Code generation for RISC machines

Yedema, W.F.D.

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

MASTER'S THESIS

Code generation for RISC machines

by
W.F.D. Yedema

Supervisors: dr. ir. C. Hemerik and dr. ir. A. Augusteijn

Eindhoven, December 2002

Abstract

RISC machines are all very much alike. So much alike, in fact, that when one has made a compiler for one RISC machine, it should be the case that the effort to retarget that compiler to another RISC machine is limited. However, this is often not the case, for the simple reason that other targets are not taken into account from the start.

This report presents a retargetable back-end of a compiler. The back-end uses two machine description files, which are used mainly to configure the instruction selection and assembly file emission phases, the two most machine dependent stages.

The back-end can handle 2-address instructions, instructions that perform `a:=a+b` instead of `a:=b+c`, by removing the register conflicts they introduce prior to scheduling. A scheduling heuristic is presented for non-interlocking processors, aimed at reducing register pressure. A formalization of a known local register allocation algorithm is presented.

Contents

Chapter 1

Introduction

Programming a RISC[1] machine in assembly is easy. That is, in fact, what a RISC machine is all about. The instructions are executed in the same order as you read them in the assembly file, and the machine executes at most one instruction at a time. This may all seem very obvious to the uninitiated, but there are plenty of examples where none of this holds, eg. VLIW processors.

As opposed to CISC machines, a RISC machine has specific instructions to access memory, eg. an `add` works only on registers, not on memory. Usually, a RISC machine has more registers than CISC machines, and aside from the program counter no special purpose registers.

A consequence of these characteristics is that, from the programmers point of view, most RISC machines are very much alike. In a compiler these differences manifest themselves mostly in the back-end. To avoid creating a new back-end for every new RISC machine, we wish to have a formalism that describes its characteristics. This formalism should then be used to parameterize the compiler to behave according to these characteristics. In this way one has the characteristics of each machine nicely organized in one file instead of scattered around in the many source files of the compiler.

The formalism should be powerful enough to describe an arbitrary RISC machine in the level of detail required by the compiler. This means that we have no need for a formalism that can also be used to *design* a RISC machine, like VHDL. The formalism should describe the machine on a much more abstract level. A compiler for a RISC machine usually requires information about the Instruction Set Architecture (ISA), and registers, but little more.

One powerful feature not often seen in RISC machines is guarded instructions, instructions that are executed only when a certain condition is met. This feature alone is responsible for a number of changes in almost all stages of the back-end with respect to a back-end without support for guarded instructions.

This report describes a compiler back-end that is parameterizable with a machine description formalism, and has support for features such as guarded instructions. The back-end has been tested for speed and quality with the ARM7 instruction set.

The ARM7 compiler back-end was made as part of the ThumbScrews project. The goal of the ThumbScrews project was to create a processor and compiler tool chain that would result in a 40% reduction in code size with respect to the ARM7-Thumb ISA, one of the processors best suited for application that require small code size. This reduction was achieved by using application specific instruction sets. By designing an instruction set for a single application one can make use of frequently occurring sequences of code, and make a single instruction out of them. In the hardware the instructions are decoded into (sequences of) ARM instructions, using a so called pre-processor. The drawback of the compacted code is that it is slower in execution, the pre-processor is a stack machine which usually requires more instructions than register based machines for the same function. Fortunately, since it is only pre-processor, one can switch off the pre-processor and run normal ARM7 or ARM7-Thumb code. Another advantage of the pre-processor approach is that it can relatively easily be retargeted towards another processor.

[1] see Appendix B for abbreviations and definitions.

Due to the requirement to switch off the pre-processor, the compact-code compiler must be able to work together with the compiler for the processor itself. We could use the compiler provided with the host processor for that, which is what we have done. But if you have your own compiler for the processor as well, you can perform much better whole program analysis, improving code density. Furthermore development itself is made easier by providing a single compiler instead of two.

The ThumbScrews project team consisted of members of the Compiler Technology cluster and Processor Oriented Architectures cluster, in the Information Technology and IPA groups of the Information and Software Technology sector of Philips Research in Eindhoven.

This report is organized as follows. In this chapter I introduce the problem, and give some background information. In chapter 2 I explain the order of the stages of the code generator. Chapter 8 introduces the machine description formalism. In chapter 3 I explain some of the local optimizations done for the ARM processor. In chapter 4 I explain the first part of the local register allocation phase, which involves solving register conflicts on DAGs. In chapter 5 the scheduling algorithm is explained. Chapter 6 explains the second part of the local register allocator. Chapter 7 explains if-conversion, and finally some benchmarks in chapter 9, and conclusions and future work in chapter 10. Chapter B lists some frequently used definitions and abbreviations.

1.1 An overview of a compiler

A compiler is a program that translates a string written in one language to a string written in another language. More commonly, it reads a file written in the input language, and writes a file written in the output language. The input file is traditionally named the *source* file. The compiler we will consider here translates a C source file to an assembly file. This translation is performed in a number of steps. The following is a list of phases of the compiler, although it is incomplete it helps to illustrate what goes on inside the compiler.

Parsing and scanning The source file is read and stored as an abstract syntax tree that models the syntactical structure of the source.

Intermediate code generator The abstract syntax tree is transformed into a *Control Flow Graph* of *Basic Blocks*. A basic block is a control flow element with no internal control flow. It contains a data flow graph of machine independent instructions.

Global and local optimizations Local optimizations are optimizations within a basic block, global optimizations are optimizations that span basic blocks.

Decision tree construction A decision tree, or dtree for short, is a tree of basic blocks.

Global register allocation Global register allocation, or GRA, allocates registers to the data flow edges that span decision trees.

Instruction selection During this phase the machine independent instructions are translated to machine dependent instructions.

Instruction scheduling The data flow graph of instructions is transformed into a sequence of instructions.

Local register allocation The local register allocator allocates registers to the dtree internal data flow edges.

Dtree flattening The instructions in the dtree are transformed into guarded instructions. After this phase the dtree will contain one basic block instead of a tree of basic blocks.

Code emission Finally the assembly file is written.

The last five phases are part of what is usually called the back-end of the compiler. The code generator I have made peforms all the stages of the back-end. The instruction selector is generated using a tool developed within the compiler cluster, called doggy. The code emitter is also generated, using a tool made specifically for this back-end. I was involved in the design of both languages.

1.2 The problem

There are a lot of RISC machines, each one different from the next. Each of these RISC machines has at least one compiler. The RISC machines we are currently interested in are the ARM processor family and the MIPS processor family. To be able to quickly create a compiler for these processors we require a parameterized compiler.

To find out what should be parameterized and what not, we have to take a quick look at what the code generator does. The output of the code generator is an assembly file. Obviously the assembly file must be accepted by the assembler of the machine it was made for. This means that the emission phase, the phase during which the output file is written, must be parameterized with the syntax and semantics of the target assembly language.

A typical assembly file roughly contains two things, data and instructions. The data contains some variables and constants defined in the source program, and the instructions specify what the processor should do. For normal RISC processors the instructions perform simple operations that work on registers, such as add A and B, and put the result in C. The number of registers in a processor is usually quite limited, eg. 16 for the ARM or 32 for the MIPS.

Obviously the source language is independent of this limit on registers, therefore somewhere during compilation the compiler must assign, or allocate, registers to variables and intermediate values. This is called register allocation, and is usually split into local and global register allocation (LRA and GRA). As may become clear, one task of the code generator is register allocation. To do this successfully it needs to know the number of registers of the machine and their properties.

Due to their implementation, some instructions may have some requirements with respect to their arguments. In the ARM, for example, the first argument of the multiply must be a different register than its result. Another frequently occurring requirement is that the result register must be the *same* as the first argument. More rare requirement is a requirement for a specific register, for example a `PUSH` instruction that only works with the stack pointer. The register allocation must be able to cope with such requirements.

Throughout most of the compilation process, a compiler uses an internal instruction set. Although the internal instruction set is usually similar to the target machine instruction set, they are not equal, for reasons of retargetability. During code generation the internal instructions must be translated into actual machine instructions. This process is called instruction selection, and obviously, must be parameterized.

In summary, the code generator must be parameterized with the following:

- Number of registers, and register requirements
- Target assembly syntax and semantics
- Mapping from input instructions to output instructions

These parameters will be specified in a small amount of files preferably one. These files will be written in a language especially designed for the purpose of describing a machine. The language must have an easy learning curve since the files written in the language will not be edited frequently.

Furthermore, the aim is to deliver an industrial strength code generator, parameterized for an ARM7 RISC processor. The quality and speed of the generated code must be comparable to that of the compiler supplied by ARM.

1.3 Graphs

A compiler uses graphs to represent a program. How this is done is explained in section 1.4. This section models the directed graph and its properties, and introduces a notation for reasoning about graphs.

A directed graph, or *digraph*, G, is a pair (V, E) such that $E \subseteq V \times V$, where V is the set of *vertices*, and E is the set of *edges*. These definitions are captured in the following functions:

$$\begin{aligned} \mathit{vertices}.(V, E) &= V \\ \mathit{edges}.(V, E) &= E \end{aligned}$$

We define the following relations between vertices.

$$a \rightarrow_E b \equiv (a, b) \in E$$

We will write $a \rightarrow b$ instead of $a \rightarrow_E b$ when the edge set can be deduced from a and b. We will write $\rightarrow^+$ for the transitive closure, and $\rightarrow^*$ for the transitive reflexive closure of $\rightarrow$.

The edge $a \rightarrow b$, is an *in-edge* of b and an *out-edge* of a. Furthermore a is said to be a *parent* of b, and b is said to be a *child* of a, a is the *source* of the edge, and b the *destination*. For $u \rightarrow^+ v$, u is said to be an *ancestor* of v, and v is said to be a *descendant* of u.

$$\begin{aligned} parents.u &= \{v \mid v \rightarrow u\} \\ children.u &= \{v \mid u \rightarrow v\} \\ anscestors.u &= \{v \mid v \rightarrow^+ u\} \\ descendants.u &= \{v \mid u \rightarrow^+ v\} \end{aligned}$$

A graph is said to be *cyclic* if $(\exists\, a :: a \rightarrow^+ a)$. A digraph without cycles is called a directed acyclic graph, or DAG. On a DAG one can define the following functions:

$$\begin{aligned} handles.(V, E) &= \{v \mid v \in V \ \wedge\ parents.v = \varnothing\} \\ leaves.(V, E) &= \{v \mid v \in V \ \wedge\ children.v = \varnothing\} \end{aligned}$$

A graph with only one handle is called a *rooted* graph, and the handle is called the *root*.

One can apply a labeling to the edges and vertices of a graph by using a labeling function, $f : V \rightarrow L$ for a vertex labeling, and $g : E \rightarrow L$ for an edge labeling, where L is the set of labels. An edge with label l is written as $a \rightarrow^l b$, which is defined as

$$a \rightarrow^l b \ \equiv\ a \rightarrow b \ \wedge\ l = \lambda(a, b)$$

where λ is the edge labeling function. A graph is *uniquely* labeled if the labeling function is bijective.

A *tree* is a rooted digraph where all nodes but the root, have one parent.

1.4 Informal program model

A compiler works between two languages, the input language, and the output language. To be able to process a file of the input language one needs to have a model that can describe any program written therein. Similarly, one needs to have a model that can describe any program written in the output language. To make the translation easier we strive to use the same model for both input, and output language.

A C program consists of one or more functions. However, this is of no concern to code generation. Code generation works on the smaller building blocks of a program. Anything that should be done on function level, should be done before code generation. For this reason the algorithms used in code generation process one building block at a time. It suffices to say that a program has a set of functions.

Next we need to define a model for a function. It is common practice to distinguish the concepts *data flow*, and *control flow* in a function. Consider the C program in figure 1.1. In this example, the control flow is determined by the **while**, and **if** statements. The control flow of a program can be represented by a graph, called the *control flow graph*. Figure 1.2 depicts the control flow graph of resample. The edges (B, C), (B, F), and (E, B) model the **while** statement, and the edges (C, D), (C, E), and (D, E) model the **if** statement. The labels at the edges are the conditions you see in the program and their inverse. Edges with the condition *true*have no label.

The nodes of the control flow graph are called *basic blocks*. A basic block contains that part of the data flow that should be executed when the basic block is executed. Basic block, D, will be executed when its parent, C, has been executed and the condition of the edge (C, D) evaluates

```
void resample(float *out, float *in, int len, float *pitch)
{
  float t=0.0;
  int i=0, j=0;
  while(i+1<len) {
    out[j] = in[i]*(1.0−t) + in[i+1]*t;
    j=j+1;
    t=t+pitch[j];
    if(1.0<t){
      t=t−1.0;
      i=i+1;
    }
  }
}
```

Figure 1.1: Audio wave resampler written in C.

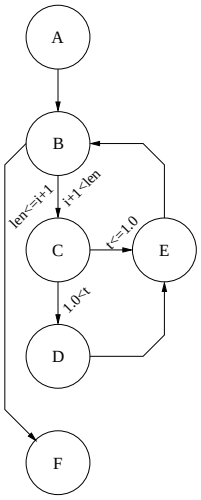


Figure 1.2: CFG of resample

to *true*. The following table gives a list of expressions for each basic block that will be evaluated when the basic block is executed.

Block	Expression
A	t=0.0
	i=0
	j=0
B	i+1<len
C	out[j] = in[i]*(1.0−t) + in[i+1]*t
	j=j+1
	t=t+pitch[j]
	1.0<t
D	t=t−1.0
	i=i+1
E	—
F	—

The data flow is determined by the expressions in this list. The actual flow is the order in which these expressions should be evaluated. For example, the flow of the variable t can be

```
t=0.0
out[j] = in[i]*(1.0−t) + in[i+1]*t
t=t+pitch[j]
1.0<t
t=t−1.0
out[j] = in[i]*(1.0−t) + in[i+1]*t
...
```

The data flow is modeled on two different levels, internal and external to the basic block. Let us start with the data flow external to the basic block. For this purpose we will see the basic block as a black box with values going in and values coming out, much like arguments and results of a function. Each value coming out of a basic block must also go into a basic block, unless we allow the existence of dead code. These value transfers can be modeled by a graph, which is done in the earlier stages of the compiler. During global register allocation they are mapped onto the finite set of registers, in order to decouple the basic blocks. The following lists a possible global register allocation on the basic blocks of the example.

