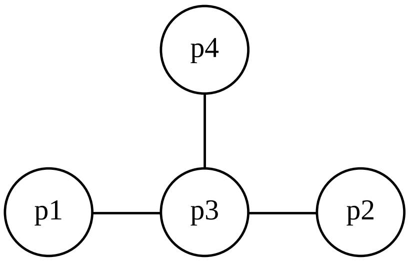
3.4.5 The Register Allocator

The register allocator in BEG is a global register allocator, i.e. it allocates register for entire procedures, and not just for single statements. BEG uses pseudo registers when allocating registers to different variables. A pseudo register is a value stored in a register. The pseudo registers are unlimited in numbers. It is then the task of the global register allocator, GRA, to assign a register to each pseudo register. To manage this task the allocator uses an *interference graph*. The nodes in an *interference graph* are pseudo registers, and the edges represent the *interference* between pseudo registers, i.e if they are both live at the same time. For example, if we have four pseudo registers p1, p2, p3 and p4, where p1 interfere with p3, and p3 with p2 and p4 the graph would look like figure 3.5(a).

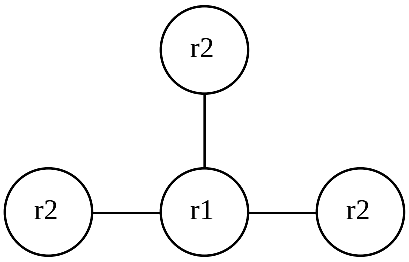
The GRA then allocates real registers for these pseudo registers using a graph coloring algorithm. For our example in figure 3.5(a) the allocation can be made using two registers, `r1` and `r2`, see figure 3.5(b).

In order to tell BEG when to use pseudo registers, two new operators are introduced. These are **BegUsePsr** and **BegAssignPsr**. When there is a **mirContent**(**mirObjectAddr**) in the CCMIR this is rewritten to **BegUsePsr** with a rewrite rule, and **mirAssign** is rewritten to **BegAssignPsr**, when the assigned variable is local. These operators have an operand called `psr`, with the domain **INT**, which is an *opaque*. The `psr` operand contains the actual pseudo register. Now, when the engine gra, the global register allocator, runs, it assigns a physical register to each pseudo register.

If the compiler should assign registers to global variables, things get a little more complicated. The reason for this is that the runtime environment of most computer architectures assume that all registers can change in a function. The solution is to save back all global variables from registers to the assigned memory when leaving the function. The same goes for entering a function, where you



(a) Interference graph



(b) Colored graph

Figure 3.5: A register allocation example

fetch the data from memory and put in the designated register. Normally, however, global data is not assigned to registers.

3.5 DSP-C Extension

The latest version of the ANSI-C front end in CoSy supports DSP[1]-extended C [9], with support for different memory types, the fixed point number type and circular arrays.

The different memory types are stated in DSP-C as a memory qualifier, for instance __D and __P for data and program memory. These qualifiers are defined in the target description file, see section 3.3. An example using these qualifiers is:

```
__D int * __P ptr;
```

which declares a pointer to an `int` stored in program memory, pointing to data memory.

The circular array type is expressed in a similar manner, with the `__circ` qualifier. For instance, to state that the pointer above is pointing to a circular array, the `__circ` qualifier must be added first in the declaration, as in:

```
_circ __D int * __P ptr;
```

Support for the type fixed point number is also added. A variable is of fixed point number type if the qualifier `__fixed` is used, as in:

```
__fixed var;
```

[1]DSP: Digital Signal Processor

Chapter 4

Sharc

4.1 Overview

The ADSP-2106x SHARC (Super Harvard Architecture Computer) [2] is a 32-bit digital signal processor with on-chip SRAM, I/O-processor and an external port. The core processor contains a program sequencer, instruction cache, two address generators, a data register file and three computation units, the arithmetic/logic unit (ALU), the multiplier and the shifter, see figure 4.1. It supports both 32-bit fixed-point and 32 or 40-bit IEEE floating point operations.

There are three buses on the SHARC, the PM bus (program memory), DM-bus (data memory) and I/O bus. The on-chip SRAM is divided into two blocks, data memory and program memory, which are used with the DM and PM bus to access two data operands on a single cycle, if the instruction can be fetched from the cache. The I/O bus and the fact that the SRAM is dual-ported makes it possible to perform a DMA-transfer during the same cycles as the normal operation.

The external port is an interface to external memory. It also contains interface to a host processor and a multiprocessor interface for communication with other SHARCs in a multiprocessor environment.

The compute operations are done on internal registers only. These are located in the data register file. The data register file has several ports to make accesses from several computation units possible in a single cycle. This allows parallel execution of two or more instructions, under some conditions. It is almost always possible to move data between registers and execute a compute instruction on one of the computation units in a single cycle. Given some register restrictions (see section 4.3.5) it is also possible to execute different compute instructions on separate units simultaneously, for example a multiply operation on the multiplier and an add operation on the ALU, and also move data between registers at the same time.

The SHARC processes instructions in three cycles, *fetch*, *decode* and *execute*. The instructions are pipelined in a three-step pipeline, which gives a throughput

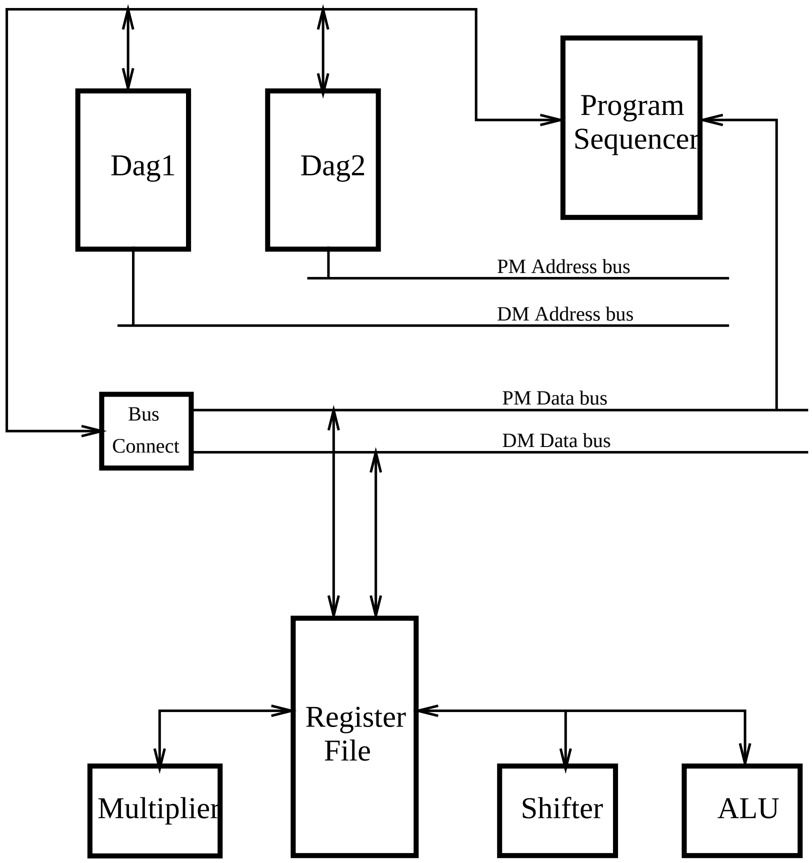


Figure 4.1: The architecture of the SHARC processor. This is only a block diagram of the core processor. The SHARC also has instruction cache, on chip SRAM, external port and an I/O processor.

of one instruction per cycle during sequential execution.

The special loop logic in the SHARC provides for efficient software loops with no overhead (for testing and branching). The loop logic works together with the pipeline to give maximum throughput.

4.2 Addressing

The ADSP-2106x's internal memory is divided into two blocks, data memory and program memory. These are accessed through separate buses to allow simultaneous transfer of instruction word and operand in a single cycle. Addressing memory is done indirectly, by using data address generators (DAG). There is one DAG for each one of the DM and PM address buses, called DAG1 and

DAG2. DAG1 generates 32-bit addresses on the DM address bus and DAG2 generates 24-bit addresses on the PM address bus, for accessing data and program memory, respectively.

Each one of the DAGs has a set of registers. There are four kinds of registers, *Index (I)*, *Modify (M)*, *Base (B)* and *Length (L)* registers. DAG1's registers are numbered 0-7 and DAG2's 8-15, i.e. I0 is a DAG1 address register and M9 a DAG2 modify register.

An I register contains a pointer to memory, and an M register contains a modify value used when accessing memory. The B and L registers are Base and Length registers and are used for modulo addressing (for circular buffers) and are explained in section 4.2.2.

One DAG cannot be used to access the other memory block, which means some registers cannot be used with the DM or PM operators. For example:

```
PM(I4,M9)       /* Illegal because I4 is in DAG1 */
DM(I4,M9)       /* Illegal because M9 is in DAG2 */
DM(I4,M4)       /* OK. */
```

Also, when moving data between a DAG register and memory, a register in the DAG used to generate the address cannot be accessed because this will cause incorrect results. For example:

```
B14 = PM(I9,M9);  /* Incorrect result because DAG2 is used in
                     memory access and B14 is a DAG2 register. */
```

The DAG register sets has also alternate (secondary) sets of registers. Which set is used is determined by mode flags in the processor. Each half of a DAG register set can be switched separately which allows passing pointers between alternate register sets.

4.2.1 Address Register Modification

There are two different ways of accessing memory, *pre-modify* and *post-modify*.

In pre-modify operation the address is determined by adding the modify value to the contents of an I register. Modify value can be contents of an M register or an immediate value. The contents of the I register is not changed in this operation. In post-modify operation the address is determined by the contents of an I register only. But instead the contents of the I register is modified by the given modify value after the memory is accessed. These operations are shown in figure 4.2. Pre-modify mode is used by writing the modify register before the address register, and post-modify mode is used by writing the address register first. For example:

```
DM(M0,I0);      /* Pre-modify */
PM(I9,2);       /* Post-modify */
```

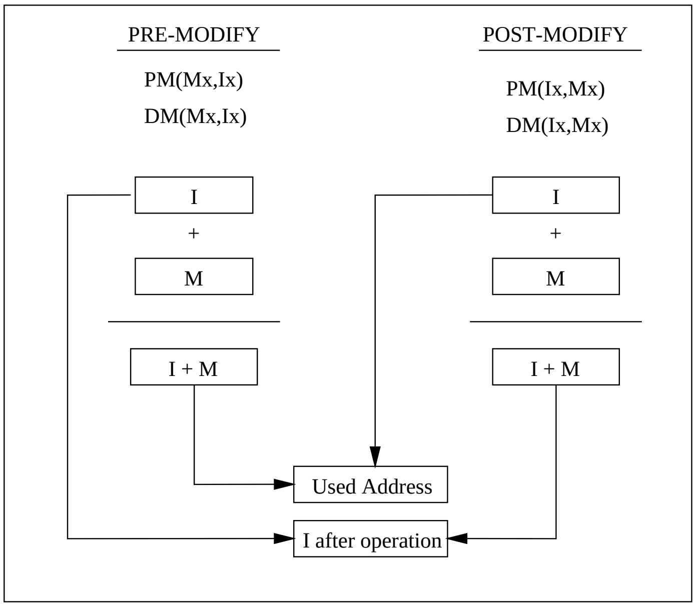


Figure 4.2: Pre-modify and Post-modify operations. Pre-modify mode is selected by writing the modifier before the address register, and means that contents of the address register and the modifier are added and the resulting address is accessed. Post-modify is selected by writing the modifier after the address register and means that only the contents of the address register is used to determine the address to access, but the address register is modified with the given modifier after the operation.

4.2.2 Circular Buffer Addressing

The ADSP-2106x allows modulo addressing, which is a hardware implementation of circular buffers. The `B` and `L` registers in DAG1 and DAG2 is used for this operation.

Each `I` register has a corresponding `B` and `L` register with the same number. In normal indirect addressing operation, `L` is set to zero (default). To use module addressing an `L` register `Lx` can be set to the length of the circular buffer, and the `Bx` register set to the start address of the buffer. Setting a `B` register loads the corresponding `I` register with the same value. After this is done, modifying the `Ix` register always results in a new address in `Ix` which is in the given interval, which starts at `Bx` and has the length given in `Lx`. Whenever the value of `Ix` gets bigger than start+length, the value wraps around. See figure 4.3 for an example addressing sequence.

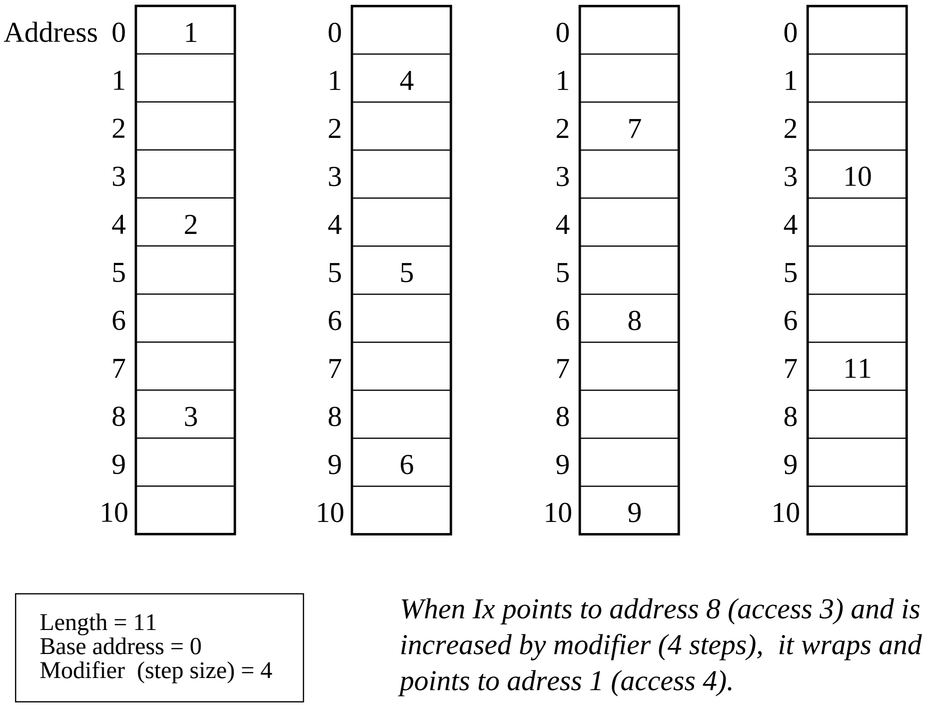


Figure 4.3: Circular data addressing

The modulo addressing does not work in pre-modify mode, because the `Ix` value is not modified in this mode. Checking of the limits and Wrapping around is done when the contents of `Ix` is to be updated.

4.3 Computation Units

In this section the three computation units ALU, multiplier and shifter are briefly described.

The computation units perform operations in both fixed-point and floating-point formats. The registers used are the same but the contents of them are interpreted different depending on the mode. The mode is determined by the prefix of the register name when writing assembler instructions:

```
F0 = F1 * F2;          /* floating point multiply */
R0 = R1 * R2;          /* fixed point multiply */
```

4.3.1 Arithmetic/Logic Unit

Except common arithmetic operations such as addition and subtraction and logical operations, the ALU supports averaging[1], min/max operations, clipping[2]and the absolute value operation. It reads one or two operators from the data register file and writes back the result. Reading is done on the first half of a cycle and writing is on the second, so the same register can be used both for reading and writing.

The ALU operates in different modes, to allow the user to have control over the operation. These modes can be seen in figure 4.4. The saturation[3]mode applies to fixed point operations and the truncation mode to both fixed point and floating point operations. The RND32 bit is set depending on whether 32 bit IEEE or 40 bit extended IEEE numbers are used. The truncation and RND32 modes affects the multiplier also, see section 4.3.2.

Bit name	*Set*	*Clear*
ALUSAT	Enable saturation	Disable saturation
TRUNC	Truncation	Round to nearest
RND32	Round to 32 bits	Round to 40 bits

Figure 4.4: Operation modes of the ALU

4.3.2 Multiplier

The multiplier reads two input registers and multiplies them and puts the result in a register. In floating point operations all the registers are data registers. In fixed point multiply operation the result can be accumulated in one of the Multiplier's internal registers MRF and MRB[4], which are 80-bit registers. Several consequent multiply results can then be accumulated in one of these registers.

[1]Averaging is a combined addition and division by two, in a single instruction.

[2]Clipping is used to limit the absolute value of a number, preserving its sign.

[3]Saturation means when register overflows, it gets the maximum value. When register underflows it gets the lowest value.

[4]There are two sets of registers in the SHARC, but only one set is available at the time and the set can be switched by modifying a flag. These sets are called foreground and background

In the case of fixed-point multiply, the format of the operands can be explicitly declared in the instruction. The format is given in a modifier with letters to specify whether the input is fractional or integer and signed or unsigned. For instance, the instruction

```
R3 = R1*R2 (SUI)
```

multiplies the signed integer value in R1 with the unsigned integer value in R2 and puts the result in R3.

If the format is omitted the default formats will be used. Default formats and the letters of the different formats are listed in figure 4.5.

Modifier	*Explanation*
(XYZZ)	X = S or U Y = S or U ZZ = I or F or FR
(SF)	Default for 1-input operations
(SSF)	Default for 2-input operations

X	: The first input
Y	: The second input (if any)
S or U	: Indicates corresponding input is signed or unsigned
I	: Integer inputs,
F	: Fractional inputs and
FR	: Fractional inputs and that output should be rounded.

Figure 4.5: Optional modifiers for multiplier fixed point operations

Depending on whether the input operands are fractional or integer numbers, different part of the MR-registers contain the result. Bits 63-32 contain the fractional result and bits 31-0 contain the integer result. Parts of the MR-registers are called MR2–0 and can be accessed and moved to data registers, see figure 4.6. There is also an instruction to clear the MR-registers.

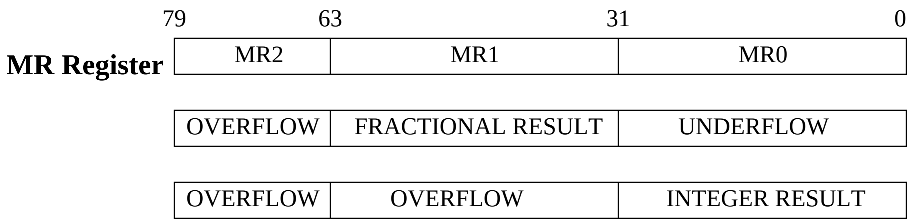


Figure 4.6: Result of fractional and integer multiplication in MR-registers

registers, hence the suffix F and B in the MR-registers. Although in the case of MR, both registers are available all the time.

Because these registers are part of the multiplier, accesses and operations on these are compute operations and not move operations, which means that these operations cannot be run in parallel with other compute operations as opposed to other register move instructions. See section 4.3.4 for details of parallel execution of instructions.

4.3.3 Shifter

The shifter supports left and right arithmetic and logical shifting, bit manipulation operations and bit field manipulation operations. It can take up to three input registers, and writes the result to an output register. All registers must be data registers. If three input registers are used, the first input register must be the same register as the output register. For example:

```
Rn = Rn OR LSHIFT Rx BY Ry
```

Logical shifting always inserts zero values from left or right when shifting. Arithmetic shifting inserts zero values when shifting left and repeats the sign bit instead of inserting zero when shifting right, to preserve the sign of the number.

The logical and arithmetic shift operations are the same for shifting left and right. The sign of the second operand determines shift direction, positive values shifts the source to the left and negative values to the right.

4.3.4 Parallel Execution

The ADSP-2106x supports parallel execution of instructions in different ways. Most instructions can be a combination of one *compute* and one *move* operation. Compute operations involve the computation units described earlier, and move operations move data between registers and between registers and memory. A *modify* operation can be done instead of a move operation. Modify operations modify the contents of address registers. However parallel execution limits the size of immediate offsets when addressing memory.

Moving data between memory and registers is done by using the address registers and modify registers or immediate offsets, and the DM or the PM operator, as in:

```
    Ra = Rx + Ry,   DM(Ia,Mb) = Ux;
```

where `Ia` is an address register, `Mb` a modify register and `Ux` any of the ADSP-2106x's registers. Here the compute operation is the addition and move operation is a move from a register to data memory. Modify register can be replaced by a 6-bit immediate offset, but then `Ux` is restricted to only data registers. This restriction also applies to shifting operations with immediate values. A move operation can also be between any two registers, as in:

```
    Ra = Rx - Ry,   Ia = Rb;
```

Immediate memory accesses or assigning immediate values to registers cannot be combined with a computation. Modifying address registers with immediate values cannot be combined with a computation either. The addressing is discussed in more detail in section 4.2.

4.3.5 Multifunction Computations

Besides the parallel execution of compute and move operations discussed in section 4.3.4, there are special instructions to do two simultaneous computations, using the ALU and the multiplier at the same time, or using dual functions in the ALU.

The multiplier can execute a multiply operation while the ALU executes one of add, subtract, average, fixed-point to floating-point or floating-point to fixed-point conversion, or floating-point abs[5], min or max operations.

Because the ALU and the multiplier needs to access the data register file at the same time when executing in parallel, they cannot have the same registers as operands. In fact different operands are restricted to different set of registers. How registers can be used in parallel is shown in figure 4.7. However the destination registers can be any data registers.

The ALU also supports dual functions, which can be simultaneous addition and subtraction, although these functions must use the same registers. For example:

```
Ra = Rx + Ry,   Rs = Rx - Ry;
```

where Ra, Rs, Rx and Ry can be any four registers. Dual functions can be combined with multifunction operations, so the multiplier can be used at the same time:

```
Fm = F3-0 * F7-4,  Fa = F11-8 + F15-12,  Fs = F11-8 - F15-12;
```

where Fm, Fa and Fs can be any registers and the registers used in addition and subtraction must be the same, one of F11 to F8 and F15 to F12, as seen in figure 4.7.

4.4 Program Sequencing

The SHARC processes each instruction in three clock cycles:

- **Fetch:** The instruction is read from instruction cache or program memory.
- **Decode:** The instruction is decoded and the processor state is modified to get ready to execute the instruction
- **Execute:** Execution of the instruction.

[5]abs is an instruction which computes the absolute value of its operand

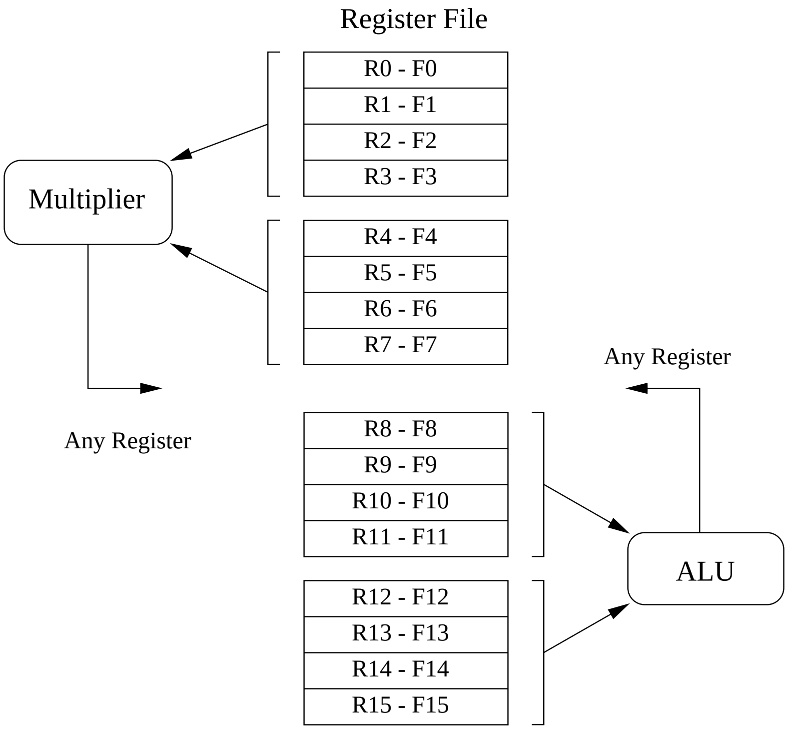


Figure 4.7: Input registers for multifunction computations

To increase throughput, there is a three-step pipeline in the ADSP-2106x, which means in the best case one instruction can execute in every clock cycle. The throughput can decrease if program execution is non-sequential, i.e. because of jumps and subroutine calls.