$r0$=out, $r1$=in, $r2$=len, $r3$=pitch, $r4$=t, $r5$=i, $r6$=j

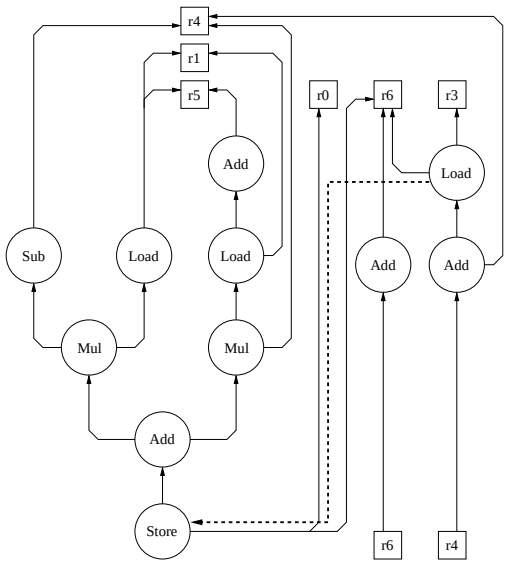


Figure 1.3: Data dependence graph of Basic block C

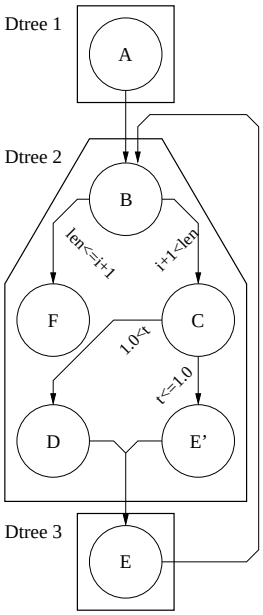


Figure 1.4: CFG with dtree

BB	Regs read	Regs written
A	—	r4, r5, r6
B	r2, r5	—
C	r0, r1, r3, r4, r5, r6	r4, r6
D	r4, r6	r4, r6
E	—	—
F	—	—

Register *r0* is not written in basic block C because it contains the *address* of out.

So a basic block consists of a set of registers coming in, a set of registers going out, and some expressions that compute the output from the input. The expressions are constructed from simple operations, each of which can be performed by an instruction. The basic block internal data flow models the value transfers between input, instructions, and output. Figure 1.3 depicts the data flow of basic block C. As you can see the basic block internal data flow is modeled by a graph, called the *data dependence graph*. The data dependence graph models where data is coming from, instead of where it is going to, which is the case for a *data flow graph*. The boxes depict input and output nodes, they are modeled by pseudo instructions, called respectively *rdreg* and *wrreg*. The solid arrows are the actual data dependencies between instructions.

Unfortunately there may be other dependencies that cannot be modeled by data dependencies. In this example pitch and out may point to the same address, if that is so the *Load* of pitch may not precede the *Store* of out. We model this kind of dependency with an *after constraint*, depicted in the figure with a dashed arrow.

Let us summarize for a bit. A program consists of a set of functions. A function is modeled by a control flow graph of basic blocks. Data edges between basic blocks are modeled implicitly by global registers, and a basic block is a data dependence graph of instructions with extra edges to enforce an order.

The introduction mentioned that some RISC machine support guarded instructions. A guarded instruction will only write its result if the supplied condition evaluates to true. They *can* be modeled by the control flow graph, but it helps to make them more explicit.

Consider a basic block with one in-edge, for example C in figure 1.2. The instructions in this basic block will be executed when the condition of the in-edge is *true*. In other words, each instruction is guarded by the condition of the in-edge. So if a basic block has a single in-edge

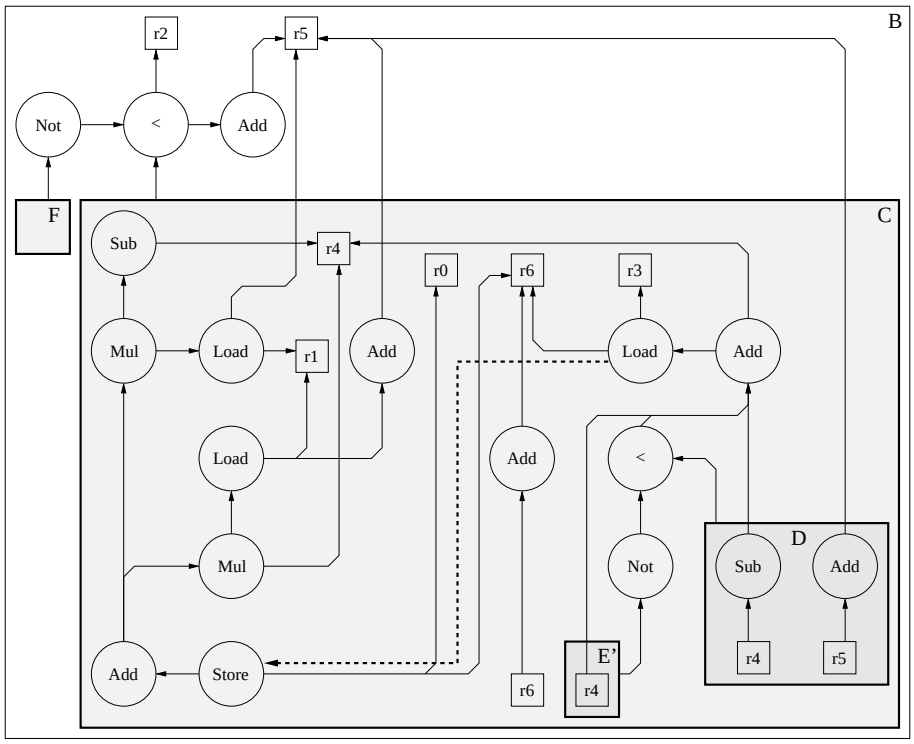


Figure 1.5: Data dependence graph of dtree 2

the instructions of the basic block can be transformed to guarded instructions. We make this possibility explicit by grouping the basic block and its parent together in a single unit, called a *decision tree* , or *dtree*. Other basic blocks can be added to this dtree, as long as they have a single in-edge, figure 1.4 depicts the control flow graph of resample with dtrees. As you can see, something funny has happened here, there is an extra basic block, E'. The reason for this will become clear in a moment. Because a dtree is acyclic, we can merge the data flow graphs of each basic block in the dtree to form a single data flow graph, as shown in figure 1.5. As you can see, basic block D writes registers $r4$ and $r5$, these registers contain the variables t and i respectively. But basic block C also needs to write to variable t, it cannot do so now because then we could have 2 writes to a register in a single trace. We solve this by adding an extra basic block E' in which we write register $r4$, and which is only executed when the condition of edge (C, D) evaluates to *false*.

Now we have a control flow graph of dtrees. A dtree is a control flow tree of basic blocks and a data flow graph of instructions. And a basic block is a subgraph of the data flow graph of the dtree.

We will formalize the following entities.

- Control flow graph
- Decision tree
- Basic block
- Data flow graph

1.5 Formal program model

We have a set of instruction instances[2] I, a set of basic blocks BB, and a function $bb : I \to BB$. We do not impose an order on the instructions yet, we will use the data flow graph for that.

The following definition formalizes the control flow graph. We restrict the labeling of an edge to instructions in the parent basic block, if it has a label.

▼ Definition 1

A *control flow graph* $CFG = (V, E)$ is a directed edge-labeled graph, such that:

- $V \subseteq BB$
- $\Sigma \subseteq I \cup \{\varepsilon\}$ is the set of labels
- if $\lambda : E \to \Sigma$ is the edge labeling function, and $u \to^l v$, then $bb.l = u \vee l = \varepsilon$

□

If there is control flow from basic block u to basic block v, (u, v) will be an edge in CFG. The edge set for the control flow graph in figure 1.2 is:

$$\{\ (A,B),(B,C),(B,F),(C,D),(C,E),(D,E),(E,B)\ \}$$

Some edges are labeled with an instruction instance, typically the instruction that computes the condition required to follow that control flow edge. When a vertex has only one out-edge, it will not have a condition, in that case the label will be ε, for example edge (A, B) in figure 1.2.

▼ Definition 2

A control flow graph $CFG = (V, E)$ is partitioned into *decision trees* $T_i = (D_i, C_i)$, such that:

- $D_i \in \{D_0, \cdots, D_n\}$ where $\{D_0, \cdots, D_n\}$ is a partitioning of V
- $C_i = E \cap (D_i \times D_i)$
- for $i \neq j \wedge u \in D_i \wedge v \in D_j$ we have $u \to v \ \Rightarrow\ u \in leaves.T_i \wedge v = root.T_j$

□

We model the ordering of instructions in a decision tree with an instruction dependence graph.

▼ Definition 3

An *instruction dependence graph*, $idg.T_i = (W, F)$, is a DAG of instruction instances such that:

- $W = \{j \mid j \in I \wedge bb.j \in T_i\}$
- $u \to_F v \ \Rightarrow\ bb.u \to_E^* bb.v$

□

$u \to v$ is to be interpreted as, "instruction u must be executed after instruction v". Some of the edges in the instruction dependence graph are after constraints, and some are data dependence edges. We distinguish the two using an edge labeling function μ.

$$\begin{array}{rcl} \mu & : & F \to \mathbb{B} \\ \mu.e & = & \left\{ \begin{array}{lcl} true & : & e \text{ is a data flow edge} \\ false & : & e \text{ is a not data flow edge} \end{array} \right. \end{array}$$

Using this function we define a *data dependence graph* as follows:

[2] I will often simply write *instruction* when it is clear that I mean *instruction instance*.

▼ Definition 4

Given an instruction dependence graph, $idg.T_i = (W, F)$, we define a *data dependence graph* as a graph $ddg.T_i = (V, E)$ such that:

- $V = W$
- $E = \{e \mid e \in F \wedge \mu.e\}$

□

We define a *data dependence edge* as follows:

▼ Definition 5

$$a \rightsquigarrow b \;\equiv\; a \to b \wedge \mu.(a, b)$$

□

We will write $\rightsquigarrow^+$ for the transitive closure, and $\rightsquigarrow^*$ for the transitive reflexive closure of $\rightsquigarrow$.

Chapter 2

An Overview of the Back-End

This chapter gives an overview of the code generator. It explains the phases that have been implemented, in what order they take place, and why they take place in that order.

The input of the code generator is a set of D-trees and some data declarations. The data declarations can be trivially translated to the target assembly language as long as the input provides the same types as the assembly file requires. This is usually not a problem, most assembly languages support data declarations for a 32 bit *word*, a 16 bit *half-word*, and an 8 bit *byte*. The one thing to remember is that the order of declarations should remain the same. This matter will not be discussed further in this report.

```
{ P is a set of dtrees }
Read machine description
{ P is a set of dtrees, R is a set of registers, M is a machine description }
for all T ∈ P →
  { T = (B,C) is a dtree }
  Back-end
  { S is a register allocated schedule of N emitable instructions }
  Emit S
done
```

This is a rough sketch of a code generator. At this stage we do not perform inter-dtree program transformations, these should be performed in the front-end of the compiler. Therefore we can simply generate code one dtree at a time.

All the actual work is done in the program called *Back-end* . The back-end generates a *schedule* S for T. A schedule for T is a sequence of instructions where each instruction in T occurs once, and after its children.

The schedule is *register allocated* for R if all instructions that have a result are assigned one register, in such a way that no instruction shall write to a register before all instructions data are data dependant on the previous writer have read it. We could say an instruction stops using a register after it's last data dependent parent.

An instruction is *emitable* if there exists one assembly instruction on the targeted machine that implements it.

The post-condition of *Back-end* is established in a number of steps,

- *Instruction selection* transforms unemitable instructions into emitable ones.
- *Instruction scheduling* transforms a DAG into a schedule.
- *Register allocation* assigns registers to all children in the schedule.
- *Dtree flattening* transforms a dtree of basic blocks into a single basic block.

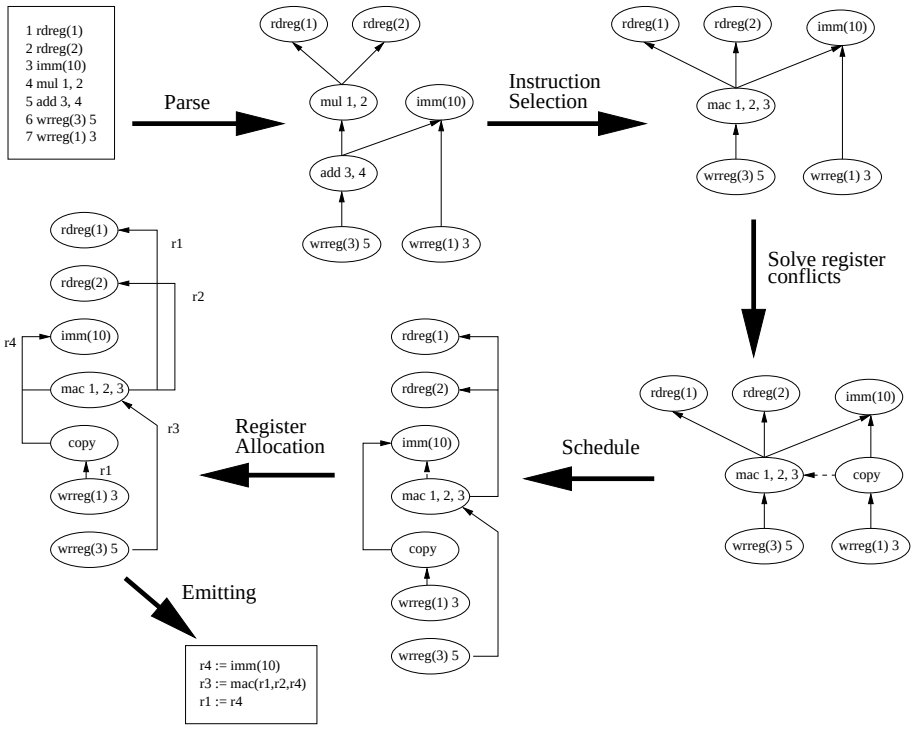


Figure 2.1: Overview of the backend

- *Global Register Conflict Solving.*

  Due to the fact that global register allocation has already been done, a dtree may be partially register allocated. Also, the front-end does not guarantee that a register allocated schedule is possible with the given register allocation. To this end, we may have to add instructions. This is done in this step.

Figure 2.1 depicts the order of these steps, dtree flattening has been left out for clarity reasons. The instructions `rdreg(n)` and `wrreg(n)` signify a read and a write to register n, respectively. The following sections will give an overview of these steps, starting with instruction selection.

2.1 Instruction selection

Instruction selection translates all instructions in a basic block from the input intermediate instruction set into the machine instruction set. These translations are usually very simple, eg.: one translation could replace 1 or 2 input instructions with 1 or 2 machine instructions, eg. in figure 2.1 a `mul` and `add` are replaced by a `mac`.

Due to large overlap in input instruction set and machine instruction set we have merged the two instructions sets. In this instruction set not all instructions can be written to the assembly file, so called *non-emitable* instructions. During instruction selection we must transform the instruction dependence graph such that all instructions are *emitable*.