The program sequencer has three registers which contain addresses of instructions to execute, decode and fetch. The Program counter is the execute address register. The program counter is stored in a PC-stack when calling subroutines and looping.

Instructions are fetched from program memory, which is addressed by the data address generator DAG2 (see section 4.2), or from the internal instruction cache (see section 4.4.2). If the instruction can be fetched from the cache, then program and data memory can be accessed simultaneously for getting operands. Otherwise there is a conflict and the instruction fetch must be delayed one cycle. This decreases the instruction throughput, but can be avoided if the instruction resides in cache memory.

The sequencer also supports loops with no overhead which takes advantage of the pipelining. Loop termination condition is evaluated before it is time to jump to beginning of a loop and prefetch is done from beginning of the loop while executing the end of the loop. There is also a loop address stack and a loop counter stack to support nested loops.

4.4.1 Branches

Two instructions can cause a branch: `CALL` and `JUMP`. `CALL` also causes the PC to be pushed to the PC-stack so that the program flow can proceed after an `RTS` (return from subroutine).

Branches cause the prefetched instructions to become illegal, because the program flow is transferred somewhere other than the prefetched instructions were read from, which results in two delay cycles so that the new instruction can be fetched and decoded. To prevent this, delayed branches can be used. In that case the processor executes the two instructions following the branch while the new instruction is fetched and decoded. Also the return address in `CALL`s is adjusted in delayed calls so that execution continues in the correct place. See figure 4.8 for a comparison of delayed and non-delayed branches.

4.4.2 Instruction Cache

The on-board instruction cache of the SHARC has 32 entries. Each entry consists of a register pair, where an instruction and the address of the instruction is stored. There is also a bit for each entry to specify if the entry is valid or not.

The entries are divided into 16 sets, where each set has two entries. Each entry has also a bit called LRU (Least Recently Used) to mark which entry of the set has been used least recently. This is to decrease the number of cache overwrites caused by several instructions repeatedly stored at the same offset.

Because there are 16 sets of entries to store instructions, the four least significant bits of the address is used to map an instruction to the cache. These

JUMP INSTRUCTION:

Non-delayed jump

```
      INSTRa
      JUMP NEW
      INSTRb
      INSTRc
NEW:  INSTRd
```

Cycle	Instruction
1	INSTRa
2	JUMP NEW
3	-NOP-
4	-NOP-
5	INSTRd

Delayed jump

```
      INSTRa
      JUMP NEW (DB)
      INSTRb
      INSTRc
NEW:  INSTRd
```

Cycle	Instruction
1	INSTRa
2	JUMP NEW
3	INSTRb
4	INSTRc
5	INSTRd

CALL INSTRUCTION:

Non-delayed call

```
      INSTRa
      CALL SUB
      INSTRb
      INSTRc
      INSTRd

SUB:  INSTR1
      RTS
      INSTR2
      INSTR3
```

Cycle	Instruction
1	INSTRa
3	CALL SUB
4	-NOP-
5	-NOP-
6	INSTR1
7	RTS
8	-NOP-
9	-NOP-
10	INSTRb

Delayed call

```
      INSTRa
      CALL SUB (DB)
      INSTRb
      INSTRc
      INSTRd

SUB:  INSTR1
      RTS (DB)
      INSTR2
      INSTR3
```

Cycle	Instruction
1	INSTRa
3	CALL SUB
4	INSTRb
5	INSTRc
6	INSTR1
7	RTS
8	INSTR2
9	INSTR3
10	INSTRd

Figure 4.8: Delayed and non-delayed branch. The left columns show code which does not use delayed branch, and the right columns show code which does. The tables below each part of code show what instruction is executed during consecutive cycles. `NOP`'s (No OPeration) are inserted automatically by the processor when it shouldn't execute any instruction.

four bits are not stored together with the address, because they are implied by the position of the instruction in the cache.

Each time an instruction is to be fetched, four least bits of the address is used to check the cache entry. If the address matches one of the two entries in the set, a cache hit occurs, and the LRU bits of the entries are updated.

Thanks to the double storage, up to two instructions can repeatedly be mapped to the same cache index without decreasing hit rate. If three or more instructions in a loop has the same four lower bits the cache usage gets very inefficient. This can be avoided by moving one of the instructions to another address.

4.4.3 Loops

Loops in the ADSP-2106x's program sequence can be implemented with the `DO-UNTIL` instruction. This instruction provides loops with no overhead, which otherwise means extra cycles for checking loop termination condition, modifying a counter and branching to the beginning of the loop.

The program sequencer stores information about the loop in stacks to support nested loops. The stacks have space for information about six loops, which makes six levels of nested loops easily possible. The address of the first instruction of the loop is stored in the PC-stack, address of the last instruction, loop type and the termination code are stored in the loop address stack and the loop counter is stored in the loop counter stack.

To take full advantage of the three step instruction pipeline, the loop termination testing is done two instructions before the last instruction of the loop (at location $e-2$ where e is the end-of-loop address). If the condition is false, the first instruction of the loop is fetched, otherwise the instruction following the last loop instruction is fetched and the loop stacks are popped.

Loops can be counter-based and non-counter-based. A counter-based loop can look like this:

```
        LCNTR=30, DO label UNTIL LCE;
        INSTR1;
        INSTR2;
label:  INSTR3;
        INSTRa;
        ...
```

where the `INSTR1-3` is part of the loop and executes 30 times. `LCNTR` is the loop counter and is initialized to number of times the loop should execute. `LCE` is the termination condition of the loop which means Loop Counter Expired. The loop counter `LCNTR` is decreased two instructions before the last loop instruction, because the test is done before the last two instructions execute. Therefore this loop counter is already decremented when the last two instructions of the loop executes.

Non counter-based loops are basically the same. The `LCNTR` initialization is not present and the termination condition is some common condition flag other than LCE. For example:

```
        DO label UNTIL NE;
        INSTR1;
        INSTR2;
        INSTR3;
label:  INSTR4;
        INSTRa;
        ...
```

where the loop keeps executing until the condition NE becomes true (Not Equal to zero, which occurs when the zero flag is false). Here too, the condition is checked two cycles before the last instruction of the loop.

4.5 The SHARC Runtime Environment

The runtime environment of the SHARC includes the usage of the stack, parameter passing to functions and how function calls are performed. Analog Devices has developed a runtime environment for the SHARC processor. This will be fully used, to ensure compatibility with software written and compiled by other compilers.

Function calls will be implemented by putting the return address in the stack frame, and jumping to the function being called. Exiting the function is done by jumping to the return address previously put in the stack frame. When a function is called, the run time environment assumes that the registers in the following set is unchanged:

```
r3,r5,r6,r7,r9,r10,r11,r13,r14,r15,
i0,i1,i2,i3,i4,i5,i8,i9,i10,i11,i14,i15,
m0,m1,m2,m3,m8,m9,m10,m11,
mrf,mrb,mode1,mode2,ustat1,ustat2
```

The rest of the registers change when performing a function call. This means that the registers belonging to the unchanged set has to be stored on the stack if they are used inside a function.

The runtime environment supports programming in the C language, which is exactly what we want. This means that a function's return address is stored on the stack, i.e. the runtime environment support an arbitrary level of function nesting. Parameters to functions are passed in registers or on the stack. The stack lies in data memory and also contains local and temporary variables. A typical view of the stack is shown in figure 4.9.

The parameters of a function is passed according to the following rules:

- Up to three arguments are passed in registers.
  The first argument is passed in register r4, the second in register r8 and the third in register r12.

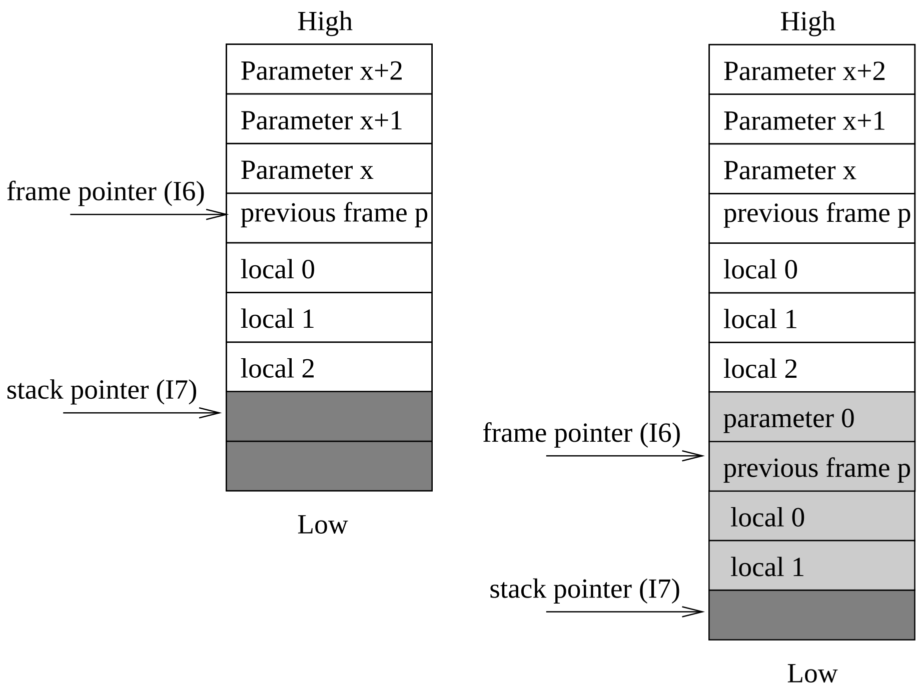


Figure 4.9: The stack configuration. To the left is the stack when inside the main function, to the right when inside a function called from main.

- Once an argument has been passed on the stack, all remaining arguments to the right are also passed on the stack.
- All arguments bigger than 32 bits are passed on the stack. This includes for instance structures passed by value.
- The last named argument in a function call with variable number of arguments is passed on the stack.

When arguments are placed on the stack, they are pushed there from right to left, see figure 4.9.

Chapter 5

Analysis

This chapter covers the analysis part of the thesis, and discusses important issues like parallel execution of instructions and how to optimize loops, by using special loop instructions in SHARC.

Another important part of the analysis is finding out how to use the post-modify addressing mode, which is also discussed in this chapter.

5.1 Parallel Instructions

The SHARC supports execution of instructions in parallel and this should be supported by the compiler for better performance. The scheduler generated by BEG can be configured as an instruction packer, which can be used to implement parallel instructions. The scheduler specification part of the CGD describes the processor to the scheduler so that it can pack instructions.

5.1.1 The Scheduler

Definition 5.1.1 *Given sets of producer and consumer classes, a* ***Latency Matrix*** *is a specification of latencies between these classes.*

Each rule in the CGD description can have a `PRODUCER` or `CONSUMER` clause which tells the scheduler which class the generated instructions belong. For example:

```
RULE o:mirMult (f1:freg, f2:freg) -> fr:freg;
PRODUCER FpuOut;
EMIT {
        fprintf(FILE,   "%s = %s * %s", REGNAME(fr),
                                        REGNAME(f1),
                                        REGNAME(f2));
}
```

This instruction is a multiply of two floating point registers.The producer class is defined to be `FpuOut`. This way, if extra cycles are needed by the processor to produce a floating point result, a latency matrix, as defined in definition 5.1.1, can be written to tell the scheduler about it. The latency matrix might look like this:

```
LATENCIES        DEFPROD AluOut   FpuOut;
        DEFCONS 0       0        5;
```

where the latency of a fpu result is said to be 5, and that there is no latency after an `alu` operation. `DEFCONS` and `DEFPROD` classes are special and are used in all rules where no classes are explicitly given. This example has no consumer classes defined, except the default consumer class `DEFCONS`. Therefore the matrix only contains one row.

Definition 5.1.2 *A* ***Resource Template*** *is a set of resources of the processor an operation uses in each cycle of execution.*

A list of resources are defined in the scheduler description. Resources can have several instances, which are then listed next to the resource. A resource template is a list of elements where each element can be a resource, a specific instance of a resource or a list of resources, as defined in definition 5.1.2. For example:

```
RESOURCES
        Decode,
        Alu,
        Fpu,
        Read (rd1, rd2),
        Write;

TEMPLATES
        aluop   Decode Read Alu Write;
        fpuop   Decode Read 2*Fpu () () () Write;
        bothop  Decode (rd1 rd2) (Alu Fpu) Write;
```

Here the `aluop` template uses four cycles, the `fpuop` template eight cycles and the `bothop` template four cycles. Several consecutive cycles using the same resource can be written as n*resource, where n is the number of repetitions, i.e. `2*Fpu` means that two cycles using the `Fpu` resource. Empty parenthesis denote cycles where no resources are used. Multiple resources used on a single cycle can be written inside parenthesis, i.e. `(Alu Fpu)` means both the `Alu` and the `Fpu` resource are used on the same cycle. In the `aluop` and the `fpuop` template the resource `Read` is given, which means either the instance `rd1` or `rd2` can be used, but in `bothop` both the `rd1` and `rd2` instances are used in the second cycle.

5.1.2 A Scheduler Description For The SHARC

Compute and move

The SHARC can execute one compute and one move operation in most cases, and sometimes one compute and two move instructions. For this, two resources called `Move` and `Compute` are defined.

- **Compute** resource has three instances, one for each of the computation units `Alu`, `Multiplier` and `Shifter` (see 4.3 on page 22).
- `Move` resource has two instances, `mv1` and `mv2`, because there are two address generators (DAGs) and two blocks of memory which can be accessed simultaneously.
- `dm` denotes the data memory accesses and
- `pm` denotes the program memory accesses.

This gives the following set of resources:

```
RESOURCES
        Compute (Alu, Multiplier, Shifter),
        Move    (mv1, mv2),
        pm,
        dm,
        noshiftimm;
```

There are some combinations of `compute` and `move` which are not allowed. To implement this kind of restrictions, virtual resources can be defined and included in some of the templates, i.e. the `noshiftimm` resource above is used to prevent immediate shift and some move operations from being packed together. This happens because the packer treats this resource as being reserved by an operation. Therfore other operations using it cannot execute at the same time.

Multifunction computations

As discussed in section 4.3.5 on page 25 a multiply operation and some `ALU` operation can be executed in parallel if they use distinct set of registers. Optimizing the code for this is a problem when using BEG, because the match engine that generates a match tree is run before the register allocator, so the physical registers that actually will be used is not known during matching. Because of this, different rules with different costs and with register conditions cannot be used, which otherwise would be a good solution.

One way to solve this problem is to inform the scheduler about the possibilities of packing together these instructions after the registers have been determined. The templates for a rule can be determined dynamically, and if the scheduler is configured to run after the register allocator, selected physical registers are accessible for the code that selects a template for the rule. This will allow selection of an appropriate template to enable packing of instructions if the registers selected are correct.

However, this is not an optimal solution. There is no way to tell BEG that it should prefer to choose some sets of registers so that the packer will be able to pack some instructions. A register is either allowed or not. The register allocator will blindly select some registers for these instructions, and if it happens to be a correct set of registers, packing will be possible.

To restrict the available registers for a rule is possible, which is another way to solve the problem, but this will limit the register allocators choices, possibly generating spill code.

The solution chosen in this thesis is to allow a command line option, to choose between the two alternative solutions above. If the option is given, an identical set of rules with cheaper costs are used, which restricts the available registers. If the option is not given, no check of registers are done.

5.2 Analyzing And Optimizing Loops

Since SHARC has special instructions for managing loops, and loops are very important to optimize because they are used very often in applications for DSPs, makes optimization of loops a high priority to implement in the compiler.

First, some definitions need to be done.

Definition 5.2.1 *A* ***Basic block*** *consists of a sequence of statements, containing no jumps, except for the last statement in a basic block [10].*

Definition 5.2.2 *The* ***Control Flow Graph*** *is a graph in CCMIR describing the program flow, where the nodes are basic blocks. There is a directed edge between B1 and B2 if B2 follows B1 in the program flow [5].*

Definition 5.2.3 *A* ***Loop*** *is a set of basic blocks, S, containing a header basic block h, where*

> *From any basic block, there is a path leading to h.*
>
> *There is a path from h leading to every block in S.*
>
> *All paths from a basic block outside S, leading into S, goes through h.*

As an example study the C program in figure 5.1. A CoSy front end compiles this code into the internal structure, the CCMIR-tree. Parts of this tree contains the control flow graph, the CFG. The CFG is, according to definition 5.2.2, a directed graph, where the nodes are basic blocks (definition 5.2.1). The CFG for the example in figure 5.1 is shown in figure 5.2. The loop in the program, as defined in definition 5.2.3, consists of basic blocks `bb5` and `bb6`. The header basic block of the loop is `bb5`, since `bb6` has a path to `bb5`. And `bb5` has a path leading to `bb6`. There are also no other paths to `bb6` from some basic block outside the set of basic blocks constituting the loop.

```
x=2+y;
if (x == 0) {
  x++;
  foo(y+x);
}
else {
  for (i=0;i<10;i++) {
      arr[i]=x;
   }
}
```

Figure 5.1: A loop.

In figure 5.2 the CFG contains two special nodes. These are the first basic block in a procedure and the last basic block in a procedure. These basic blocks contains only one statement, beginprocedure vs. endprocedure, and are used to be able to determine the start and end of a procedure. In CCMIR these statements are called **mirBeginProcedure** and **mirEndProcedure**.

5.2.1 The Loop Analyzer

Since the ANSI-C front end does not translate loops into the constructs available for loops within CCMIR, this has to be done afterwards. Therefore an engine which analyses the CCMIR-tree and finds loops has to be run. The alternative is indeed to detect loop constructs in the input language, for instance a C-program containing a `for` statement. However, there is a great advantage in letting an algorithm detect the loops, by analyzing the CCMIR-tree, since then, it is possible to detects loops that aren't explicitly stated to be so. For instance if you use the `goto` statement in the C language, it is possible to construct a loop without using the `for`- or `while`-statements. Another advantage is that after some transformations to the CCMIR, new loops could have been introduced. Then these can be detected too, and further optimizations is possible.

CoSy contains a loop analyzer that detects all loops in the control flow graph. It annotates each **mirProcedure** in CCMIR with a functor called **LOOPS**. [ref] It contains all loops in the procedure. Each loop, described by the domain **mirLoopMarker**, have some attributes. For instance all loops have an init basic block, a pre-header basic block, a test basic block, a body basic block, a increase basic block, and a list of loop variables. **mirLoopMarker** also has a loop kind, stating the type of the loop. It can be a `whileDo` loop, which means that the test is performed first, and if it evaluates to true, an iteration of the loop is performed. The other type is the `repeatUntil` type. It has the test expression last in the loop, i.e the loop is always performed at least once.

The list of loop variables is the most important information in the **mir-LoopMarker**. A loop variable can be of four different types:

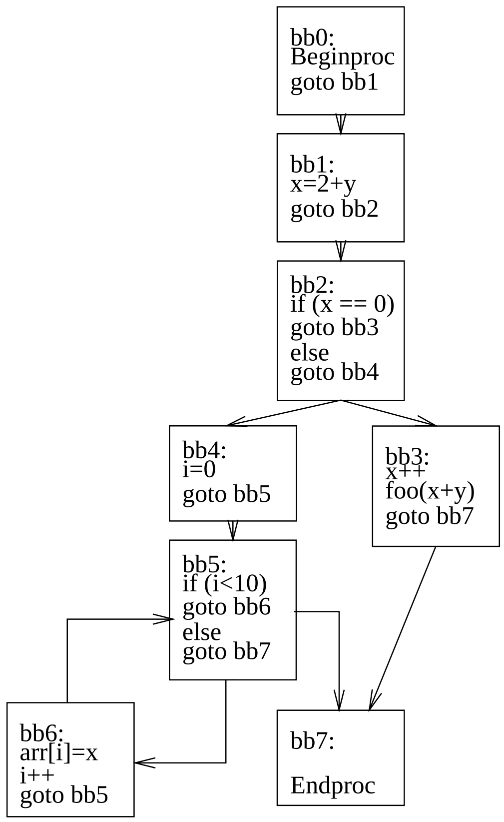


Figure 5.2: A control flow graph for the C-example in figure 5.1.

- A **mirLoopVar** is a variable which is updated once in each loop iteration. The variable does not need to determine the end of the loop.

- A **mirBasicIndVar** is a basic induction variable, i.e. it has an expression on the form:

  ```
  v = v + <exp>
  ```

  where `exp` is an arbitrary expression. The variable does not need to determine the end of the loop.

- A **mirLoopControlVar** is a basic induction variable like above, and is used to test the end of the loop.

- A **mirIterCounter** is a basic induction variable on the form:

  ```
  v = v +/- 1
  ```

The test of the loop should be a single expression. This loop variable also has information about the number of loop iterations.

5.2.2 The DO-UNTIL Instruction

The SHARC has hardware support for executing loops without flushing the pipeline, which means that the evaluation of the test in the loop is executed in advance. This makes it possible for the SHARC processor to know which instructions to read and decode, so that the pipeline will not be stalled. The loop instruction has the syntax:

```
LCNTR = <value,register>, DO <label> UNTIL <cond>
```

The condition can be any condition from the status register, but the most usual is the `LCE`-condition. It terminates the loop when the loop counter, `LCNTR`, reaches zero. The `label` is the label for the last instruction in the loop.

To be able to emit code for this loop instructions, two statements has to be added to the CCMIR. One that constitutes the beginning of the loop, and one corresponding to the last instruction of the loop.

The statement that constitutes the beginning of a loop, called **xirLoop-Begin**, needs a field of type **mirEXPR**, which is an expression evaluating to the number of iterations in the loop. This field is referred to as `IterCount` in the rest of this thesis. When the number of iterations is constant, the problem is easy, since then the `IterCount` is a **mirIntConst** containing the constant value. Otherwise, if the value is not a constant, the problem can be solved if we know the test expression, the update expression and the init expression of the loop control variable, and if they have the following form:

init expression: $\texttt{var} = \texttt{c}$

update expression: $\texttt{var} = \texttt{var} \pm 2^n$

test expression: $\texttt{var} < \texttt{val}, \texttt{var} > \texttt{val}$

where `c` is an integer constant, $\texttt{n} \geq 0$ is an integer constant, `var` is the loop control variable, and `val` is a value that can be evaluated before the loop starts. If these conditions are met, the `IterCount` can be calculated. The easiest case is of course when n equals to zero, i.e the loop variable increments, or decrements, with the value one in each iteration.

The `IterCount` expression can be evaluated in a register which is used to initialize the loop counter. The `IterCount` will then have the value $|c - val|/2^n$. The division by 2^n can be expressed by right shifting n steps. This makes it much more efficient than an ordinary division. It is not certain that this method will be optimal if any denominator is allowed. It can be more optimal if the size of the loop is very large, however a loop which is both incremented by a number n where $n \neq 2^k, k \geq 0$, that also iterates many times isn't that usual. Therefore it will not be considered further.

5.2.3 Further Optimizations

So far, no *real* loop optimizations have been discussed. Many different optimizations can be performed on loops, for instance induction variable eliminations and moving of loop independent code. However, these are rather complex algorithms that takes to long to implement. They will also probably be available in future versions of CoSy, therefore they will not be discussed or implemented in this thesis work.

But, since the SHARC has the post-modify addressing mode, which is extra useful in loops, an attempt to implement an algorithm for detecting when this addressing mode can be used will be discussed. The idea is to find all pointers in a loop and rewrite them, so that they support this addressing mode. The pointers that can be rewritten are pointers to an array, where the elements are visited in a sequence. For instance, consider the loop in figure 5.3. This loop visits all elements in an array, and sets the value to 5. The statements in the body of the loop is shown in figure 5.4. The first statement in the figure shows that the address pointing to the element at position `i` in the array **`arr`** is calculated by adding the address of **`arr`** by the number `i`. If different sizes are allowed in the target architecture, for instance byte and word arrays, there would also be a **mirMult** node in the CCMIR tree, which multiplies the variable `i` with a constant.

```
for (i=0; i<10; i++) {
   arr[i] = 5;
}
```

Figure 5.3: A simple loop written in C

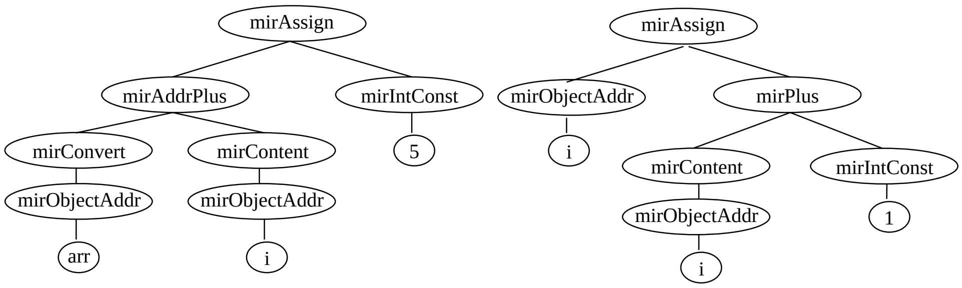


Figure 5.4: The CCMIR-tree for the two statements in the loop body.