We have the following specification for instruction selection.

{ b is a DAG of instructions }
Instruction selection
{ b is a DAG of emitable instructions }

2.2 Solve register conflicts

Since some of the instructions in the DAG have already been assigned a register, due to global register allocation, we have to make sure that all possible schedules are register allocatable. This is done by adding after constraints and register copies such that register usages will never overlap. In figure 2.1 before solving register conflicts, the instructions `rdreg(1)` and `wrreg(1) 3` use register 1 at the same time, this is solved by introducing a copy.

Global register conflicts are not the only form of conflicts however. There are other forms of conflicts caused by the presence of so-called 2-address instructions, ie. instructions that perform $a := a \oplus b$ instead of $a := b \oplus c$. How to find and solve register conflicts will be explained in chapter 4. The specification for the register conflict solver is as follows.

{ b is a DAG of emitable instructions }
Register conflict solving
{ b is a conflict free DAG of emitable instructions }

2.3 Instruction scheduling

Instruction scheduling is the process of finding a good sequence of instructions for a given DAG of instructions. The quality of a sequence is evaluated by a cost function. Our cost function is aimed at reducing register pressure to a certain level. We will not try to find the best sequence, since that problem is NP-complete. However, we do need to define when a sequence is good enough. In our case a sequence is good enough when it doesn't use more registers than we have. Observe that we are not particularly interested in finding a sequence that uses as few register as possible, there is no benefit in *not* using a register.

The specification for instruction scheduling is as follows.

{ b is a DAG of emitable instructions }
Instruction scheduling
{ $S.b$ is a schedule of emitable instructions }

We will use the *list scheduling* algorithm for this step. This algorithm is $\mathcal{O}(N\ logN)$, where N is the number of instructions, and assuming the cost function is $\mathcal{O}(1)$. We will derive the list scheduling algorithm in section 5.1.

2.4 Local register allocation

During local register allocation all instruction results of a scheduled basic block will be assigned a register. The resulting register allocation is valid when no result is prematurely overwritten, the result of an instruction must be available to all instructions that use it. Since we have a limited supply of registers it is not always possible to compute an assignment, when this is the case we need to add instructions such that it is possible. This can be done either by duplicating an expression, or by storing and reloading a value from memory.

At this stage we require that the partial allocation is valid, so global register conflicts must have been solved already. Instruction selection should also take place before register allocation, since that step may transform the dependence graph in such a way that the register allocation is no longer valid.

This gives us the following specification for local register allocation.

{ $S.b$ is a schedule of emitable instructions }
Local register allocation
{ $S.b$ is a register allocated schedule of emitable instructions }

In chapter 6 we will derive an algorithm for local register allocation, and we will present the Conservative Furthest First heuristic proposed in [FL98].

2.5 Dtree flattening

Dtree flattening transforms a dtree of basic blocks into a single basic block. This is done by adding a guard to all instructions of all basic blocks but the root. There are many places where one can put this step, before or after any of the other steps. This usually boils down to before or after all of the other steps. When we do it before all other steps we can probably generate better code, but it also adds a lot of complexity to the following steps. For that reason we have chosen to do it after the other steps.

We have the following specification for dtree flattening.

{ $(\forall\, b : b \in B : S.b$ is a register allocated schedule of emitable instructions) }
Dtree flattening
{ S is a register allocated schedule of emitable instructions }

This reduces complexity since now we can perform the other steps on each basic block separately.

2.6 The Back-end

To conclude this chapter we combine the specifications, giving us a more detailed specification of *Back-end* . The following chapters will discuss all steps in detail.

Back-end
{ $T = (B, C)$ is a dtree, R is a set of registers }
for all $b \in B \rightarrow$
{ b is a DAG of instructions }
Instruction selection
;{ b is a DAG of emitable instructions }
Global register conflict solving
;{ b is a conflict free DAG of emitable instructions }
Instruction scheduling
;{ $S.b$ is a conflict free schedule of emitable instructions }
Local register allocation
;{ $S.b$ is a register allocated schedule of emitable instructions }
end
;{ $(\forall\, b : b \in B : S.b$ is a register allocated schedule of emitable instructions) }
Dtree flattening
{ S is a register allocated schedule of emitable instructions }

Chapter 3

Instruction selection

During instruction selection we translate instructions of the input intermediate assembly language into instructions of the output language, as depicted in figure 3.1. This is not such a big step as it may seem. We are generating code for a RISC machine, and have tuned the intermediate language for that. This results in a large overlap, functionally speaking, between input and output instructions.

What characterizes an output instruction is that we know how to write it to an assembly file. This information is provided by the machine description. The machine description could also provide this information for input instructions, merging the two instruction sets. This is useful since, due to the large overlap, most output instructions are one-to-one mappings of input instructions.

The merging of the two instruction sets changes the problem a bit. We no longer need to translate the instructions, now we have to transform the instruction dependence graph into a graph of *emitable* instructions. An instruction is emitable if we know how to write it to an assembly file.

¿From chapter 2 we have the following specification.

{ b is a DAG of instructions }
Instruction selection
{ b is a DAG of emitable instructions }

We perform the transformations on the instruction dependence graph by means of graph rewrite rules. A graph rewrite rule matches a subgraph and conditionally replaces the root of the subgraph with a new graph. Since only the root instruction of the matching graph is replaced, these transformations may lead to *dead code*, code whose result will never be used. Therefore, after the transformations have been applied we must perform *dead code removal*.

The rewrite rules are a part of the machine description, although they are not written in the language presented in chapter 8. The machine description language of chapter 8 specifies how to emit instructions, thereby defining which instructions are emitable. The rewrite rules specify how to transform instructions into emitable instructions.

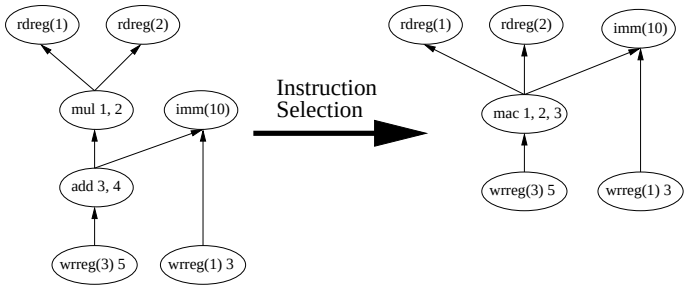


Figure 3.1: Instruction selection

We have developed a language to denote these rewrite rules, and a generator tool called *doggy*, which stands for DAG Optimizer Generator, to generate C code from these rewrite rules. This chapter will present the language for these rewrite rules, and how they are used in instruction selection.

3.1 Graph rewrite language

We will use the following notation to describe the language:

"foo"	signifies the literal input text *foo*
foo	signifies the non-terminal *foo*
(foo)	signifies a single occurrence of *foo*
{ foo }	signifies 0 or more occurrences of *foo*
{ foo }+	signifies 1 or more occurrences of *foo*.
foo//bar	signifies 1 or more occurrences of *foo*, separated by *bar*.
[foo]	signifies at most 1 occurrence of *foo*.
foo|bar	separates two alternatives *foo* and *bar*

The top-level description of a rewrite file, called *doggy* for DAG optimizer generator, is specified as follows:

```
doggy ::= "TYPES" { TypeDefinition }
          "MACHINE" "DESCRIPTION" { InstructionDescription }
          "EXTERNAL" { ExternalFunctionDefinition }
          "FUNCTIONS" { FunctionDefinition }
          "RULES" { RewriteRule }
```

In the `TYPES` section one must define the types that are used in the following sections. There is one mandatory type, `dep`, the type of a data dependence edge. Beside the types defined in this section there is another, internal, type `OPR`. This is the type of an instruction, eg, `add` is of this type.

The *InstructionDescription*s describe how instructions in a doggy file should be translated to C, ie: the name of the instruction, the number, order, and kind of arguments, and possibly some attributes the instruction has. For example.

```
add(dep,dep)          MOP_add(2,1)  COMMUTATIVE
```

This line describes the instruction `add` with two data dependency edges as arguments, `MOP_add` is the name it will be given in the generated code. The order of the arguments in the generated code is denoted by `(2,1)`, the second `dep` is the first argument. The `COMMUTATIVE` attribute allows the generator to swap the arguments of this instruction.

An *ExternalFunctionDefinition* defines a C function such that it can be used in the `FUNCTIONS` and `RULES` sections.

The *FunctionDefinition*s describe functions that can be used in the rewrite rules. For example

```
OPR get_reverse_op
  (x=eql)  -> neq,        /* equal -> not equal */
  (x=lt)   -> geq,        /* less than -> greater or equal */
  (x=neq)  -> eql,        /* not equal -> equal */
  ...
```

Given a comparison instruction `x` , this function computes the opposite comparison instruction.

3.1.1 Rewrite rules

The *RewriteRule* section describes the graph rewrite rules. The matching and replacement graphs are written as functions, a function representing a single instruction.

InsPattern ::= AppliedInstruction "(" [Pattern // ","] ")"
AppliedInstruction ::= Ident | "[" Ident // "," "]" | "ANY"

Ident is an identifier, which in the case of AppliedInstruction is an instruction name. An AppliedInstruction, as defined by its production rule, can be a single specific instruction, a set of instructions, or an arbitrary instruction. The arguments of the instruction are, again, patterns to be matched. An InsPattern is one option of the BasicPattern, we also wish to match an arbitrary dependency, and an integer argument.

Pattern ::= BasicPattern | Ident ["=" BasicPattern]
BasicPattern ::= InsPattern | "_" | Int

By naming vertices in a Pattern we can connect the replacement to the rest of the graph, and we can specify conditions on parts of the matching graph. A name should occur at most once in a Pattern, one can test equality of vertices in the condition of the replacement, as discussed below. The "don't care" pattern, "_", is used to leave unused leafs of the matching graph unnamed.

A Pattern can have a number of replacements, one of which will be chosen, based upon their condition and the order in which they appear. When no condition is met, no replacement will take place.

Rule ::= Pattern Rhs ";"
Rhs ::= { Replacement }+
Replacement ::= ["|" Expression] "->" Expression [Wherepart]

The first Expression is the condition that must be met before applying the rewrite rule. The second Expression is the replacement to be inserted when the pattern matches and the condition holds. The WherePart can be used to define variables that are used in the condition and in the replacement.

Wherepart ::= "WHERE" { Type Ident "=" Expression }

Expression is a C expression, using the same operators and precedence. See [ANS89] for the grammar of C expressions.

When a condition is met, and a replacement is to be made, the root of the replacement graph will inherit the parents of the root of the matching graph.

Some examples, taken from a doggy file used for ARM7 instruction selection and peephole optimizations.

- The ARM7 has an instruction called "reverse subtract", the assembly instruction `RSB a,b,c`, will perform $a := c - b$. This is a useful instruction since only the last argument can be an immediate value. In doggy, the instruction has the form `rsb (b,c)` , the result a remains unmentioned. The following rewrite rule introduces this instruction:

  ```
  sub (c, b) | IS_CONSTANT(c) -> rsb (b, c);
  ```

  `IS_CONSTANT` is a C function that determines whether or not dependency a is a constant value.

- The ARM7 also has a *multiply and accumulate* instruction, `MLA a,b,c,d` will perform $a := b * c + d$. The following two rules will introduce this instruction.

  ```
  add([imul,umul](b,c),d)        -> mla(b,c,d);
  sub(b,[imul,umul](c,imm(d)))   -> mla(c,imm(-d),b);
  ```

- A multiply instruction is an expensive instruction, it uses a lot of cycles, with respect to other instructions. When it is possible we want to replace the multiply instruction with another, cheaper, instruction. The following rule replaces a multiply by a *shift* instruction. A shift instruction performs a multiplication by a power of 2.

  ```
  [imul,umul](x,c) | IS_CONSTANT(c) && IS_POWER_OF_TWO(c)
      -> lsli(x,n)
      WHERE int n = GET_POWER_OF_TWO (c);
  ```

  The C functions `IS_POWER_OF_TWO` and `GET_POWER_OF_TWO` are used to determine if the value c is a power of two, and to compute $^2\log c$, respectively.

Doggy has also support, among other things, for typed edges, and overloaded instructions. These are beyond the scope of this report.

The generated graph rewriter provides a function that, given a vertex in the graph, will apply the first rule with a matching pattern and a condition that holds. The user must provide an implementation of the API through which the rewriter will examine and modify the graph.

The rewriter does not enforce an order in which the graph must be traversed. We traverse the graph until no more rewrite rules can be applied. This implies that the rewrite rules must be such that termination is guaranteed, ie. it must not be possible, in any way, to rewrite (part of) a replacement graph into the matching graph corresponding to the replacement. For example, suppose we have the following two rules.

```
[imul,umul](x,c) | IS_CONSTANT(c) && IS_POWER_OF_TWO(c)
    -> lsli(x,n)
    WHERE int n = GET_POWER_OF_TWO (c);
lsli(x,c) -> imul(x,1<<n) ;
```

These two rules obviously cause the rewriter to fail to terminate. Another issue is the completeness of the instruction selector, after instruction selection all instructions must be emitable. Since the rewrite rules and the emitable characteristic are both specified in the machine description the back-end itself cannot guarantee completeness of the instruction selector.

In our code generator we traverse the graph in a more or less arbitrary order. There are probably other strategies that sometimes lead to better results, but, since the number and size of the rewrite rules is usually small, this implementation has been good enough until now. The number and size of the rewrite rules is usually small, due to the fact that the intermediate instruction set of the input is already targeted towards RISC machines.

3.2 Instruction selection using rewrite rules

Some properties that are implemented in many different ways in instruction sets are addressing modes, and how conditions are used and computed. The two most common ways to compute conditions are by letting all instructions have the possibility to modify a condition flag register, in which case you have instructions like "*jump on 'greater than' condition*", or by having specific instructions to compute conditions, in which case you have instructions like "*is a greater than b*". In our intermediate instruction set we have chosen to use specific "condition" instructions, because that case has a more explicit data flow. Although it may seem obvious, if the target machine uses a condition flag register, we do not rewrite these condition instructions into what the target machines requires, yet. We may need the information modeled by the condition instructions to perform if-conversion.

An addressing mode the way an instruction computes a memory address. In the most simple form you can only specify the address using a register. With more complex addressing modes you can specify an address using a small expression of registers and constants, and sometimes with a side-effect on a register. The intermediate instruction set supports the following addressing modes.

direct (Rx), the address is stored in a register

indexed ($Rx + i$), a base address from a register incremented by an integer value

scaled ($Rx + Ry * f$), a base address incremented by a scaled offset. The integer scaling factor can be 2 or 4, as specified by the instruction.