In order to use the post-modify addressing mode we would instead like to have the structure in figure 5.5. In the figure it is obvious why post-modify can be used. First, the pointer to the array is referenced, then it is incremented. A note might be that the semantics of the program have been changed between figure 5.4 and figure 5.5. For instance, the variable `i` has changed type from an integer to a pointer, with reference type integer.

The big problem with this approach is that BEG uses pattern matching on statements. Since there are two statements above describing one instruction, namely the memory access instruction with post-modify addressing mode, this approach can be very hard to implement in BEG, by using rules. This is because BEG assumes that each statement should produce at least one instruction, and therefore no non-terminals can be changed from one statement to another.

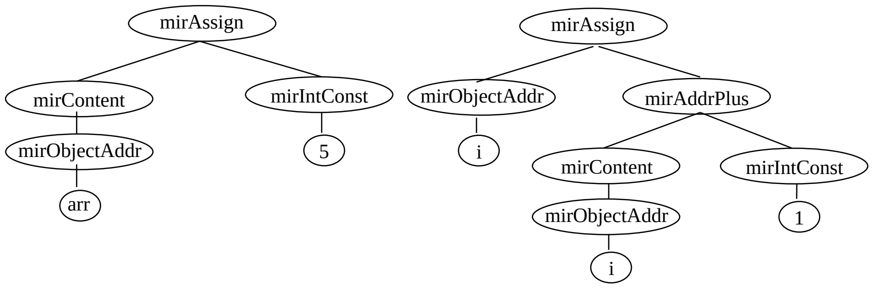


Figure 5.5: The CCMIR-tree for two statements that makes post-modify possible.

A solution to this problem can be to introduce a new statement in CCMIR, for instance **xirModify**. It should have the following structure:

```
xirModify <
  Obj : mirObjectAddr,
  Expr  : mirEXPR,
  Value : mirEXPR
>
```

The `Obj` field contains the pointer to perform this post-modify addressing operation on. The `Expr` evaluates to the value that should be stored at the location indicated by the pointer. The `Value` should be a **mirIntConst**, since it simplifies the implementation, but it could also be an arbitrary expression. When this new statement has been introduced it is an easy task to implement rules for emitting code for this statement. It will probably be sufficient to have one rule, taking one or two register non-terminals and emitting code for the assignment.

The gain of implementing support for this addressing mode can in certain cases be rather big. For instance the loop example in figure 5.3 will without using post-modify would look like in figure 5.6.

However, if the post-modify mode would be used, the code could be much faster, as shown in figure 5.7.

The advantages is obvious. First, several lines have been moved to before the loop. For instance, the initialization of the modify register now lies before the loop. Second, The `r0` register isn't needed anymore. However, this can be eliminated only if the variable `i`, in our example, isn't referenced within the loop. If it is referenced, the variable is still needed.

```
r0 = 0;

_L1:    LCNTR=10, DO __sll0-1 UNTIL LCE;

_L2:    m0 = r0;
        r1 = 5;
        i0 = i6;
        MODIFY(i0,-10);
        r0 = r0 + 1,            DM(m0,i0) = r1;
__sll0:
```

Figure 5.6: Assembler code for a small loop.

```
r0 = 0;
i0 = i6;
        MODIFY(i0,-10);
m0 = 1;
_L1:    LCNTR=10, DO __sll0-1 UNTIL LCE;
        r1 = 5;
        DM(i0,m0,) = r1; /* post-modify */
__sll0:
```

Figure 5.7: Assembler code for a small loop, using post-modify addressing mode.

Of course, this suggestion is only a special case of optimizing by using post-modify addressing mode. But the idea can without to much work be generalized so it supports a larger subset of all loops. What has to be done then is to implement an algorithm that finds the last reference to the array in an loop and rewrites that one. This will not be implemented in this thesis due to lack of time.

Chapter 6

Design And Implementation

The main problem when starting to write a back end in BEG is to find the appropriate non-terminals. This can be done in many different ways. None of the approaches has to be wrong, but the choice and design of non-terminals can affect the number of rules that has to be written.

The main effort in the thesis is to write the rules. The rules are many in number since the SHARC is a complex processor with many different kinds of registers and complex instructions. In addition to this, several engines has to be written, for instance, the datagen engine, which is responsible for generating global variables. The engine lower, that rewrites parts of the CCMIR-tree so that BEG can read it, must also be written. These parts are described further below.

6.1 The Non-terminals

The non-terminals has been designed incrementally. That means, we started with a small subset of the non-terminals, and wrote rules for these. Then we added additional non-terminals supporting further instructions and program structures. This has meant some redoing of the non-terminals, but that has probably made the design better.

6.1.1 Design Decision

The approach on designing non-terminals has been to avoid having any attributes on the non-terminals. Instead, when we have discovered that an attribute could be useful, this non-terminal has been split into several, each on corresponding to a some value of the attribute. For instance, an pointer to the data memory can either be a pointer to a circular array, or a pointer to some other data. These two cases must be dealt with separate. We have then introduced two non-terminals for this register. These are, described in CGD-style:

```
dadrreg         REGISTERS<regid>;
```

```
dcircreg        REGISTERS<regid>;
```

The first one, `dadrreg`, means a pointer to some data in memory, not a circular array, that is stored in the DAG1 address register file. The second one is thus the same except that the pointer points to a circular array, see 4.2.2 on page 21.

The main reason for this design decision, is that this approach makes the rules easier to write, and to understand. It also makes it easier to write chain rules, since the attributes doesn't have to be considered.

6.1.2 The SHARC Non-terminals

Here follows a complete list of the non-terminals for the SHARC back end. All non-terminals of type `REGISTERS` is shown in figure 6.1. Figure 6.2 contains all `ADRMODE` non-terminals along with the rest of the non-terminals, such as those of type `UNIQUE`.

6.1.3 Chain Rules

The chain rules are used for "connecting" a non-terminal to another. The chain rules can be described using a directed graph, where the nodes are the non-terminals, and the edges is defined as follows: If a chain rule exists from non-terminal A to non-terminal B, there is an edge from node A to node B. The chain rules are described using this graph in figure 6.3.

Name	*Type*	*Description*
reg	REGISTERS <regr>;	A value stored in any of the data registers, can be both integer/fixed point or float.
returnval	REGISTERS <r0>;	The data register r0, which is used for the return value of a function call.
argreg	REGISTERS <r4,r8,r12>;	One of the data registers that are used for passing parameters to a function.
dadrinreg	REGISTERS <regr>;	A pointer to data memory temporary stored in a data register.
padrinreg	REGISTERS <regr>;	A pointer to program memory temporary stored in a data register.
dcircinreg	REGISTERS <regr> (length : int);	A pointer to a circular array in data memory temporary stored in a data register.
pcircinreg	REGISTERS <regr> (length : int);	A pointer to a circular array in program memory temporary stored in a data register.
inreg	REGISTERS <regr>;	Any data address register stored in a data register. Used for storing pointers in memory.
mrreg	REGISTERS <regmr> (Type : mirTYPE);	The accumulate registers in the multiplier. The type is used to determine if the contents is integer or fixed point.
dadrreg	REGISTERS <regid>;	Any data address register.
padrreg	REGISTERS <regip>;	Any program address register.
dcircreg	REGISTERS <regid>;	Any data address register, containing a pointer to a circular array.
pcircreg	REGISTERS <regip>;	Any program address register, containing a pointer to a circular array.
direg	REGISTERS <regid>;	Any data address register, can be either circular- or ordinary pointer.
pireg	REGISTERS <regip>;	Any program address register, can be either circular- or ordinary pointer.
ireg	REGISTERS <regi>;	Any address register in DAG1 and DAG2.
dmodreg	REGISTERS <regmd>;	The modify registers in DAG1.
pmodreg	REGISTERS <regmp>;	The modify registers in DAG2.

Figure 6.1: The REGISTERS non-terminals.

Name	*Type*	*Description*
paddr	ADRMODE (reg : Register; offset : int);	Addressing a program memory address by an address register and a constant offset.
pmaddr	ADRMODE (reg : Register; mod : Register);	Addressing a program memory address by an address register and a modify register offset.
daddr	ADRMODE (reg : Register; offset : int);	Addressing a data memory address by an address register and a constant offset.
dmaddr	ADRMODE (reg : Register; mod : Register);	Addressing a data memory address by an address register and a modify register offset.
dmcirc	ADRMODE (reg : Register; mod : Register);	The same as dmaddr, except that the address points to a circular array.
pmcirc	ADRMODE (reg : Register; mod : Register);	The same as pmaddr, except that the address points to a circular array.
mult	ADRMODE (r1 : Register; r2 : Register; r1Type :mirTYPE; r2Type : mirTYPE; Type : mirTYPE);	Used when multiplying two fixed point registers.
multspec	ADRMODE (r1 : Register; r2 : Register; r1Type : mirTYPE; r2Type : mirTYPE; Type : mirTYPE);	Differs from mult in the constraints on the registers. Here the registers have harder constraints, so the instructions can be packed better.
arg	(length : int);	One parameter passed to a function.
arglist	(length : int);	All parameters to a function.
cc	UNIQUE (Relation : mirRELATION);	The condition code of a compare.
BegHomeLoc;		Special non-terminal for spilling registers to the stack.
nihil;		Used for the mirNil statement and for expressions that are not assigned to a variable, which produces no code.

Figure 6.2: The ADRMODE non-terminals and other non-terminals

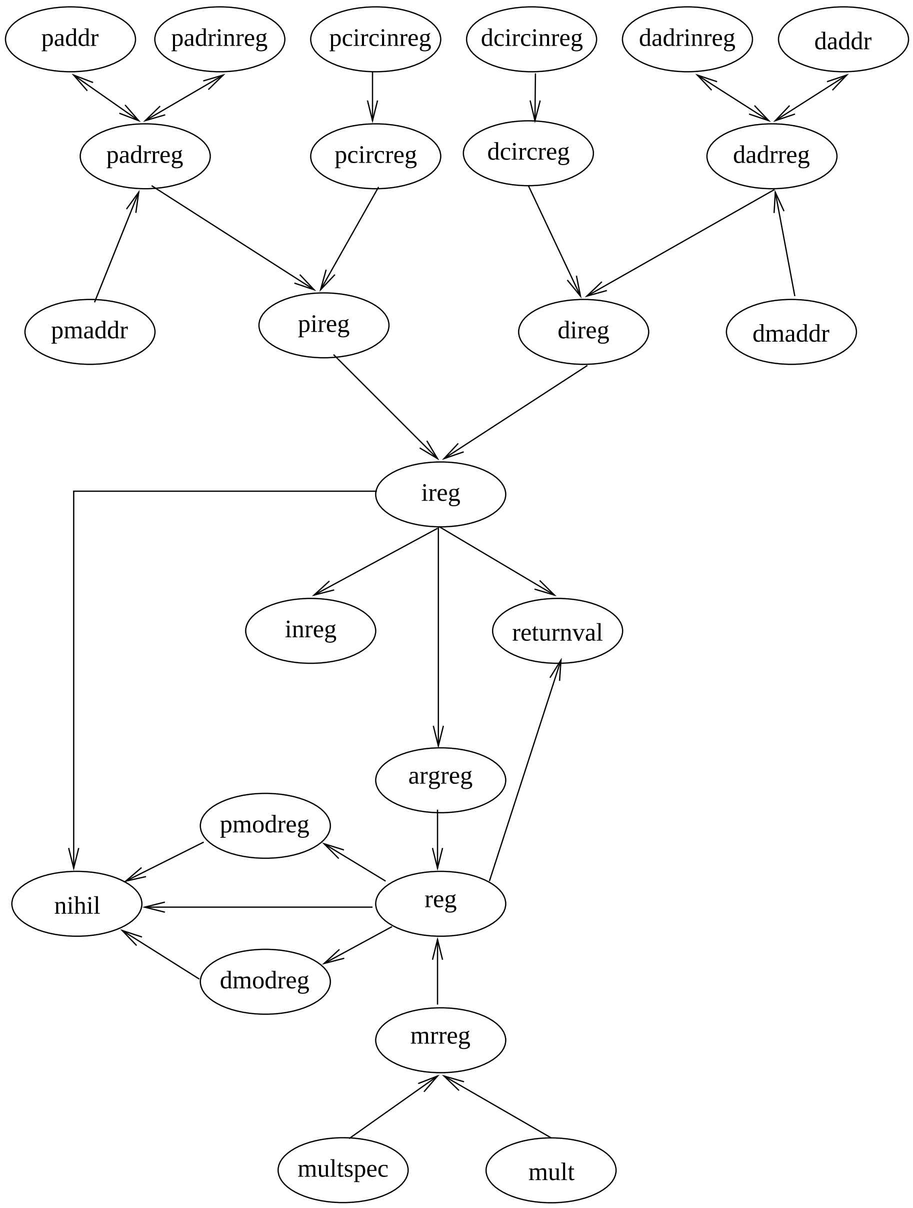


Figure 6.3: The graph describing the chain rules.

6.2 The Rules

The rules that had to be written are quite large in number, but they are rather easy to understand. There rules were divided into different categories, corresponding to different nodes in the CCMIR tree they operate on. The categories are, along with some rule examples, described in the sections below.

6.2.1 Simple Statements

The simple statement rules matches statements belonging to the domain **mirSimpleSTMT**, such as **mirAssign** and **mirFuncCall** [5].

A typical rule here is the **mirAssign** rule that takes the non-terminal `inreg`, which is an address pointer stored in a data register, and saves it in memory.

```
RULE [simpst_assigndmaddri] o:mirAssign ([l:dmaddr], r:inreg);
CONDITION {
        IS_POINTER(o.LhsType)
}
KILL l;
TEMPLATE mvstoredpre;
COST 3;
EMIT {
        emit(myst, MOVD_RP_R, l.mod, l.reg, r);
}
```

This rule takes two non-terminals, `dmaddr` and `inreg`, along with the **mirAssign**-statement, as a matching pattern. This means that the edges `Lhs` and `Rhs` under **mirAssign** must be reduced by other rules to the corresponding non-terminals in order for this rule to be applied. In addition to that the condition clause must be fulfilled for the rule to match. This clause says that the left hand side type of the assign must be a pointer, i.e. the value should be stored in memory. The cost of the rule is 3, which is the cost used for all single instructions. The template clause tells the instruction packer that this instruction uses the resources stated in the template `mvstoredpre`. If this rule is matched the emit clause tells the emit engine which assembler code to emit to the output file.

6.2.2 Control Statements

Control statements are statements in the CCMIR belonging to the domain **mirControlSTMT**, such as **mirGoto**, **mirIf** and **mirReturn** [5]. The three mentioned here are the most important one, and of them the most interesting is probably the **mirIf**. Below is a rule for this statement.

```
RULE [cntl_jcc] mirIf (cc, t:mirObjectAddr, e:mirObjectAddr);
COST 3;
BARRIER;
EMIT {
```

```
    mirBasicBlock   Then = (mirBasicBlock) t.Obj;
    mirBasicBlock   Else = (mirBasicBlock) e.Obj;
    mirRELATION     rel = cc.Relation;

    if (mirBasicBlock_equal(Then, engineStateGet->next_block)) {
      /*
       * Save jump by reversing targets and condition
       */
      Then    = (mirBasicBlock) e.Obj;
      Else    = (mirBasicBlock) t.Obj;
      rel     = reverse_relation(rel);
    }
    emit(myst, IF,          condition_code(rel));
    emit(myst, JUMP,        mirBasicBlock_get_Name(Then));

    if (! mirBasicBlock_equal(Else, engineStateGet->next_block)) {
      emit(myst, JUMP, mirBasicBlock_get_Name(Else));
    }
}
```

The rule takes the non-terminal `cc` and two **mirObjectAddr** under the statement **mirIf** as a pattern. The `cc` non-terminal is previously matched from a **mirCompare** statement, and contains the condition code for the jump. The rule also has the `BARRIER` clause, which means that the scheduler is prohibited to move code across the rule, i.e. no instructions after this instruction can be moved backwards, and no instructions before this instruction can be moved forwards. This is obvious since the `JUMP` instruction changes the sequential flow of the program an jumps to some other address. The emit part consists of two instructions, that are executed on the same cycle. First is the `IF` with some condition code, and then comes the actual jump. However, if the else part basic block isn't the next basic block to be emitted an additional jump instruction has to be made. The emit clause also shows that each emit clause has it's own scope, i.e. local variables may be used.

6.2.3 Simple Expressions

Simple expressions are expressions belonging to the domain **mirSimpleExpr**. These are quite many, and describes expressions that are constants, objects, address operations and bit-manipulations , such as **mirIntConst**, **mirObjectAddr** or **mirBitInsert** [5]. The rule described below matches the expression **mirContent**.

```
RULE [si_contentd] o:mirContent ([da:daddr]) -> r:reg;
CONDITION {
        IS_INTEGER(o.Type) || IS_REAL(o.Type) || IS_FIXED(o.Type)
}
TEMPLATE mvloaddpre;
```

```
COST 3;
EMIT {
        emit(myst, MOVD_R_DP,   r, da.offset, da.reg);
}
```

This rule matches a **mirContent**, containing the non-terminal `daddr`, and reduces this to a `reg` non-terminal. The `daddr` is an `ADRMODE` non-terminal, so this rule will take the address associated with `daddr`, and read that address from memory into a register, represented by the non-terminal `reg`. The condition is of course that the value can be stored in a register, thus the `CONDITION` clause states that the value should be of type integer, real or fixed. The resources are stated in the `TEMPLATE` clause. However, this rule will only be used on global variables, since if the associated object below a **mirContent** will be rewritten, in the lower engine, to the expression **BegUsePsr** stating that the object should be stored in a register. See the lower engine in section 6.4.

6.2.4 Unary Expressions

The unary expression rules are rules matched to expressions corresponding to expressions in the domain **mirOP1**. These are expressions having one operand, such as **mirAbs**, **mirNeg** and **mirConvert**. An example rule is presented for the **mirConvert** expression.

```
RULE [un_convintdadrreg] o:mirConvert(ir:reg) -> or:dadrreg;
COND {
        IS_INTEGER(o.FromType)
        && IS_POINTER(o.Type)
        && IS_DATASPACE(GET_PTR_SPACE(o.Type))
        && (! IS_CIRCULAR(o.Type))
}
TEMPLATE mvreg;
COST 3;
EMIT {
        emit(myst, MOV_R_R,     or, ir);
}
```

This rule is used when converting an integer to a pointer into the data memory. The rule matches the expression **mirConvert** taking a register non-terminal, an reduces this to a `dadrreg` non-terminal. The `CONDITION` clause is that the `FromType` should be integer, and the type converted to should be a non-circular pointer to data memory. The emit clause consists of a simple instruction moving the data register to the data address register.

6.2.5 Binary Expressions

The binary expressions rules are rules that matches the expressions in the domain **mirOP2**. These are expressions that have two operands, such as **mirAdd**, **mirDiff**, **mirDiv** and **mirMult**. Here is a rule for **mirDiff**:

```
RULE [bi_mrmultdiff] o:mirDiff(mr:mrreg, m:mult) -> omr:mrreg;
COND {
        IS_INTEGER_OR_FIXED(o.Type)
}
TEMPLATE multiply;
COST 2;
TARGET mr;
EMIT {
        char buf[16];
        get_mult_mod(buf, m.r1Type, m.r2Type, o.Type);
        emit(myst, MULSUB, mr, mr, m.r1, m.r2, buf);
        omr.Type = o.Type;
}
```

This rule is used when we have an expression looking something like this: $a = b - c * d$, where a ,b ,c and d are arbitrary expressions. The condition of this rule is that the type of the expression is integer or fixed point. This depends on the hardware constraint that the multiplier accumulator register only can store these types. The matching pattern includes the non-terminals `mrreg`, which is a multiplier accumulator register, and the `mult` non-terminal, which is an `ADRMODE` non-terminal, since it doesn't emit any code when reducing to it. The rule then reduces this to a `mrreg`. It has to be the same accumulator register, hence the `TARGET` clause must state that the `mrreg` non-terminal in the pattern is the target register. I.e. the `mr` instance and the `omr` instance is the same. The emit function in the `EMIT` clause must have the types of the multiplied registers, since it needs to set flags, depending on the type. For instance, the result of multiplying a signed integer with an unsigned integer is a signed integer. The cost of this rule is lower that three, since it actually does two computations in one, the multiplication and the subtraction.

6.2.6 Chain Rules

The chain rules are described in section 6.1.3, and will not be discussed here. The chain rules are simple, and the most of them doesn't contain any `EMIT` clause.

6.2.7 Rewrite Rules

The rewrite rules is an important part of the rules. Here is an example of the **BegUsePsr** rewrite rule:

```
REWR c:mirContent (o:mirObjectAddr) -> b:BegUsePsr;
COND {
        CONTENT_PSR(c) > 0 && (IS_INTEGER(c.Type) ||
        IS_REAL(c.Type) || IS_REAL(c.Type))
}
CALC {
```

```
            b.psr = CONTENT_PSR(c);
            b.Type = c.Type;
}
```

This rule rewrites the subtree **mirContent** containing a **mirObjectAddr** to the expression **BegUsePsr**. This is done under the condition that the `Psr` field in **mirContent** has a value greater than zero, ant that the type is storable in a data register.

The `CALC` clause contains the calculations that is to be performed when the rewrite occur. Here we set the `Psr` field and the type of the **BegUsePsr**.

6.2.8 Rules For Pseudo Registers

The rules for pseudo registers are partly already described above. However, there are some important rules left. These are the rules taking the **BegUsePsr**, **BegUsePtrPsr**, **BegAssignPsr** and **BegAssignPtrPsr** and reduces them to `REGISTER` non-terminals, such as `reg`, `dadrreg`, `padrreg` and so on. An example is given below.

```
RULE [psr_usedptrtempo] o:BegUsePtrPsr -> dadrreg TEMPO;
COND {
        IS_DATASPACE(GET_PTR_SPACE(o.Type))
        && (! IS_CIRCULAR(o.Type))
}
```

This rule reduces the **BegUsePtrPsr** to a `dadrreg`. The keyword `TEMPO` tells the global register allocator that this register should be used when assigning pseudo registers to registers.

A note to be made is that there are two different pseudo registers, one for ordinary data values, corresponding to the data registers, and one for pointers, corresponding to the address registers.

6.2.9 Spill Rules

Spill rules are needed when pseudo registers are used. These are necessary because when the register allocator occasionally runs out of physical registers to assign to pseudo registers, it will generate spill code to store contents of some physical registers so that they can be used for other pseudo registers. Later, when the values are needed, these are loaded back from the temporary storage. This storage is called spill location and the size of it is determined by the register allocator.

It is up to the implementation to find storage and do the spilling. In this thesis the stack is used as spill location. In the prologue of a function the needed number of spill locations are checked and the stack pointer is modified to allocate place. There is a special non-terminal in BEG called *BegHomeLoc*, and for all registers that need to be spilled, chain rules must be written to reduce from or to this non-terminal:

```
reg -> BegHomeLoc;
EMIT {
...
}

BegHomeLoc -> reg;
EMIT {
...
}
```

The *BegHomeLoc* non-terminal is translated to an index into the spill location inside the emit code, which can be used to determine where to store and reload the contents.

Because some registers in the SHARC depend on each other and are used together, each spill index has place for three registers. For example, when storing address registers, the base and length registers for the address register are also stored, if this address register is used in circular addressing.

6.2.10 Xir Rules

Xir stands, for eXit Intermediate Representation, and is the result given from the lower engine, see section 6.4.

An example of a xir rule is:

```
RULE [xir_xirpassr4] a:xirPass (argreg<r4>) -> arg;
CONDITION {
        a.OnStack == FALSE
        && a.Index == 0
}
EMIT {
        arg.length = 0;
}
```

This rule is used when calling a function. The **xirPass** expression is inserted as a part of the **xirFuncCall**.