In the ARM7 the concept of addressing modes is generalized, in that sense that nearly every instruction can have one of its register arguments shifted by an integer or register. For example, the following two rules add shifted arguments to `add` and `or` instructions, when possible.

```
add (a,x=[lsl,lsr,asr](b, c)) -> addx(a,b,OPERATOR(x),c);
or  (a,x=[lsl,lsr,asr](b, c)) -> orx(a,b,OPERATOR(x),c);
```

In the intermediate instruction set `ld32i` is an indexed load of a 32 bit value. The following rule introduces a scaled load with arbitrary factor.

```
ld32i (addx(a,b,op,c),0)   -> ld32_xi(a,b,op,c);
```

One problem of RISC machines, especially when the instructions are encoded in a 32 bit value, is how to load a large, or unknown, value. For known value we could introduce instructions to compute the large value, hereby introducing two or three instructions. For unknown values, for example labels, we cannot do this. In that case we must keep the value in data memory and load the value from there. The ARM7 has an instruction that allows one to rotate a 12 bit immediate value to other bit position. The following rule uses this instruction to rewrite the intermediate instruction `imm`.

```
imm(int a) | CAN_BE_ROTATER_IMM(a) && PHASE_NORM()
  -> sh_imm(base,rot)
WHERE
  int base = GET_IMMEDIATE_BASE(a)
  int rot  = GET_IMMEDIATE_ROTATE(a);
```

The `PHASE_NORM` function is used to distinguish phases in the rewriting of the entire graph.

See appendix A.2 for the instruction selection rewrite rules used for an ARM7 code generator.

3.3 Machine Description Requirements

Since the rewrite rules are a part of the machine description, this stage has no requirements with respect to the other part of the machine description.

Chapter 4

Solving Register Conflicts

As has been mentioned before, the dependence graph that is the input to the code generator is partially register allocated. This means that, if we do nothing about it, not every possible schedule of the graph will be a valid one, ie. one that can be register allocated. For example, instruction Y reads input to the DAG from register A, and instruction X writes output of the DAG also to register A. If a path $Y \rightarrow^{+} X$ exists, register A must hold two values at the same time, something we wish to avoid.

Another problem that might occur in the partially allocated graph is that an instruction is required to write its result to output register A and output register B.

We call these problems *register conflicts*. To solve these conflicts we may be forced to add instructions or edges to the DAG. The purpose of this stage is to make sure that all possible schedules the scheduler can produce are register allocatable. Figure 4.1 depicts this stage. In this example register 1 is required as argument to `mac` instruction and as result for the `imm` instruction, since this impossible a copy is added.

4.1 Simple register conflicts

The inputs and outputs of a DAG are modeled by the *rdreg* and *wrreg* pseudo instructions. A *rdreg* occurs before any other instruction, *wrreg*s occur last. A *rdreg* occurs only in the root basic block of a decision tree, a *wrreg* only in a leaf. A register can have at most one *rdreg* or *wrreg* in a basic block. Observe that we can have more than one *wrreg* of the same register in a decision tree, just not in a basic block. A *rdreg* is never connected to a *wrreg* of the same register.

Figure 4.2 depicts the conflicts we discuss in this section. We can identify the following elements in this picture.

- $\sqcup$ is a *rdreg*, a the letter inside the cup names the register that is read.
- $\sqcap$ is a *wrreg*, a the letter inside the cap names the register that is written.
- $\bullet$ is an arbitrary instruction
- $\rightarrow$ is a data dependency edge
- $\dashrightarrow$ is a path in the DAG

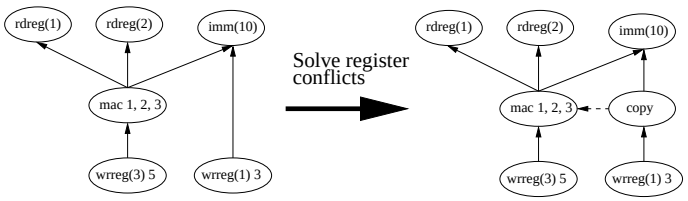


Figure 4.1: Solving register conflicts

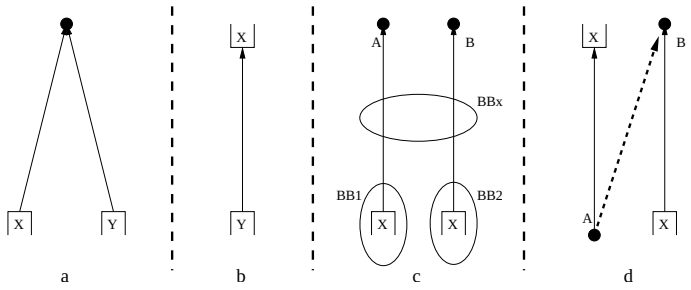


Figure 4.2: simple register conflicts

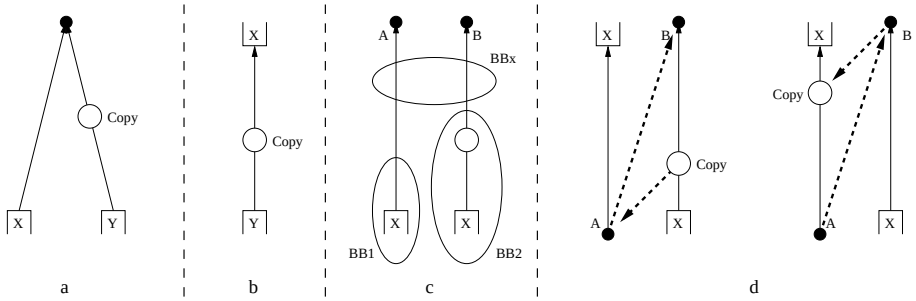


Figure 4.3: conflict solutions

An instruction that must write to two registers can be caused by only two things, the instruction has two *wrreg*s as parents, or, the instruction itself is a *rdreg* and has a *wrreg* as parent. An instruction with two *rdreg*s as children is not a conflict, each child represents a separate argument to the instruction.

A register that must hold two values can be caused by two *wrreg*s of the same register, or a *rdreg* and a *wrreg* of the same register. Two *rdreg*s of the same register cannot occur. Two *wrreg*s of the same register in itself is not a problem, it only becomes a problem if the register must hold both values at the same time, eg. if the children of the *wrreg*s are in the same basic block. Something similar holds for a *rdreg* and *wrreg* of the same register, this is only a problem if the parent of the *rdreg* will occur after the child of the *wrreg*.

Figure 4.3 depicts the solutions to these conflicts.

However, these are not all possible conflict situations. We can also have an indirect conflict, for example figure 4.5. For this case the simple solutions are not enough, we need a more systematic approach. But first we make things worse.

4.2 Introducing 2-address instructions

Not all register conflicts involve global registers, certain machines have so-called *2-address* instructions, an instruction whose result register is the same as one of its argument registers. These instructions may cause conflicts in the local register allocation. For example when the instruction is not the last consumer of its argument register.

2-address instructions propagate register requirements upwards or downwards, depending on where the requirement comes from. Figure 4.4 displays one register conflict involving a 2-address instruction. Instruction C writes its result to register X, and since it is a 2-address instruction (represented by the arc), B must also write its result to X. This conflict is similar to the conflict in figure 4.2d. The solution there was to add a copy either before A, or below B. In this case the second option might not work, since $C \rightarrow^{+} A$ does not necessarily hold.

The characteristics of the *rdreg*s and *wrreg*s do not apply to 2-address instructions. So we can have two instructions in the same basic block that propagate the same register upwards, or

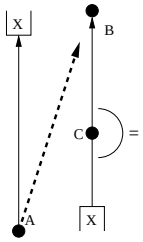


Figure 4.4: A register conflict with a 2-address instruction

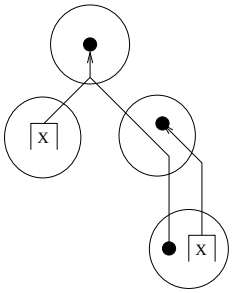


Figure 4.5: An indirect conflict

downwards, or both. In the most extreme case, we can have four instructions, $c \rightsquigarrow a$, and $d \rightsquigarrow b$, where a and c propagate the register downwards, and c, and d upwards. As we shall see later, we may have to introduce no less than four register copies to solve this conflict.

The information of how a register is propagated is quite important, if in the previous example c and d do not propagate a register, the conflict can be solved with one register copy. If we did not have the propagation information we could not distinguish the two.

4.3 Problem specification

In order to identify 2-address instructions we loosely define the function *modify* as follows, $modify.a.b$ holds iff a is a 2-address instruction, modifying the result of b.

¿From chapter 2 we have the following specification.

{ b is a DAG of emitable instructions }
Register conflict solving
{ b is a conflict free DAG of emitable instructions }

We define an equivalence relation ($\sim$) on the instructions of DAG b, such that $i \sim j$ means that instruction i will write to the same register as instruction j. Using the short hand $reg.r = \{rdreg(r), wrreg(r)\}$, we define $\sim'$ as follows:

$$\begin{aligned} j \rightsquigarrow i \ \wedge \ modify.i.j \quad &\Rightarrow \quad i \sim' j \\ (\exists\, r :: i, j \in reg.r) \quad &\Rightarrow \quad i \sim' j \end{aligned}$$

and $\sim$ as the transitive, reflexive, symmetrical closure of $\sim'$. Each instruction in an equivalence class writes its result to the same register. Therefore *rdregs* and *wrregs* of different registers must not be in the same equivalence class:

$$SingleReg: \quad (\forall\, i, j, r, s : i \sim j \ \wedge \ j \in reg.r : i \in reg.s \ \Rightarrow \ s = r)$$

This can easily be established by adding register copies above *wrreg*s or below *rdreg*s. Observe that this takes care of the cases (a) and (b) of figure 4.2.

What remains are the conflicts concerning instructions that may write to the same register at the same time. Two instructions that write to the same register must be ordered such that one will write after all uses of the other:

$$NonOverlapping: \quad a \sim b \ \Rightarrow \ (\forall\, c, d : c \rightsquigarrow a \ \wedge \ d \rightsquigarrow b : b \rightarrow^* c \ \vee \ a \rightarrow^* d)$$

A DAG (V, E) is conflict free iff:

$$SingleReg \ \wedge \ NonOverlapping$$

We propose the following program:

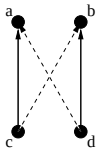


Figure 4.6: the basic conflict

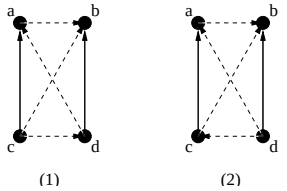
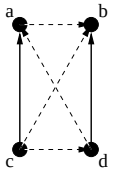


Figure 4.7: The two worst cases



Insert a register copy of a

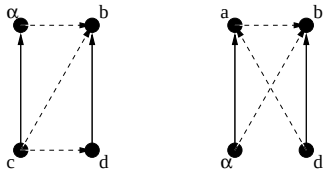


Figure 4.8: An example step

{ b is a DAG of emitable instructions }
Establish SingleReg
{ b is *SingleReg* }
Solve conflicts
{ b is a conflict free DAG of emitable instructions }

We will not go into detail on *Establish SingleReg* here, since there hardly is any detail worth mentioning. The rest of this chapter will discuss the second step.

4.4 Solving a conflict

A pair of data edges are potentially in conflict if *NonOverlapping* does not hold. That means that in any such situation we have $d \rightsquigarrow a$, $c \rightsquigarrow b$, $b \not\rightarrow^+ c$, and $a \not\rightarrow^+ d$. The straight forward solution is to add either an edge $a \rightarrow d$ or $b \rightarrow c$. However this is not possible when $d \rightarrow^+ a$ respectively $c \rightarrow^+ b$ exist, since that will introduce cycles, something we do not want in a DAG. When both these edges exist we have a conflict that can only be solved by introducing register copies. In any conflict situation the following holds:

$$\textit{Conflict}: \quad a \sim b \wedge c \rightsquigarrow a \wedge d \rightsquigarrow b \wedge d \rightarrow^+ a \wedge c \rightarrow^+ b$$

This is depicted in figure 4.6.

In the worst case there is also an edge between a and b, and an edge between c and d, and all instructions propagate the register. This boils down to two cases, depicted in figure 4.7.

$$\textit{Case 1}: \quad a \rightarrow^+ b \wedge c \rightarrow^+ d$$

$$\textit{Case 2}: \quad a \rightarrow^+ b \wedge d \rightarrow^+ c$$

We will solve these conflicts one step at a time. There are two basic steps we can perform.

- Insert a register copy, α, of a, such that $c \rightsquigarrow \alpha \rightsquigarrow a$, or similarly for b. We will use Greek letters to denote register copies.
- We can add an edge $a \rightarrow d$, or $b \rightarrow c$ to solve the conflict. This is only possible if the edge doesn't introduce a cycle.

Since we change the graph, we must recompute the equivalence relation after each step.

Introducing a register copy will, in the worst case, result in two new situations, eg.: if α is a copy of a, we get the situation

$$a \sim b \wedge \alpha \rightsquigarrow a \wedge d \rightsquigarrow b \wedge d \rightarrow^{+} a$$

and the situation

$$\alpha \sim b \wedge c \rightsquigarrow \alpha \wedge d \rightsquigarrow b \wedge c \rightarrow^{+} b$$

When we have $a \rightarrow^{+} b$ we also get $\alpha \rightarrow^{+} b$, and when we have $d \rightarrow^{+} c$ we also get $d \rightarrow^{+} \alpha$. Figure 4.8 depicts an example of a step introducing a register copy.

One special case is when a or c is a register copy itself, eg. if a is a register copy, say β, the first situation becomes:

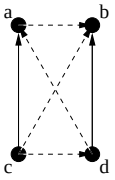
$$\beta \sim b \wedge \alpha \rightsquigarrow \beta \wedge d \rightsquigarrow b \wedge d \rightarrow^{+} \beta$$

This is a basic conflict where both a and c are register copies. In this situation we require that a is not related to any other instruction:

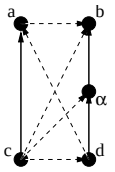
$$\alpha \rightsquigarrow \beta \;\Rightarrow\; \neg(\exists\, i :: i \sim \beta)$$

Which means that this situation is not a conflict, since $\beta \sim b$ does not hold. Simply put, a data edge between two copies can never be in conflict with another data edge.

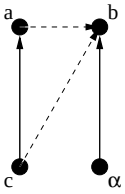
Let us investigate *Case 1.*



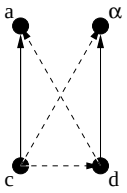
1. Insert a copy of b. There is an edge between c and α, since $c \rightarrow^{+} \alpha \;\Leftarrow\; c \rightarrow^{+} d \wedge d \rightsquigarrow \alpha$



2. This situation contains two conflicts, $c \rightsquigarrow a$ with $\alpha \rightsquigarrow b$, and $c \rightsquigarrow a$ with $d \rightsquigarrow \alpha$. We consider the two conflicts separately.