6.3 The Datagen Engine

The datagen engine is responsible for generating global data. It also puts initializations of local arrays as global data. For instance if a local array is initiated with some elements, these elements must be stored somewhere, so that every time the function is called, these elements are placed in the local array, that resides on the stack.

The SHARC processor has an architecture file associated with it. This file contain information about the different segments of memory. For instance, global data memory segments, and so on. To specify in which segment the data is stored, one uses the assembler directive:

```
.SEGMENT/type <segname>;

  /* Assembler code or data here */

.ENDSEG;
```

The type is either `PM` or `DM`, for program memory or data memory, respectively. The `segname` should correspond to a name of a segment given in the architecture file.

The datagen engine goes through all **mirObject** that are of type **mirGlobal**, and checks if they belong to the data memory or the program memory. All global objects belonging to the program memory are stored in one segment and all global objects belonging to the data memory are stored in another. The reason for only having two segments for the global data is that all architecture files are obliged to have these two segments.

The datagen engine also goes through all local variables to find the ones of type array or struct, that need to be initialized. These initial values are then stored in a separate segment. We have chosen to put this segment in data memory, but that can easily be changed by an option to the compiler. The ideal case should probably be to have half of them in data memory and the other half in program memory, because then the instruction packer could probably do a better optimization of the code.

Another task for the datagen engine is to give a label to each **mirBasicBlock**. This is done by giving them the prefix string "_L" followed by an number. This label is needed when jumping between basic blocks.

All this has been implemented in the datagen, however not all code has been written from scratch. In this thesis, an existing datagen engine was taken and rewritten. This method was also used for the lower engine, see section 6.4.

6.4 The Lower Engine

The lower engine is responsible for rewriting the CCMIR so that BEG can understand it. The result from this engine is called xir, which stands for eXit Intermediate Representation. Since BEG doesn't understand constructs such as the `LIST` functor, these have to be flattened. For instance, the **mirFuncCall** statement shown below has a field `Params`, which is a `LIST` functor, i.e. a linked list of **mirEXPR**, representing the parameters to the function.

```
mirFuncCall <
        Proc [primary]:         mirProcAddr,
        Params [primary]:       LIST[primary](mirEXPR),
        Res [primary]:          mirADDR
        >
```

This is rewritten to the structure in figure 6.4.

The tree in figure 6.4 is a part of the lowered **mirFuncCall**, called **xirFuncCall**. This statement can then be handled in BEG. Note that the arguments

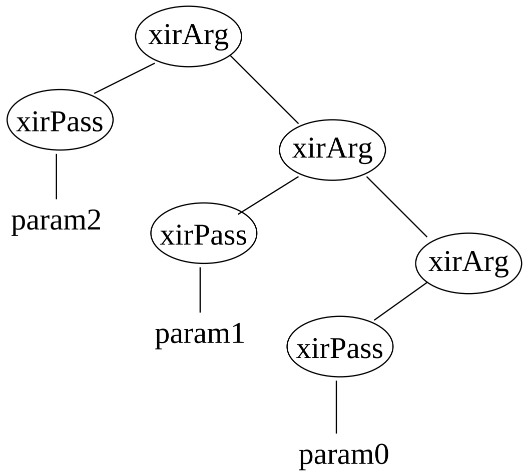


Figure 6.4: The lowered structure of parameters to functions.

are in backwards order if a depth first search of the tree is performed, however the matching will be performed using depth first search, and the arguments will then be in correct order.

The lower engine is also responsible for assigning a pseudo register to each variable. This is performed by extending **mirContent** with a field called **PSR**, with type `INT`. Each variable object, below the **mirContent** node will then have an unique pseudo register. Then, it is up to the register allocator to allocate registers for these pseudo registers. Since the SHARC has address registers, **PSR**'s are also assigned to variables that are pointers. These are then handled with separate rules.

The lower also sets some type information in certain statements. In order to avoid having attributes in the non-terminals, the type is instead stored in the statement. For instance, **mirAssign** gets two new fields, **LhsType** and **RhsType**, corresponding to the type information in the children nodes of the **mirAssign** statement.

6.5 Parallel Instructions

As discussed in section 5.1.2 on page 35, a scheduler description has to be written to direct the scheduler to pack instructions together where possible. The list of resources defined in section 5.1.2 on page 35 will be used:

```
RESOURCES
        Compute (Alu, Multiplier, Shifter),
        Move    (mv1, mv2),
        pm,
        dm,
```

```
            noshiftimm;
```

To simplify future changes of what resources different instructions might use, some level of abstraction is used. Because of this, different templates are used in some cases even if the resources used by the instructions are actually the same. Also instructions where usage of one of the resources block the others, the blocked resources are also said to be used by the instruction.

The following set of templates will be defined:

```
TEMPLATES
  DEFTMPL       (Alu Multiplier Shifter mv1 mv2 pm dm);
  mvreg         (mv1 mv2 noshiftimm);
  mvloaddpre    (mv1 mv2 dm noshiftimm);
  mvloadppre    (mv1 mv2 pm noshiftimm);
  mvloaddpost   (mv1 dm);
  mvloadppost   (mv2 pm);
  move          (mv1 mv2 pm dm noshiftimm);
  modify        (mv1 mv2 pm dm noshiftimm);
  modifyimm     (Alu Multiplier Shifter mv1 mv2 pm dm noshiftimm);
  mvstoredpre   (mv1 mv2 dm noshiftimm);
  mvstoreppre   (mv1 mv2 pm noshiftimm);
  mvstoredpost  (mv1 dm);
  mvstoreppost  (mv2 pm);
  puts          (mv1 dm);
  alu1          (Alu Multiplier Shifter);
  alu2          (Alu Multiplier Shifter);
  alu2cmp       (Alu Multiplier Shifter);
  aluspec       (Alu);
  multiply      (Alu Multiplier Shifter);
  mulspec       (Multiplier Shifter);
  shift1        (Alu Multiplier Shifter noshiftimm);
  shift2        (Alu Multiplier Shifter);
  shift3        (Alu Multiplier Shifter);
  ldimm         (Alu Multiplier Shifter mv1 mv2);
  ldadrimm      (Alu Multiplier Shifter mv1 mv2);
  ldmrimm       (Alu Multiplier Shifter);
```

Description of the templates are shown in figure 6.5. All templates are defined to describe a single cycle. The SHARC actually uses three cycles for each instruction in fetch, decode and execute steps, but because of the execution pipeline the throughput is one instruction per cycle. Hence the scheduler description is simply written without the fetch and decode steps, because these are automatically performed in parallel.

The `aluspec` and `mulspec` templates are special (see 5.1.2 on page 35). These templates are used to pack ALU and multiply operations. Because of the register restrictions this can be done in two ways, with separate rules, hence there will be two versions of the rules for multiply, add, subtract, average and conversion operations. The rules for **mirPlus** will be shown here as an example.

Template	*Description*
mvreg	Move between two registers
mvloaddpre	Move from data memory to register with premodify
mvloadppre	Move from program memory to register with premodify
mvloaddpost	Move from data memory to register with postmodify
mvloadppost	Move from program memory to register with postmodify
move	General move
modify	General modify of a Ix-register with an Mx-register
modifyimm	Modify of a Ix-register with an immediate value
mvstoredpre	Move from register to data memory with premodify
mvstoreppre	Move from register to program memory with premodify
mvstoredpost	Move from register to data memory with postmodify
mvstoreppost	Move from register to program memory with postmodify
puts	Push a register to the stack (assembler macro)
alu1	Unary ALU operation
alu2	Binary ALU operation
alu2cmp	Compare operation
aluspec	ALU operation with special register restrictions
multiply	General multiply operation
mulspec	Multiply operation with special register restrictions
shift1	Shifting a register with an immediate amount
shift2	Shifting a register with an amount stored in a register
shift3	Shifting and logical-or of a register with an amount stored in a register
ldimm	Load an immediate value to a register
ldadrimm	Load an immediate value to an address register
ldmrimm	Load an immediate value to a Multiplier register

Figure 6.5: Description of used scheduler templates

The first version of the rule only determines the physical registers used in the instructions and sets correct template. The rule for **mirPlus** looks like this:

```
RULE [bi_plus] o:mirPlus(r1:reg, r2:reg) -> r:reg;
COND {
        IS_INTEGER_OR_FIXED(o.Type)
}
TEMPLATE {
        (IS_REGSET23(r1,r2) ? tmplaluspec : tmplalu2)
}
COST 3;
EMIT {
        emit(myst, ADD, r, r1, r2);
}
```

where the `TEMPLATE` clause consists of a expression returning a template object. If the registers `r1` and `r2` are in the registers sets 2 and 3[1] respectively, then template will be `aluspec`, which only uses the `ALU` resource, otherwise the template will be `alu2`, which uses `ALU`, `Multiplier` and `Shifter` resources and a multiply operation will not be packed together with this add operation.

This rule is just a normal rule and the packing is depending on the randomly selected registers. If packing is critical, another rule should be used with register restrictions to make sure that the registers are selected to allow packing. To allow the user to make this decision, a command-line option will be read and checked as a condition for the rule to match. This version of the rule looks like this:

```
RULE [bi_plusspec] o:mirPlus(   r1:reg<r8..r11> ,
                                r2:reg<r12..r15> ) -> r:reg;
COND {
        IS_ON_OPTMF &&
        IS_INTEGER_OR_FIXED(o.Type)
}
TEMPLATE aluspec;
COST 2;
EMIT {
        emit(myst, ADD, r, r1, r2);
}
```

Here the first operand of **mirPlus** is forced to be one of the physical `r8`, `r9`, `r10` or `r11` and the second operand is forced to be one of `r12`, `r13`, `r14` and `r15`. The macro `IS_ON_OPTMF` in the `CONDITION` clause checks whether the `optMultiFunction` variable of the config struct of the engines state struct is true:

```
#define IS_ON_OPTMF     (st->config->optMultiFunction != FALSE)
```

[1]Register set 0 is <r0,r1,r2,r3>, set 1 is <r4,r5,r6,r7>, set 2 is <r8,r9,r10,r11> and set 3 is <r12,r13,r14,r15>

This option is set in the init function of the emit engine, depending on the command-line option given.

This version of the rule will always have correct registers to allow packing, and it will be selected more often than the other version because it has a lower cost, as long as the multifunction option is given. However using, this rule might limit the registers available for the register allocator, because it forces the register allocator to use these specific register sets for these operands.

6.6 The Emit Engine

In this section operation of the emit engine will be described.

6.6.1 Initialization

The emit engine starts by calculating frame offsets and store these in the state struct. In the `calculate_frame` function the variables `local_size` and `spill_size` are set, and then `local_base` and `spill_base` are calculated based on those. `local_base` is the base offset of the local variables and `spill_base` is the base offset of the spill location. These offsets are used when calculating actual addresses of objects. Other state variables such as the output filename are also initialized.

6.6.2 Emitting

An important issue while emitting is the presence of parallel instructions. Since each rule usually emits a single instruction, it is not possible to do packing of instructions in the rules. This must be handled separately. The solution used in this thesis is to have an `emit` function which can be called from all the rules, which takes care of packing internally.

The emit function

The emit function takes two or more arguments. First argument is the state struct of the emit engine, and the second argument is the id of the instruction which will be emitted. The file `sharcemit.h` contains an enumeration for instruction id:s, which looks like:

```
enum OPCODES {
 ...
 MOVD_DO_R,
 MOVP_DO_R,
 ...
 ADD,
 SUB,
 AVG,
 COMP,
```

```
  ...
}
```

The file `sharcemit.c` contains the emit code and the list of opcode structs. Every opcode object contains an instruction id, the string to emit, number of parameters, types of the parameters and the instruction class it belongs. The parameter types are:

- `reg` : Register.
- `feg` : Register with a floating point prefix in the name.
- `val` : Integer value.
- `var` : Variable.
- `lbl` : Label.
- `mod`, `mod1`, `mod2` : Different modifiers.
- `cond` : Condition code.
- `str` : String (plain text).

Different type of parameters are emitted differently and the parameter types are also needed for reading the correct type of optional argument when the emit function is called. This way only one function can be used everywhere even if different number and type of arguments needs to be passed to the emit function.