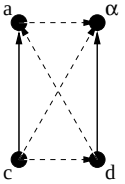


(a)



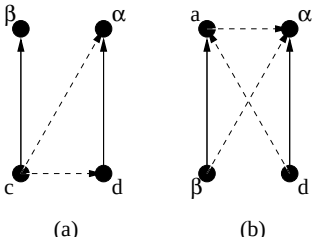
(b)

3. Add the edge $a \rightarrow \alpha$, solving case a. Since both cases are part of the same instruction dependence graph, we have to add the edge to both cases. We continue with case b.



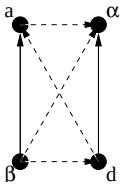
4. This situation is different from the first, since this is not a worst case situation. In the worst case situation all instructions propagated a register, in this case α does not, since it is a register copy.

   Observe that inserting a copy of α does not help since that results in the same situation. We insert a copy of a.

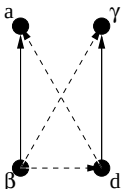


(a) (b)

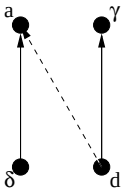
5. Add the edge $\beta \rightarrow d$ to both cases, solving case a. We continue with case b.



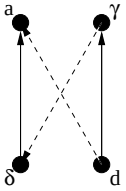
6. Insert a copy of α, this introduces an edge, $\gamma \rightsquigarrow \alpha$, between two copies. Since a data edge between two copies does not conflict with any other edge, that part of the conflict is solved. We continue with the other situation.



7. Insert a copy of a, this introduces an edge $\beta \rightsquigarrow \delta$, we continue with the other situation.



8. Add the edge $\gamma \rightarrow \delta$, solving *Case 1*



Swapping steps 2 and 3 with steps 4 and 5 will give the same result, save the names for the register copies. Swapping steps 6 and 7 will also give the same result.

Figure 4.9 depicts the original graph with the added register copies and edges. This solution will produce the following schedule.

b ; α ; a ; δ ; γ ; d ; β ; c

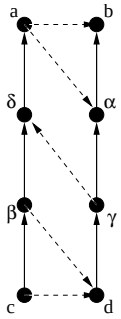
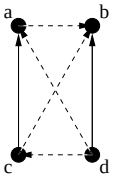


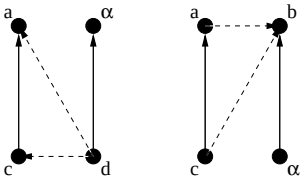
Figure 4.9: Case 1 conflict solution

Case 2 can be solved as follows:

1. Initial situation.

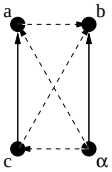


2. Insert a copy of b.

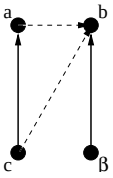


(a) (b)

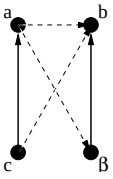
3. Add the edge $\alpha \to c$ to both cases, solving case a. We continue with case b.



4. Insert a copy of b, this introduces an edge, $\alpha \rightsquigarrow \beta$, between two copies. We continue with the other situation.



5. Add the edge $a \to \beta$, solving *Case 2*.



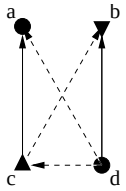


Figure 4.10: A simple conflict

If, in step 3 we add the edge $a \rightarrow \alpha$ instead of $\alpha \rightarrow c$ we end up with a similar solution.

We have not used the direction of propagation in these solutions, since, in the worst case, all instructions, except for copies, propagate registers. If we are not dealing with a worst case conflict we want to find a solution with as little copies as possible. What we have done upto now is insert copies to impose an order on the use of a register. We also want to insert copies such that the equivalence class is minimal.

Take the situation depicted in figure 4.10, The symbols ∇ and Δ indicate downward and upward register propagation respectively. To solve the conflict, we can either add a copy of a or b. Suppose we add a copy α of a, then $a \sim b$ will no longer hold, but due to the upward register propagation of c, $\alpha \sim b$ will hold and since $d \rightarrow^+ \alpha$, instructions α, b, c, and d are in a conflict situation.

If we add a copy β of b, then $a \sim b$ will still hold, but since $\beta \rightarrow^+ a$ does not hold a, b, c, and β are not in a conflict situation. Since d does not propagate a register upwards $a \sim \beta$ does not hold, the conflict is solved. The best solution to this conflict is to add a copy of b.

The steps that need to be taken to solve such a simpler conflict are the same as for the worst cases, the difference is that we may be able to solve the conflict early by adding edges and copies at the right places.

4.5 Solving all conflicts

We have shown that an individual conflict can be solved. We still need to investigate whether that means that all conflicts in a DAG can be solved. Since we add edges and instructions to the graph we may introduce new conflicts.

The solutions given in the previous section can insert up to two register copies in one data edge of the original DAG. We have already seen that an edge between two copies is unrelated to any other edge, this means that we never insert a copy between two copies. When all edges of the original DAG have had two copies inserted, all edges are unrelated, and hence no conflicts exist. When there are two original edges left, only they can have a conflict.

So we can solve all conflicts one by one, without having to worry about termination.

One thing we haven't looked at is the order in which conflicts are solved. Conflicts can be related, so solving one conflict may also solve another, but it may also make another conflict more difficult. When there are many conflicts in a graph, the order in which we solve the conflicts will become more important. There are not many RISC machines with a lot of 2-address instructions, in fact you'll find it hard to find even one. With only a few 2-address instructions, the amount of conflicts in a graph will be too little to make the order of solving them important.

It should be noted that the use of the approach discussed in this chapter has uncovered a missing conflict solution in the algorithm that was used, and thought to be free of such bugs.

4.6 Machine Description Requirements

Solving register conflicts requires two things from the machine description. Obviously, we need to know which instruction performs a register copy. The register copy is recognized by the input instruction name of the instruction, which should be `ident`.

We also need to know whether an instruction is 2-address or not. this is done in the machine description language using a `require` statement, see chapter 8.

Chapter 5

Scheduling

Instruction scheduling is used to put the instructions of a DAG into a sequence, as depicted in figure 5.1. The order in which the instructions are put is relevant here, a wrong order can cause a bad register pressure, which may lead to spilling instructions added during local register allocation.

For certain processors interlocking is another reason to perform instruction scheduling. Interlocking occurs when the processor pipeline stalls due to a read and write of the same register occur at the same time. We have chosen to ignore this goal for the moment, since the processor we are writing a back-end for, the ARM7, does not have interlocking.

The scheduling algorithm we use is a list scheduling algorithm, a proof of which can be found in the next section. This chapter will also discuss several scheduling heuristics, including my own which has been targeted towards reducing register pressure.

5.1 List scheduling

The list scheduling algorithm constructs a sequence of elements, often called a schedule, S, from a DAG $G = (V, E)$. In the following we will often use S, if it is possible, where the set of its elements is meant. A schedule of G is a total order such that for any $a, b \in V$ we have $a \sqsubset b \Leftarrow b \rightarrow a$, with no duplicate elements, $(\forall\, a, b : b \sqsubset a : a \neq b)$. This gives us the specification we have seen in section 2.3.

Pre: G is a DAG
Post: $S = V \wedge (\forall\, a, b : a \rightarrow b : b \sqsubset a) \wedge (\forall\, a, b : b \sqsubset a : a \neq b)$

Besides the schedule, we have a set of unscheduled elements, U. The following invariants hold for this set:

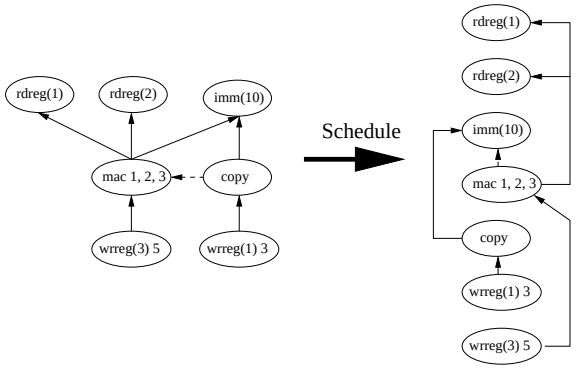


Figure 5.1: Scheduling

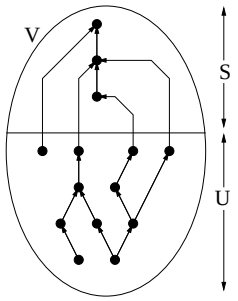


Figure 5.2: List scheduling

P1: $S \cap U = \varnothing$
P2: $S \cup U = V$

See figure 5.2 to get an idea of how S, U, and V are related. We overload the operator $\sqsubset$ to work on both scheduled and unscheduled elements:

$$a \sqsubset b = \begin{cases} a \in U & \mathit{false} \\ a \in S \wedge b \in U & \mathit{true} \\ a \in S \wedge b \in S & i < j \mid S.i = a \wedge S.j = b \end{cases}$$

We choose the following predicates as loop invariant:

P3: $(\forall\, a, b : a \in S \wedge b \in V \wedge a \to b : b \sqsubset a)$
P4: $(\forall\, a, b : a, b \in S \wedge b \sqsubset a : a \neq b)$

And for the guard we use $U \neq \varnothing$

Initially we have $S = [\,]$ and $U = V$. This trivially satisfies P1, P2, P3, and P4. We add an element best from U to the schedule using the statement.

$\mathsf{S} := \mathsf{S} +\!\!+ [\mathsf{best}]$

We will use σ as a shorthand for this substitution, so instead of P1.$(S := S +\!\!+ [best])$ we will write P1.(σ). Let us see what happens when we add the element best to the schedule:

P1.(σ)
$\equiv$ { substitution }
$(S +\!\!+ [best]) \cap U = \varnothing$
$\equiv$ { $\cap$ over $+\!\!+$ }
$S \cap U = \varnothing \wedge best \notin U$
$\equiv$ { P1 }
P1 $\wedge$ $best \notin U$
■

Hence, σ becomes $\mathsf{S}, \mathsf{U} := \mathsf{S} +\!\!+ [\mathsf{best}], \mathsf{U} \setminus \{\mathsf{best}\}$. Trivially, this statement does not affect invariant P2. Next we investigate invariant P3.

P3.(σ)
$\equiv$ { substitution }
$(\forall\, a, b : a \in (S +\!\!+ best) \wedge b \in V \wedge a \to b : b \sqsubset a)$
$\equiv$ { split off $a = best$ }
$(\forall\, a, b : a \in S \wedge b \in V \wedge a \to b : b \sqsubset a) \wedge$
$(\forall\, b : b \in V \wedge best \to b : b \sqsubset best)$
$\equiv$ { P3, definition of *children* }

$$
\begin{array}{ll}
 & \text{P3} \ \wedge (\forall\, b : b \in \mathit{children.best} : b \sqsubset \mathit{best}) \\
\equiv & \quad \{ \text{ definition of } \sqsubset \ \} \\
 & \text{P3} \ \wedge \ \mathit{children.best} \subseteq S
\end{array}
$$

■

So we cannot pick an arbitrary element best from U, only one whose children have already been scheduled. Last we investigate the invariant P4.

$$
\begin{array}{ll}
 & \text{P4}.(\sigma) \\
\equiv & \quad \{ \text{ substitution } \} \\
 & (\forall\, a, b : a, b \in S +\!\!+ [\mathit{best}] \ \wedge \ b \sqsubset a : a \neq b) \\
\equiv & \quad \{ \text{ split off } a = \mathit{best},\ b \neq \mathit{best} \ \Leftarrow \ a \in S +\!\!+ [\mathit{best}] \ \wedge \ b \sqsubset a \ \} \\
 & (\forall\, a, b : a, b \in S \ \wedge \ b \sqsubset a : a \neq b) \ \wedge \ (\forall\, b : b \in S \ \wedge \ b \sqsubset \mathit{best} : \mathit{best} \neq b) \\
\equiv & \quad \{ \text{ P4}, \ b \sqsubset \mathit{best} \ \Leftarrow \ \mathit{best} \notin S \ \} \\
 & \text{P4} \ \wedge \ (\forall\, b : b \in S : \mathit{best} \neq b) \\
\equiv & \\
 & \text{P4} \ \wedge \ \mathit{best} \notin S \\
\equiv & \quad \{\mathit{best} \in U\} \\
 & \text{P4}
\end{array}
$$

■

This gives us the following program:

```
{ G is a DAG }
S, U := [ ], V;
do  U ≠ ∅ →
  best := a ∈ U | children.a ⊆ S;
  S, U := S ++ [best], U \ best;
od
{ S is a schedule for G }
```

This program can be improved by keeping track of the set $\{a \in \mathrm{U} \mid \mathit{children.a} \subseteq \mathrm{S}\}$, the set where best is taken from. We call this set the *candidate set*. See figure 5.3 to get an idea of how C is related to the other sets.

P5: $\quad C = \{a \in U \mid \mathit{children.a} \subseteq S\}$

The initial value of C can be found using the initial values of S and U.

$$C = \{a \in V \mid \mathit{children.a} \subseteq \varnothing\} = \mathit{leaves.G}$$

We change the guard of the loop to $C \neq \varnothing$ in the hope that we can remove U from the program. This is not a problem since it can be shown that $C = \varnothing \ \Rightarrow \ U = \varnothing$.

For the invariant P5 we investigate the change to the set C.

$$
\begin{array}{ll}
 & C.(\sigma) \\
= & \quad \{ \text{ substitution } \} \\
 & \{a \in (U \setminus \{\mathit{best}\}) \mid \mathit{children.a} \subseteq (S +\!\!+ [\mathit{best}])\} \\
= & \quad \{ \text{ split off } a \neq \mathit{best} \ \} \\
 & \{a \in U \mid \mathit{children.a} \subseteq (S +\!\!+ [\mathit{best}])\} \ \setminus \ \{\mathit{best}\} \\
= & \quad \{\mathit{best} \in \mathit{children.a} \ \vee \ \mathit{best} \notin \mathit{children.a}\} \\
 & \big(\{a \in U \mid \mathit{children.a} \subseteq (S +\!\!+ [\mathit{best}]) \ \wedge \ \mathit{best} \notin \mathit{children.a}\} \cup
\end{array}
$$

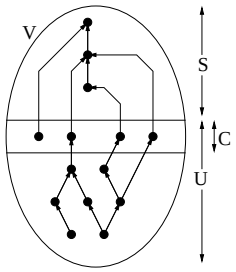


Figure 5.3: List scheduling with candidate set

$$
\begin{array}{ll}
 & \{a \in U \mid children.a \subseteq (S +\!\!+ [best]) \wedge best \in children.a\}\ )\ \setminus\ \{best\} \\
= & \quad \{best \notin children.a \ \Rightarrow\ children.a \subseteq S,\ children.a \subseteq S \ \Rightarrow\ a \in C \vee a \in S \Rightarrow best \notin children.a\} \\
 & (\ \{a \in U \mid children.a \subseteq S\} \cup \\
 & \ \ \{a \in U \mid children.a \subseteq (S +\!\!+ [best]) \wedge best \in children.a\}\ )\ \setminus\ \{best\} \\
= & \quad \{best \in children.a \ \equiv\ a \in parents.best,\ parents.best \subseteq U\ \} \\
 & (\ \{a \in U \mid children.a \subseteq S\} \cup \\
 & \ \ \{a \in parents.best \mid children.a \subseteq (S +\!\!+ [best])\}\ )\ \setminus\ \{best\} \\
= & \quad \{\text{ definition of } C,\ a \in parents.best \ \Rightarrow\ a \neq best\ \} \\
 & (C \setminus \{best\}) \cup \{a \in parents.best \mid children.a \subseteq S +\!\!+ [best]\}
\end{array}
$$

■

¿From this we can conclude that σ should be the following:

S, U, C := S ++ [best], U \ {best}, (C \ {best}) ∪ {$a \in parents.$best | $children.a \subseteq$ S ++ [best]}

So to add an element to the schedule we should use the statement as shown above. However, it can be simplified somewhat. Observe that the set U, is only used in the assignment to itself, and since the post condition has no requirements for this set, it can be removed from the program, resulting in the code of figure 5.4.

```
{ G is a DAG }
C := leaves.G;
S := [ ];
do  C ≠ ∅ →
  let best ∈ C;
  S, C := S ++ [best], (C \ {best}) ∪ {a ∈ parents.best | children.a ⊆ (S ++ [best])};
od
{ S is a schedule for G }
```

Figure 5.4: The list scheduling algorithm

Next we have to fill in the statement **let** best ∈ C . This is typically done by computing a cost for each element in C, and selecting the one with the lowest cost. This boils down to the following statement:

best := $a \in C \mid (\forall\, b : b \in C : cost.(a, S) \leq cost.(b, S))$

Given this list scheduling algorithm, we have to do two things. We have to define the elements we need to schedule, and we have define the cost function.

5.2 Computing the register pressure

The goal of our instruction scheduler is to keep the register pressure in a basic block below a certain level. The register pressure at an instruction is defined as the number of registers that are in use upon termination of that instruction. We formalize this as follows, instructions are assumed to be in total order.

$$\begin{aligned} kill.d &= i \mid i \rightsquigarrow d \wedge \neg(\exists\, j : j \rightsquigarrow d : j \rightarrow^{+} i) \\ liveset.d &= \{i \mid d \rightarrow^{*} i \wedge kill.i \rightarrow^{+} d\} \\ pressure.d &= |liveset.d| \end{aligned}$$

In words, $kill.d$ is the instruction that uses the result of instruction d last, $liveset.d$ is the set of instructions that are defined, but not killed at instruction d, and $pressure.d$ is the number of values that are live at instruction d. Also the range $[d, kill.d)$ is called the *live-range* of d.

In figure 5.5, instruction F is the *kill* of instruction D. The liveset at instruction D is $\{A, C, D\}$, and therefore the *pressure* at instruction D is 3. The live-range of D is $[D, F)$.

We define the pressure of a schedule as the maximum pressure of its elements:

$$pressure.S = (\uparrow i : i \in S : pressure.i)$$

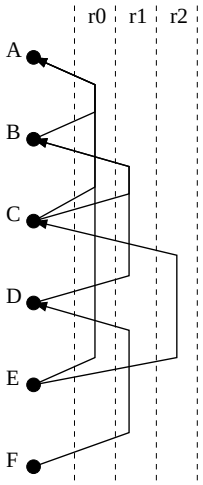


Figure 5.5: Register pressure

Using this function we can formalize the goal of the scheduler as follows

$$pressure.S \leq N$$

where N is the number of available registers.

5.3 Scheduling heuristics

As can be seen in figure 5.4, the *cost* function is used to determine which instruction is scheduled from all candidates. This *cost* function must reflect the goal we have in mind for scheduling. We are trying to optimize the register pressure of a schedule. We do not need to minimize the register pressure, we just need to keep it below a certain level, eg. the amount of actual registers.

Warren describes a heuristic for an instruction scheduler for the IBM RISC System/6000 Processor in [War90]. His scheduler has an extra goal, minimizing *interlocks*. An interlock occurs when an instruction needs to read from a register where the preceding instruction still needs to write to. For example, suppose the pipeline architecture is *Fetch, Decode, Read* (registers), *Execute, Write*. Then an execution of a sequence of instructions will look like this:

Clock	t	t+1	t+2	t+3	t+4	t+5	t+6
Ins i	Fetch	Decode	Read	Execute	Write	—	—
Ins i+1	—	Fetch	Decode	Read	Execute	Write	—
Ins i+2	—	—	Fetch	Decode	Read	Execute	Write

So in clock cycle $t + 4$ instruction i writes its result back into a register, but instruction $i + 1$ reads the registers it requires in cycle $t + 3$. So if instruction $i + 1$ uses the result of instruction i, it will have to wait until instruction i is finished. This is what we call an interlock.

For ease of presentation the heuristic is presented as a subsetting process. Starting with the set of candidates:

1. *Refine the subset to those instructions with the smallest earliest-time.*

   This is used to prevent interlocks as much as possible. When an instruction is scheduled, its children are given an earliest time such that no interlock will occur.

2. *Refine the subset to those instructions with the largest distance to the root.*

   The distance is measured in clock cycles.

3. *Refine the subset to instructions of the most important class.* Where the classes are as follows, in decreasing importance:

   (a) Register copies

   (b) Instructions with no result

   (c) Most instructions

   (d) Loads (from memory)

   (e) Instructions with no dependencies

4. *Refine the subset to those instructions that introduce the largest number of new candidates.*

5. *From the subset, select that instruction that came first in the input.*

Tiemann describes the instruction scheduling heuristics used in the GNU Compiler Collection, also known as the Haifa instruction scheduler, in [Tie89]. Since then the heuristics have been slightly changed. This scheduler also takes into account interlocks, but is used for processors without interlocking as well. GCC version 2.95.2 uses the following instruction scheduling heuristics:

1. *Refine the subset to those instructions with the largest distance to the root.*

   The distance is measured in clock cycles.

2. *Refine the subset to those instruction that decrease the register pressure most*

3. *Refine the subset to instructions of the most important class.* Where the classes are as follows, in decreasing importance:

   (a) Independent of last scheduled instruction

   (b) Non-data dependent on last scheduled instruction

   (c) Data dependent on last scheduled instruction

4. *Refine the subset to those instructions with the most children*

5. *From the subset, select that instruction that came first in the input.*

With the hope of getting a better schedule, we will take a look at a heuristic that uses a cost function that is more dependent on the schedule being generated. The downside of this is that the cost function is no longer a constant time function.

1. *If present, select an instruction that does not increase the register pressure*

2. *If present, select an instruction without children*

   For example, stores. These instructions need not necessarily decrease register pressure, a result can be stored more than once.

3. *Refine the subset to those instructions that introduce new candidates.*

   Selecting an instruction that introduces no new candidates at this point can only increase register pressure, there is no advantage.

4. *Refine the subset to those instructions that contribute to the expression with the highest coverage.*

   We prefer to finish scheduling one expression before start with another. The coverage is measured in number of instructions.

5. *Refine the subset to those instructions that contribute to the largest expression.*

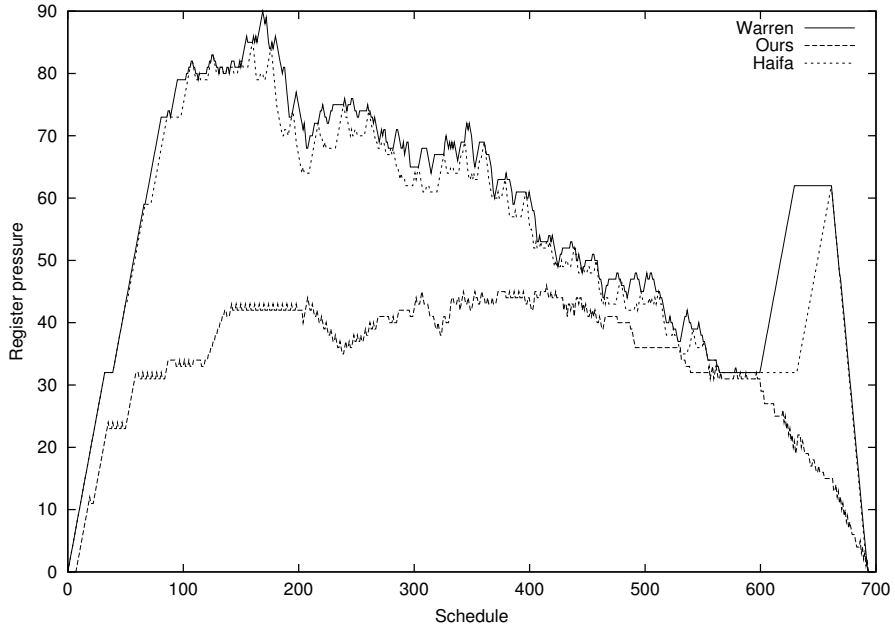


Figure 5.6: Register pressure comparison on the function dct32 of the MPEG audio decoder, MAD

6. *Refine the subset to those instructions with the longest path to an expression root.*

   Instructions with a long path to an expression root will have a higher chance of a short life time than instructions with a short path to an expression root.

Figure 5.6 depicts the register pressure at each instruction in a schedule of a particularly nasty data dependence graph. Vertically you have the register pressure in number of registers, and horizontally you have the schedule by instruction index. There are a few things that attract attention in this graph.

- The sudden increase in register pressure at the end for the Haifa and Warren heuristic.
- The steepness in the beginning for the Haifa and Warren heuristic.
- The "steps" in the beginning for our heuristic.

The sudden increase in register pressure is caused by several *load–store* combinations. The first heuristic used in the Haifa and Warren scheduler is the distance to an expression root, preferring instructions with greater distance. When you have several *load–store* combinations, the loads are all at a greater distance than the stores, with the effect that first all loads are scheduled, and then all stores.

To prevent interlocking, the heuristic used by Tiemann and Warren will try to schedule several expressions at the same time, since different expressions are not data dependent. The downside is that when you get a data dependence graph with a lot of expressions, they will still try to schedule them at the same time resulting in a register pressure explosion. This is clearly visible in the graph between instruction 0 and 100.

The "steps" in the register pressure of the schedule generated with our heuristic are caused by one big expression in the data dependence graph that has a lot of leaves at the same distance. This would be improved if we try to schedule one subexpression at a time, but this is not a trivial thing to do, and is left as future work.

The graph of figure 5.6 was generated from the highly optimized function dct32 of the MPEG audio decoder MAD[Les]. Since this is a critical function for the audio decoding, a reduction in register pressure is quite relevant. See chapter 9 for more benchmarks.

5.4 Machine Description Requirements

There are no machine description requirements for instruction scheduling.

Chapter 6

Local Register Allocation

After instruction scheduling we can assign registers to vertices of the data dependence graph, as depicted in figure 6.1. We cannot do this arbitrarily since a register can only contain one value at a time. In chapter 5 we were introduced to the concept of *live-ranges*, the period in which a register is used for one value. When we do not have enough registers we will need to split some live-ranges, by either duplicating instructions, or storing values to memory.

The specification of local register allocation is as follows.

{ $S.b$ is a schedule of emitable instructions }
Local register allocation
{ $S.b$ is a register allocated schedule of emitable instructions }

We wish to do the local register allocation in two steps. First we want to split live-ranges such that all live-ranges can be assigned a register. Second we will assign registers to all live-ranges. Both steps have to take the register requirements into account. This gives us the following specification.

{ $S.b$ is a schedule of emitable instructions }
Split live-ranges
{ $S.b$ is a register allocatable schedule of emitable instructions }
Assign registers
{ $S.b$ is a register allocated schedule of emitable instructions }

In this chapter we have formalized the local register allocation algorithm presented in [FL98].

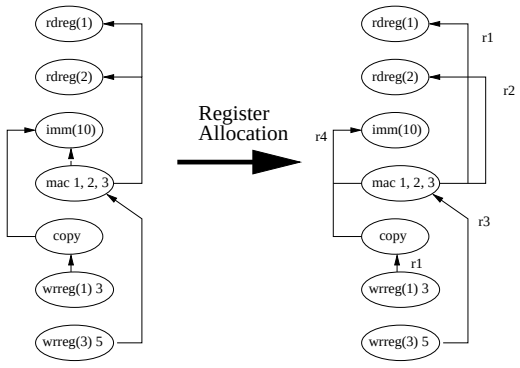


Figure 6.1: Local register allocation

6.1 Register assignment

After register assignment all live ranges must have a register assigned in such a way that overlapping live ranges do not use the same register. We identify a live range by the instruction that produces the value. A schedule is register allocated iff:

$$(\forall\, i : 0 \leq i < N \wedge \mathit{parents}.S.i \neq \varnothing : \mathit{reg}.i \in F.i)$$

Where $F.i = R \setminus U.i$ is the set of free registers at instruction i, and $U.i$ is the set of registers in use during the life time of instruction i, defined as follows.

$$\begin{array}{rcl} U.i & = & \{\mathit{reg}.j \mid 0 \leq j < i \wedge S.j \in \mathit{liveset}.S.i\} \\ & \cup & \{\mathit{reg}.j \mid i < j < N \wedge S.i \in \mathit{liveset}.S.j\} \end{array}$$

In figure 6.3, $R = \{r0, r1, r2\}$ which means $U.B = \{r0, r2\}$, since $r0$ is used by A, and $r2$ is used by C. The only remaining register is $r1$, and therefore $F.B = \{r1\}$. We will use the following invariants to establish the post condition.

P1: $(\forall\, i : 0 \leq i < n \wedge \mathit{parents}.\mathrm{S.i} \neq \varnothing : \mathit{reg}.i \in F.i)$
P2: $(\forall\, i : n \leq i < N : \mathit{reg}.i \notin R)$

The choice for the first invariant is obvious. Using invariant P2 we require that prior to register assignment no instruction has had a register assigned. We have added this invariant to simplify the computation of $F.n$:

$$\begin{array}{ll} F.n = & R \setminus \big(\{\mathit{reg}.j \mid 0 \leq j < n \wedge S.j \in \mathit{liveset}.S.n\} \cup \\ & \quad \{\mathit{reg}.j \mid n < j < N \wedge S.n \in \mathit{liveset}.S.j\} \big) \\ = & \quad \{\ \mathrm{P2},\ S.j \in \mathit{liveset}.S.n \ \Rightarrow\ 0 \leq j \leq n\ \} \\ & R \setminus \{\mathit{reg}.j \mid j \neq n \wedge S.j \in \mathit{liveset}.S.n\} \end{array}$$

■

To successfully assign a register to instruction n, the set $F.n$ must not be empty, and since *liveset* is reflexive, this means that $|\mathit{liveset}.S.n| \leq R$, for all n. This precondition is established by Split live-ranges.