The different instruction classes are needed because instructions must be in a certain order when they are packed together. For example, compute instructions must be written first, followed by a move instruction. The different instruction classes are:

- `moveop` : Move instruction.
- `modop` : Modify instruction.
- `aluop` : ALU instruction.
- `mulop` : Multiply instruction.
- `shftop` : Shift instruction.
- `flowop` : Instruction that alters program flow (i.e. JUMP).
- `miscop` : Other instructions and emitted things (i.e. comments, plain text etc).

The emit function has an internal FIFO to store the emitted instructions before it actually writes them to the output file. This is needed because the cycle number of the instructions must be checked to see which instructions should be packed together, before they actually are emitted. Instructions must also be sorted because they need to be in correct order when packed[2].

Besides the cycle number, the rule number in which the instruction was emitted is checked. Instructions emitted from the same rule has the same cycle number, but this does not mean that these should be packed together. The scheduler only considers one rule at a time when assigning cycle numbers to the rules. When emitting several instructions in a single rule, the `emitw` function can be used to force packing instructions. The `emitw` function marks the instruction which is its argument to be packed with the instruction emitted next. For example:

```
emitw(state, MOVD_DP_R, ...)
emit(state, ADD, ...)
```

will emit the code:

```
Rx = Ry + Rz,  DM(M0,I0) = Rz;
```

regardless of what cycle number and rule number the current state contains.

As an example, consider the following rules:

```
RULE [simpst_assigndmaddr] o:mirAssign ([l:dmaddr], r:reg);
TEMPLATE mvstoredpre;
EMIT {
        emit(myst, MOVD_RP_R, l.mod, l.reg, r);
}

RULE [bi_plus] o:mirPlus(r1:reg, r2:reg) -> r:reg;
TEMPLATE alu2;
EMIT {
        emit(myst, ADD, r, r1, r2);
}
```

If these are matched and are to be emitted next to each other, the scheduler will assign same cycle number to them, because the templates mvstoredpre and alu2 use different sets of processor resources. The emit function stores these in the FIFO, and when the FIFO has enough number of instructions, it starts to check them and emit them. These two instruction will first be sorted in the function `sortops`. The sort order is:

```
(aluop, mulop or shftop) < (moveop or modop) < (other classes)
```

This means that compute instructions will be emitted first, then the move or modify instruction and finally all other instructions (comments etc).

Following lines are part of the oplist vector containing op structs:

[2]This a limitation in the assembler which is used to assemble the generated code. A better assembler could allow custom order of instructions.

```
OpCode op_list[] = {
  ...
  { MOVD_RP_R, "DM(%s,%s) = %s", 3,{ reg, reg, reg }, moveop },
  { ADD,       "%s = %s + %s",   3,{ reg, reg, reg }, aluop },
  ...
}
```

According to these, the `emit` function will generate the following line of code in the output file:

```
Ra = Rb + Rc,   DM(Md,Ie) = Rf;
```

The emit functions also does special checking for labels, the `IF` function etc to emit correct output.

6.7 Loop Optimization

The loop instruction in SHARC has been implemented for loops with a known iteration count. The implementation has been done as suggested in section 5.2.2 on page 39. An entire new engine was written for this purpose, called sharcloop. The sharcloop engine uses the result from the loopanalysis engine, i.e. the **mirLoopMarker** added to each **mirProcedure**.The sharcloop engine can take two options, `loopopt` and `pm`. The `loopopt` option tells the engine to perform loop optimization regarding the `DO-UNTIL` instruction. The `pm` option tells sharcloop to perform post modify optimization where possible, according to the discussion in section 5.2.3 on page 40.

Chapter 7

Conclusion

This chapter discusses the results from the compiler and the conclusions that can be drawn from these results. A comparison with the existing compiler from Analog Devices, the `g21k` compiler, is also done. The compiler is also tested with different test suites, along with some small examples written for this purpose. Comments on the CoSy system, regarding limitations and bugs, are discussed. A summary is also given last in this chapter.

7.1 Testing

The SHARC processor is often used in embedded systems and is therefore not so easy to test using standardized test suites. The compiler was tested by compiling the code on a Sun SPARCstation to SHARC assembler form. Then this code was copied to a PC with the Windows NT operating system, with the assembler `asm21k` installed. An object file was produced, linked with some standard libraries, and then sent to the SHARC processor for execution. The SHARC processor sits on a card, plugged into the PC. There is a test suite distributed with the compiler `lcc` [11]. This test suite is about 5 kLOC, and it only uses the `printf` function. This makes it easy to compare results, by using the unix command diff.

More testing of the compiler would be advised. Larger test suites are available from ACE. Due to lack of time, these tests has not been run.

The testing part is divided into two separate parts. First, testing can be done to ensure that the compiler can compile the whole language, i.e. the whole C language is understandable by the compiler. Second, the assembler code produced by the compiler can be tested by running the program on the SHARC processor and check that the right results is given. These different test is described in the sections below.

7.1.1 Testing C-language Coverage

To test the C-language Coverage, the test-suite from lcc [11] has been used, as discussed above. This test suite consist of one large file that tests the entire language, about 5 kloc, and some additional files for programs doing specific tasks. For instance, a small function that solves the eight queen problem is available.

All these files have passed through the compiler without any error messages. There are of course some warning messages, since the compiler doesn't know about, for instance the printf function.

The compiler can therefore be said to support the entire C-language. This statement is perhaps a bit vague, since the testing performed isn't large. To be absolutely sure the compiler must be passed through more an larger tests. However this is unrealistic to perform within the scope of a master thesis of 20 weeks.

7.1.2 Testing Executables

The more interesting tests are actually running the program on the SHARC processor. The programs tested in previous section have also been run on the SHARC, with success.

7.2 Comparison With g21k

When the compiler was compared with the `g21k` compiler from Analog Devices all optimization switches was given to the compilers. This was done by the `-O3` option to `g21k` and the `-pm` option to our compiler. The files distributed with the lcc compiler was tested and the result was compared. The g21k compiler was also runned without optimization, to get a better comparison between the compilers.

In order to test how good the instruction packing was compared to the g21k compiler, the number of instructions packed was divided with the total number of instructions. Another comparison can be to simply compare the total number of instructions that the compiler produces for a certain code. A third way to do comparisons is to actually run the code on the SHARC processor and measure the time each program takes to execute.

Figure 7.1 shows the static comparisons discussed above, i.e. the number of instructions and the percentage of packing, for some C programs. A note here is that the bit manipulation program gives a poor result. The reason for this is a bad implementation of inserting a bit field into a register. This can be improved substantially. Due to lack of time this has not been be done in this thesis.

Figure 7.2 shows the run time for some programs.

C-file	*# instructions*			*percent packing*			*Description*
	a	*b*	*c*	*a*	*b*	*c*	
8q.c	238	164	223	0	3	3	Solves the eight queen problem. Contains much loops.
field.c	231	183	417	0	5	0.2	Some functions for testing bit manipulations.
mmov.c	165	132	176	0	7	5	Copies matrises.
matrix.c	126	83	105	0	7	2	Also copies matrises.

Figure 7.1: Comparisons of the g21k compiler and this SHARC compiler.
a is the g21k without optimization.
b is the g21k with optimization.
c is our compiler with optimization.

C-file	*time μ s*			*Description*
	a	*b*	*c*	
8q.c	53851	51593	53166	Solves the eight queen problem. Contains much loops. Loop optimization in our compiler didn't work.
matrix.c	6821	1158	3544	Copies matrises. Loop optimization in our compiler didn't work.
vss.c	1924	471	400	DSP specific function, operates on a vector.

Figure 7.2: Run time comparison of the g21k compiler and this one.
a is the g21k without optimization.
b is the g21k with optimization.
c is our compiler with optimization.

7.2.1 Comments

Because the loop analysis engine doesn't detect all iteration variables in all cases, the result is a bit missleading. The loopanalysis engine is only a beta version, and we think the code will be more efficient with a newer version of this engine. This has also, of course, a great impact on the runtime of the programs, as shown in figure 7.2. Another improvement that can be done is to develop the post modify option even more. Here, a lot of work can be done. This thesis has only implemented a very small subset of this.

Another area of improving the back end is to implement moving of registers through the ALU. This can be done by using the `PASS` instruction. Then, in theory, two move operations between registers can be performed in one cycle, for instance:

```
R1=R2, R5 = PASS r6;
```

However, the tricky part is to write rules that perform this in an optimal way. One solution might be to let 50% of the move operations use the ALU and the other 50% do it in the ordinary way. Then perhaps better performance will be achieved, if the packer packs many of the move operations together.

7.3 Conclusions on the CoSy system

The CoSy system is very large and complex. Therefore some bugs are always present. One of the most irritating bug was that BEG crashed, without giving any error message, if a curly bracket was left out from for instance an emit clause. If the CGD file is large, the error can be hard to detect, since no error message is given.

BEG also has some limitations. One substantial limitation is that several statements cannot be matched to one instruction, using a match pattern. This limitation comes from the match and lirg engines, which handle each statement as a unit, and matches rules for each of the statements separately. We would like to match two statements to one instruction, se 5.2.3 on page 40. However a workaround can be performed by creating a new statement for this instruction. The disadvantage is of course that an engine has to be written, that finds and removes the two statements and adds the new one.

Another limitation on BEG is that the register allocator only runs once, after instruction matching. This is unfortunate when some rules exist in several versions, where one of the rule has register constraints on the register non-terminals in the pattern, see section 6.5 on page 55. The matcher will then choose the rule with lowest cost, independent of whether the register constraints are fulfilled or not. It is very common in the SHARC processor, and in all DSPs, that some operations have constraints on the registers when the operation is packed with another operation and run in parallel. Therefore it would be better if this limitation could be removed. A way to remove this limitation can perhaps be to also run the register allocator before matching also.

7.4 Summary

The compiler produced in this master thesis is in some aspects as good, or even better, than the g21k compiler from Analog Devices. In some cases the g21k compiler is better that this one, though. However, our compiler is not tested as much as is required. In order to get a reliable compiler it has to go through substantially larger tests.

The compiler does however support the whole C language, and that is not so bad for a compiler developed in 10 man months. When ACE releases a new version of CoSy perhaps other loop optimizing engines, along with a better loop anlyser, will make this compiler even better. The bugs found in CoSy, for instance the instruction packing in the match engine, will also probably be corrected.

Appendix A

Common Terms And Abbreviations

- ACE - Associated Computer Experts, Amsterdam, Netherlands
- ALU - Arithmetic and Logical Unit
- BEG - Back End Generator
- CCMIR - Common Cosy Medium level Intermediate Representation
- CoSy - Compilation System
- DMA - Direct Memory Access
  A external device can access memory directly without using the main processor.
- DMCP - Data Manipulation and Control Package
- DSP - Digital Signal Processor
- EDL - Engine Definition Language
- fSDL - full Structure Definition Language
- Latency - The number of cycles two instruction must be separated with, in order to execute the instructions correctly on the processor.
- SHARC - Super Harvard Architecture Computer

Bibliography

[1] *http://www.ida.liu.se/ext/witas/.*

[2] Inc. Analog Devices. *ADSP 2106x SHARC User's Manual.* First edition, 1995.

[3] Niclas Andersson and Peter Fritzson. Overview and industrial application of code generator generators. *Journal of Systems and Software*, 1995.

[4] Hans von Someren Martin Alt, Uwe Assmann. Cosy compiler phase embedding with the cosy compiler model. In Peter A. Fritzson, editor, *Compiler Construction*, 1994.

[5] ACE Associated Computer Experts bv. *CCMIR Definition, specification in fSDL, Description and Rationale.* Number 9.20. April 1998.

[6] ACE Associated Computer Experts bv. *fSDL Definition and Generated Interfaces.* Number 9.4. May 1997.

[7] ACE Associated Computer Experts bv. *Beg - Cosy Reference Manual.* Number 9.16. April 1998.

[8] R. Landwehr H. Emmelmann, F. W. Schrrer. Beg - a generator for efficient back ends. *ACM Sigplan Notices*, 24(7):227–237.

[9] ACE Associated Computer Experts bv. *DSP-C An extension to ISO/IRC IS 9899:1990.* Number 9.5. May 1998.

[10] Jeffrey D. Ullman Alfred V. Aho, Ravi Sethi. *Compilers Principles, Techniques, and tools.* ADDISON-WESLEY, 1986.

[11] Chris Fraser <cwf@research.att.com>Dave Hanson <drh@cs.princeton.edu>. *ftp://ftp.cs.princeton.edu/pub/lcc/.*