We can write the function *liveset* defined in section 5.2 as a recursive function. For readability, and ease of notation we will represent instructions by their index into S.

$$\begin{array}{rcl} \mathit{result}.i & = & \{i \mid \mathit{parents}.i \neq \varnothing\} \\ \mathit{kills}.i & = & \{j \mid S.i \rightsquigarrow S.j \wedge (\forall\, k : S.k \rightsquigarrow S.j : k \leq i)\} \\ \mathit{liveset}'.0 & = & \mathit{result}.0 \\ \mathit{liveset}'.(i+1) & = & (\mathit{liveset}'.i \setminus \mathit{kills}.(i+1)) \cup \mathit{result}.(i+1) \end{array}$$

The function *result*.i returns a set containing the singleton i, only if instruction i produces a result. The function *kills*.i computes a set of instructions that are not live below i. Using these functions we can find a recursive definition for $F.n$. For ease of representation we will define $F.(-1) = \varnothing$. The recursive step is as follows:

$$F.(n+1) = \big(F.n \cup \mathit{reg}.\mathit{kills}.(n+1)\big) \setminus \mathit{reg}.\mathit{result}.(n+1)$$

where *reg* is lifted to sets. We can rewrite the equation $F.n$ as $F.(n-1) \cup \mathit{reg}.\mathit{kills}.n$, which gives us the program of figure 6.2.

6.2 Register requirements

An instruction can have the following requirements:

Register Assignment

$\{\ (\forall\, i : 0 \le i < N : |liveset.S.i| \le |R|) \ \wedge\ (\forall\, i : 0 \le i < N : reg.i \notin R)\ \}$
n, $F.(-1) := 0, \varnothing$;
do n $\neq$ N $\rightarrow$
  $F.n := F.(n-1) \cup reg.kills.n$;
  if $parents.$S.n $\neq \varnothing \rightarrow$
      let reg.n $\in F.n$
      $F.n := F.n \setminus \{$reg.$n\}$;
  ▯ $parents.$S.n $= \varnothing \rightarrow$
      skip
  fi;
  n:=n+1
od

Figure 6.2: simple register assignment algorithm

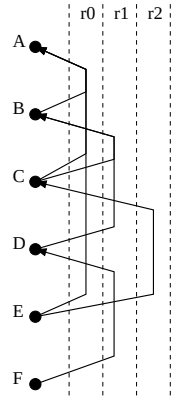


Figure 6.3: A register assignment

Explicit result register. For example, the *rdreg* pseudo instruction. An instruction with this requirement has a predetermined result register, the local register allocator should not assign a different register to this instruction.

This requirement cannot be met by the simple register assignment algorithm of figure 6.2 since the precondition $(\forall\, i : 0 \le i < N : reg.i \notin R)$ no longer holds. Since I think this problem is NP-complete without this precondition, it can most likely be reduced to the graph coloring problem, I have chosen to avoid this problem by limiting the local register allocation to a subset of R that does not contain the required registers. This solution has the disadvantage that there are fewer registers available for allocation.

Explicit argument register. For example, the *wrreg* pseudo instruction. An instruction with this requirement has a predetermined argument register. This requirement can be implemented using the previous requirement, we simply propagate the requirement upwards.

The result register must be the same as an argument register. Consider the two instructions $u \rightsquigarrow d$, where u requires that its result register must be the same as d's. This is only possible if u is the kill of d. This means that only one of d's uses can have this requirement. These conditions are established by the register conflict solver of chapter 4.

The result register must differ from an argument register. Consider the two instructions $u \rightsquigarrow d$, where u requires that its result register must be different from d's. If u is not the kill of d, this is not a problem since d and u are live a the same time and must therefore be assigned a different register anyway. If u is the kill of d we will have to make sure they are still assigned a different register. We do this by extending the live-range of d, such that their live-ranges still overlap. So $d \in liveset.u$ even if $kill.d = u$.

6.3 Splitting live-ranges

The goal of splitting live-ranges is to guarantee that $F.n \neq \varnothing$ holds during register assignment. Simply put, we have to reduce the register pressure where it is too high. This is usually done by recomputing values or by adding stores and loads. The difficult bit is of course deciding which live-range to split. We would like to split the live-ranges such that we introduce as few new instructions as possible. This problem is NP-hard [FL98].

There are a number of algorithms that give decent solutions for this problem for global register allocation, most notably GRA using graph coloring [Cha82, Bri92]. But these algorithms will not perform very well for local register allocation. The global register allocator spills when there are too many live values on entry or exit of a basic block. When a live-range is spilled it is no longer

live in a register, globally. However, it is live locally in each basic block. So we have reduced the pressure on global registers by increasing the pressure on local registers. During local register allocation we cannot do this.

There are also several heuristics known for local register allocation. The heuristic proposed in [Fre74] splits live ranges according to usage counts, the heuristics proposed in [HFG89, FL98] are based on the Furthest First algorithm, which splits the live range with the most distant reference. We will use the Conservative Furthest First heuristic proposed by Farach, but before we get to that let us take a closer look at the algorithm.

We compute a local register allocation by keeping track of a subset of *liveset* that live in registers, called L_{out}. Besides limiting the size of L_{out} to $|R|$ elements, we also require that the arguments of instruction i are live at $i-1$. This gives us the following post condition for LRA:

$$\begin{array}{rll} & (\forall\, i & : 0 \le i < N : \quad L_{out}.i \subseteq \mathit{liveset}'.i) \\ \wedge & (\forall\, i & : 0 \le i < N : \quad |L_{out}.i| \le |R|) \\ \wedge & (\forall\, i & : 1 \le i < N : \quad \mathit{children}.i \subseteq L_{out}.(i-1)) \end{array}$$

Which gives us the following invariants:

P0: $0 \le n \le N$
P1: $(\forall\, i : 0 \le i < n : L_{out}.i \subseteq \mathit{liveset}'.i)$
P2: $(\forall\, i : 0 \le i < n : |L_{out}.i| \le |R|)$
P3: $(\forall\, i : 1 \le i < n : \mathit{children}.S.i \subseteq L_{out}.(i-1))$

The following is a rough sketch of the actual algorithm:

Local register allocation
n, L_{out}.0 := 1, *result*.0;
do n $\neq N \rightarrow$
  Reduce Liveset;
  Load arguments;
  Write result;
  n:=n+1
od
$\{\ P_{0..3} \wedge n = N\ \}$

For notational convenience we introduce the following functions:

$$\begin{array}{rcl} \mathit{step}.L.i & = & (L.i \setminus \mathit{kills}.(i+1)) \cup \mathit{result}.(i+1) \\ L_{in}.(i+1) & = & L_{out}.i \cup \mathit{children}.S.(i+1) \end{array}$$

The function $L_{in}.i$ represents the instructions that must be live at the start of instruction i. The function $\mathit{step}.L.i$ computes the set of instructions that are live at the end of instruction i, using liveset L.

Substituting n:=n+1 in the invariants we get the post-condition for *Write result* :

$$P_{0..3} \ \wedge\ L_{out}.n \subseteq \mathit{liveset}'.n \ \wedge\ |L_{out}.n| \le |R| \ \wedge\ \mathit{children}.S.n \subseteq L_{out}.(n-1)$$

We establish the term $L_{out}.n \subseteq \mathit{liveset}'.n$ using the statement

Write result
L_{out}.n := $\mathit{step}.\mathsf{L}_{out}.$(n-1)

This has the effect of making the result of instruction n live. The precondition for *Write result* is as follows:

$$P_{0..3} \ \wedge\ |\mathit{step}.L_{out}.(n-1)| \le |R| \ \wedge\ \mathit{children}.S.n \subseteq L_{out}.(n-1)$$

We establish the term $\mathit{children}.S.n \subseteq L_{out}.(n-1)$ using the statement

Load arguments

L_{out}.(n-1) := L_{in}.n

This statement makes the children of instruction n live. When a child is not already live, it is restored using a load, or by duplicating its defining instruction. The precondition for *Load arguments* is as follows:

$$P_{0..3} \ \wedge \ |(L_{in}.n \setminus kills.n) \cup result.n| \leq |R| \ \wedge \ |L_{in}.n| \leq |R|$$

The term $|L_{in}.n| \leq |R|$ is introduced due to invariant P2.

We can rewrite this precondition as follows:

$$\begin{array}{ll} & |(L_{in}.n \setminus kills.n) \cup result.n| \leq |R| \ \wedge \ |L_{in}.n| \leq |R| \\ \equiv & \quad \{\ result.n \not\subseteq (L_{in}.n \setminus kills.n),\ kills.n \subseteq L_{in}.n\ \} \\ & |L_{in}.n| - |kills.n| + |result.n| \leq |R| \ \wedge \ |L_{in}.n| \leq |R| \\ \equiv & \quad \{\ \text{calculus}\ \} \\ & |L_{in}.n| \leq |R| + (0 \uparrow |kills.n| - |result.n|) \\ \equiv & \quad \{\ \text{definition } L_{in}\ \} \\ & |L_{out}.(n-1) \cup children.S.n| \leq |R| + (0 \uparrow |kills.n| - |result.n|) \\ \equiv & \quad \{\ \text{towards subset of } L_{out}\ \} \\ & |L_{out}.(n-1) \setminus children.S.n| \leq |R| - |children.S.n| + (0 \uparrow |kills.n| - |result.n|) \\ \equiv & \quad \{\ E = |children.S.n| - (0 \uparrow |kills.n| - |result.n|)\ \} \\ & |L_{out}.(n-1) \setminus children.S.n| \leq |R| - E \end{array}$$

■

Reduce Liveset should make sure there are enough free registers available for the arguments and result of instruction n. When there are not enough registers available we pick a live-range from L_{out} that is not an argument for instruction n, and split that live-range. This is repeated until there are enough registers available.

Reduce Liveset

{ Precondition: $E < |R|$ }
s:=|L_{out}.(n-1) \ $children$.S.n|;
do $|R| - E < s \rightarrow$
  let $c \in \mathsf{L}_{out}$.(n-1) \ $children$.S.n;
  L_{out}.(n-1):=L_{out}.(n-1) \ $\{c\}$;
  Split live-range c;
  s:=s-1
od
{ $|L_{out}.(n-1) \setminus children.S.n| \leq |R| - E$ }

The precondition holds if the maximum arity of an instruction is less than $|R|$, which is a reasonable demand.

The selection of c could be done using a heuristic like Conservative Furthest First, which will be discussed below.

The program presented here, *Local Register Allocation* and *Reduce Liveset*, abstracts from the method of splitting a live range. This is usually done by either duplicating an expression, or by storing and reloading the value. The choice between these two is usually made where the value is restored, *Load arguments* , or where the value is evicted, *Split live-range* . If we choose to split live-ranges in *Load arguments* , *Split live-range* is simply a **skip** .

Whether to choose to duplicate or to save and restore is determined by a number of things, usually duplicating an instruction has the preference, except when it does not decrease the register pressure, or when the instruction cannot be duplicated. The latter can happen when, for example,

the target for duplication is a load instruction, which is to be duplicated below a store instruction to the same address. We know the addresses of the two instructions may be the same when there is an after constraint from one to the other in the input. Loads and stores are not the only instructions to which these restrictions apply, after contraints must be taken into account for any instruction with a side-effect.

The postcondition of *Split live-ranges* implies $(\forall\, i : 0 \leq i < N : |L_{out}.i| \leq |R|)$. By introducing instructions for splitting live ranges we modify the livesets to match the L_{out}'s, in other words, this satisfies the precondition of *Register Assignment* .

6.3.1 Conservative Furthest First Heuristic

We implement the statement **let** $c \in \mathsf{L}_{out}.(\mathsf{n\text{-}1}) \setminus children.\mathsf{S.n}$ with a heuristic. The conservative furthest first heuristic selects a splitting candidate from *liveset* based upon the distance to the next reference, and whether or not the live range has been split already.

We select a candidate using the following statement.

$$c := i \mid i \in C \ \wedge\ (\forall\, j : j \in C : cost.j.n \leq cost.i.n)$$

where $C = L_{out} \setminus children.S.n$.

So the function *cost* is being maximized. This function, as proposed in [FL98], is defined as follows.

$$\begin{aligned} cost.i.n &= nextref.i.n + dirty.i \\ nextref.i.n &= j \mid S.j \rightsquigarrow S.i \wedge \neg(\exists\, k : S.k \rightsquigarrow S.i : n < k < j) \\ dirty.i &= \begin{cases} 0.5 & \text{if } i \text{ has been split before} \\ 0 & \text{if } i \text{ has not been split before} \end{cases} \end{aligned}$$

The heuristic as proposed in [FL98] does not consider duplicating instructions. However, the algorithm is not greatly affected by taking duplication into account. When it is possible we prefer duplication above spilling. We could tune the heuristic by changing the factor 0.5 in the function *dirty*, but the effects of that are marginal, as we have experimentally seen.

6.4 Machine Description Requirements

Local register allocation has several requirements with respect to the machine description. We need to know:

1. how many register the machine has,
2. what register requirements an instruction has, this is specified in the machine description language using a `require` statement, see chapter 8,
3. how to spill and restore a value. The spill and restore instructions are recognized by their input instruction name, `st32i` and `ld32i` respectively,
4. if an instruction has a side-effect, we cannot always duplicate an instruction with a side-effect.

Chapter 7

Dtree flattening

The output of a code generator is a sequence of instructions. A decision tree, at this stage, is a tree of sequences of instructions. Dtree flattening transforms the last into the first, as depicted in figure 7.1. This step is performed last in order to keep the other steps simple. Incorporating guarded instructions in the other steps may result in better performance of the output, but it also adds a lot of complexity we do not yet want.

We can flatten the dtree in two ways, using a branch instruction, or by guarding instructions. The choice depends on the size of each child, the number of times they are executed, and on the number of cycles it takes to perform a branch. The size of a basic block is easily computed, the number of times it is executed can be obtained using profiling. Figures 7.3 and 7.4 give an example flattening using branches and guarded instructions respectively, on the decision tree of figure 7.2. The flattening using branch instructions requires that we know how to negate the condition.

In the flattening of figure 7.4 we assume that conditions are explicit, ie. there are no instructions that modify a special *condition register*, instead there are instructions, like less, that perform the required comparisons and store the (boolean) result in a normal register. Due to the required branch in the leafs of the dtree, r4 and r5 will always be *true* below the instructions that negate them, and hence can be left out.

The use of a condition register is a little more tricky since we can use at most one register to hold the condition. We must order the basic blocks such that this is possible, scheduling nested ifs last, like was done in figure 7.4. When both then and else part have nested ifs we must introduce branches.

Profiling is a means to obtain information about the run-time behavior of a program. The compiler instruments the program with extra code to keep track of interesting information, usually an execution and cycle count for each function. This information can be read back into the compiler for use during various optimizations. With the profiling information provided by the front-end of

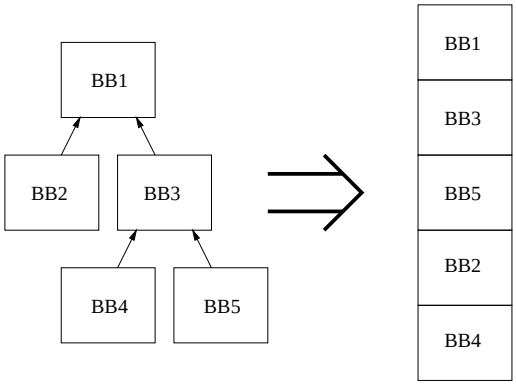


Figure 7.1: Flattening of a dtree

```
T0: { in: r1, r2 }
  r3 := load r2
  r4 := less r1, 5
  if r4 then
      r3 := add r3, 1
      store r3, r2
      goto T1
  else
      r5 := less r1, 10
      if r5 then
          r3 := add r3, 2
          store r3, r2
          goto T1
      else
          r3 := add r3, 3
          store r3, r2
          goto T1
      end
  end
```

Figure 7.2: Dtree to be flattened.

```
T0: { in: r1, r2 }
      r3 := load r2
      r4 := not-less r1, 5
      if r4 then goto L1
      r3 := add r3, 1
      store r3, r2
      goto T1
L1:   r5 := not-less r1, 10
      if r5 then goto L2
      r3 := add r3, 2
      store r3, r2
      goto T1
L2:   r3 := add r3, 3
      store r3, r2
      goto T1
```

Figure 7.3: Flattening using branch instructions.

```
T0: { in: r1, r2 }
  r3 := load r2
  r5 := false
  r4 := less r1, 5
  r3 := if r4 then   add r3, 1
        if r4 then   store r3, r2
        if r4 then   goto T1
  r4 := neg r4
  r5 := if r4 then   less r1, 10
  r3 := if r5 then   add r3, 2
        if r5 then   store r3, r2
        if r5 then   goto T1
  r5 := if r4 then   neg r5
  r3 := if r5 then   add r3, 3
        if r5 then   store r3, r2
        if r5 then   goto T1
```

Figure 7.4: Flattening using guarded instructions.

the compiler we can make a good decision about the ordering of the basic blocks. We want the basic blocks ordered in such a way that as little time as possible is spent executing the dtree. This means that when we flatten with branches, we want to branch to the least probable basic block, since taking a branch usually takes more time than not taking it. When we flatten with guarded instructions we want to schedule the most probable basic block first, since instructions with false guards still use precious cycles.

We cannot always order the children the way we want. For example, a basic block that is terminated with a function call should be put a the very end of the dtree when we cannot set the return address explicitly. We could also add an extra jump to the correct return address below the function call, but this is less efficient.

You may wonder why guarded instructions are interesting for RISC architectures, since it is mainly used to increase instruction level parallelism (ILP). On most RISC machines a branch instruction is an expensive operation. The instruction itself takes a number of cycles, and often after issuing the branch the pipeline is flushed. When we generate code for a hot loop it would be a waste if we had to add a branch for a few instructions we wish to execute conditionally. In that case guarded instructions can greatly improve performance.

We have the following specification for dtree flattening.

{ $(\forall\, b : b \in B : S.b$ is a register allocated schedule of emitable instructions) }
Dtree flattening
{ S is the "merge" of all b's, such that S is a register allocated schedule of emitable instructions }

We do not intend to reschedule and reallocate the resulting basic block, if we wanted to do that we could have flattened the dtree *before* the other stages. The "merge", in this case, is simply a concatenation of all basic blocks in the dtree. To make a valid concatenation of basic blocks, one that adheres to the ordering specified by the control flow graph, we must schedule them. We can simplify the specification to the following:

{ T is a tree of basic blocks }
Dtree flattening
{ S is a schedule of basic blocks }

We want to schedule the basic blocks such, that the average time spent on executing a dtree is minimal. The total time spent in a basic block is $t = execution\ count * time\ spent\ in\ a\ single\ execution$, assuming there are no instructions whose timing depend on their arguments. Using profiling we can obtain execution counts for each basic block, we want to schedule the basic block with the highest t first. For the scheduling of the tree we can use the list scheduling algorithm.

Chapter 8

Machine description language

The code generator is made retargetable by use of a machine description. The machine description consists of two files. One file contains the rewrite rules used for instruction selection, as explained in chapter 3. The rewrite rules are used to rewrite the data flow graph into a data flow graph of emitable instructions. The other file contains information that tells the back-end how to write an assembly file. This file contains information that tells whether an instruction is emitable or not. It describes the syntax of all emitable instructions, the syntax of the rest of the assembly file, and some compiler conventions. This chapter explains the language used to describe this part of the machine description.

¿From the previous chapters we have the following requirements:

1. We need to know which instruction is the *copy* instruction to solve register conflicts.
2. We need to know what register requirements instructions have during local register allocation. The register requirements also specify which instructions are 2-address. This is used to solve register conflicts.
3. We need to know how many registers the machine has, for local register allocation.
4. We need to know how so store and load values to and from memory to generate spill code.
5. We need to know which instructions have a side-effect, during register allocation.

The *copy*, *load* and *store* instructions are found by their name in the input instruction set, `ident`, `ld32i`, and `st32i` respectively. Requirement 2 is fulfilled by a `require` statement, as explained in section 8.4. Requirement 5 is fulfilled by the instruction classification, also explained in section 8.4. Requirement 4 is fulfilled by an assignment to the variable `NumRegisters` in the machine description, as explained in the next section.

All examples in this chapter were taken from an ARM7 machine description.

8.1 Machine Description Grammar

The machine description grammar is as follows.

MD ::= "`TYPES`" Types
"`CONSTANTS`" { ConstantDef }
"`OPERATIONS`" { InsDeclaration }
"`OUTPUT`" "`FUNCTIONS`" { Function }

The `TYPES` section is used to define the types used in the machine description, pre-defined types are `string`, `int`, `expr`, and `operator`.

The `CONSTANTS` section is used to define constants, eg. the number of register. The production rule for a constant definition is as follows.

ConstantDef ::= Ident "=" (Int | String)

The `OPERATIONS` sections hold the declaration of all emitable instructions. The grammar for an instruction declaration is explained in section 8.4.

The `OUTPUT FUNCTIONS` section describes functions that emit portions of an assembly file, for example how to start a data section. This is explained in section 8.3. First we take a look at what an assembly file usually looks like.

8.2 General assembly file layout

The layout of an assembly file is quite similar for most assembly languages. There is usually a header of some sort, followed by a number of text and data sections, followed by a footer. The text section contains the generated code. Data sections come in several forms, common data sections are for read-only data, writable data, and uninitialized data.

We have generalized the assembly file language to the following, the *italic* non-terminals are to be filled in by the machine description.

AsmFile ::= *Header* ExternalLabels { SectionName Content } *Footer*
SectionName ::= *Text* | *Readonlydata* | *Writabledata* | *UninitialisedData*
Content ::= { *Instruction* | *Label* | *Data* }
ExternalLabels ::= { *ImportLabel* | *ExportLabel* }

A remaining issue is debugging information. There are many ways to keep track of debugging information, the more popular being Stabs and DWARF [Int92], and the matter becomes even more complicated when the back-end has to generate debugging information as well. We have chosen to ignore this issue for the moment.

8.3 Output Function Grammar

The machine description file contains a number of functions to emit the elements of the assembly file. For each of these functions a C function is generated to be compiled in with the back-end. The grammar for such a function is as follows.

Function ::= "`emit`" "(" Element [Arguments] ")" FunctionBody
FunctionBody ::= "`{`" { Statement ";" } "`}`"
Statement ::= "`emit`" Emission

The production rule for Statement given here is incomplete, it will be extended later on in this chapter. An example of a very simple function is the emit function for the *Footer*.

```
emit (FOOTER)
{ emit "\tEND\n"; }
```

This will generate a C function called "emit_FOOTER" without any arguments. Some emit functions do require arguments, for example, the emit function for a label declaration is as follows.

```
emit (LABELDECL string s)
{ emit "|" s "|"; }
```

This will generate a C function called "emit_LABELDECL" with a single argument. The string `s` will be emitted using an "emit_string" C function, which is implemented in the back-end.

The production rule for Emission is as follows, observe that it is also possible to conditionally emit a string.

Emission ::= { Expression | IfExpression }
IfExpression ::= "`IF`" Expression "`THEN`" Emission
{ "`ELSIF`" Expression "`THEN`" Emission }
"`ELSE`" Emission

The following function emits the header for a section, depending on the kind of section, another string is emitted.

```
emit (SECTION string s)
{ emit
    IF    (s== "text")       THEN "\tAREA |C$$text|, CODE"
    ELSIF (s== "data")       THEN "\tAREA |C$$data|, DATA"
    ELSIF (s== "bss")        THEN "\tAREA |C$$bss|, DATA, NOINIT"
    ELSIF (s== "data1")      THEN "\tAREA |C$$cdata|, READONLY, DATA"
    ELSE                          ""
    FI ;
}
```

The section name passed to this function is the name given to the section in the input file for the code generator.

The `assert` statement can be used to state conditions which must hold for the arguments. If these conditions do not hold upon executing the function the program will abort with an error message.

Statement ::= "`assert`" Expression

For example, the following function emits a register name, since there are a limited number of registers we want to check whether the register to be emitted is valid.

```
emit (reg r)
{
  assert 0<=r && r<=15;
  emit IF 0<=r && r<13 THEN ``r'' r
       ELSIF r==13 THEN ``sp''
       ELSIF r==14 THEN ``lr''
       ELSIF r==15 THEN ``pc'';
}
```

The emit functions for data declaration are quite similar to the emit functions in the examples above. The following emit function declares a 32 bit word.

```
emit (WORD imm i)
{ emit "\tDCD " i; }
```

The argument `i` is an expression of integers and labels, due to a type definition in the `TYPES` section, it is emitted using the "emit_expression" C function, which prints it as a normal expression. The labels in the expression are emitted using the following function.

```
emit (label s)
{ emit "|" s "|"; }
```

8.4 Instruction description

For instructions we do not only wish to know how to write them to an assembly file, there are also a number of properties we need to know.

An instruction is a function, one that is evaluated by a processor. In order to generate an instruction we need to know its type, ie. the number and type of its arguments and results. This information is described by the InsDeclaration.

InsDeclaration ::= Ident "(" Arguments ")" [":" Argument]
Arguments ::= Argument//","
Argument ::= Type [Ident]

For example, the following two instruction description describes the integer add instruction.

```
iadd(reg rm, reg rn) : reg rd
iaddi(reg rm, imm i) : reg rd
```

The optional Argument after the colon, ":", is the result, `rd` in this case. We allow at most one result, and only of type `reg`.

A complete instruction description is an InsDeclaration followed by a FunctionBody, the `emit` statement in the FunctionBody can use the named arguments in the declaration.

InsDescription ::= InsDeclaration FunctionBody

The complete instruction description of the integer add is as follows.

```
iadd(reg rm, reg rn) : reg rd
{ emit "ADD" CC() " " rd ", " rm ", " rn; }
```

The `CC()` function emits the guard of the instruction. Registers are emitted using the `emit (reg r)` function.

As was explained in chapter 4, some instructions have register requirements, eg. an argument register must or must not be the same as the result register, see also the second requirement in the introduction. You can specify these requirements using a `require` statement.

Statement ::= "`require`" Requirement//","
Requirement ::= Ident ("==" | "!=") (Ident | Expression)

For example, the "multiply" requires the result register to be different from the first argument register.

```
imul(reg rm, reg rn) : reg rd
{ require rm!=rd;
  emit "MUL" CC() " " rd ", " rm ", " rn;
}
```

Unlike the `assert` statement, the `require` statement is not a run-time check, the requirements are established by properly allocating registers and introducing copies.

To avoid an explosion in the number of instruction, the generalized addressing modes of the ARM7, see chapter 3, are implemented by allowing *operators* as argument to an instruction.

```
addxi(imm i, reg rm, reg rn, opr op) : reg rd
{ emit "ADD" CC() " " rd ", " rm ", " rn ", " op " #" i; }
```

When the operator `op` needs to be printed the mnemonic of that operator is retrieved. The mnemonic is specified in the FunctionBody of the instruction associated with that operator using an assignment.

Statement ::= Ident "=" Expression

For example the `asr` instruction can be an operator to the `addxi` instruction, it also has an `emit` statement since it can be an instruction in itself as well.

```
asr(reg rm, reg rn) : reg rd
{ mne = "ASR";
  emit "MOV" CC() " " rd ", " rm ", " mne " " rn;
}
```

It should be noted that, of the Statements in the function body, only the `emit` and `assert` statements are executed. The requirement and all assignments are attributes of the instruction. So upon emitting the `addxi` instruction the `mne` attribute of the `opr` argument is retrieved and emitted.

We could use this construction for the condition codes as well. We have chosen, however, to treat the condition codes as a special case, since this will make both machine description files easier to read and create. The `CC()` function uses the `mne` attributes of the comparison instructions.

It is also useful to have a crude classification of instructions, for example instructions that use memory, control flow instructions, instruction with a side effect. These classes are used in various parts throughout the back-end. We distinguish the following classifications:

IMMLOAD	Instructions that load an immediate value.
MEMORY	Instructions that access memory.
LINK	Instructions that set the function call return address.
FLOW	Control flow instructions.
SIDE_EFFECT	Instructions with a side effect.
PSEUDO	Pseudo instructions.

An instruction should be classified as IMMLOAD when it loads the immediate argument into a register, this is especially useful during peephole optimization. The SIDE_EFFECT and MEMORY classes are used during local register allocation, to see if an instruction can be duplicated. The LINK and FLOW classes are mainly used to add after constraints that are implicit in the input. An instruction classified as PSEUDO can occur after instruction selection and is allowed to have no emit function.

An instruction can be in more than one class. Since many instructions are in the same class, it is not specified in the FunctionBody. Instead the classification is specified once, using a classification label, and is assigned to all following instructions until the next classification label.

For example, store instructions access memory, and as a "side" effect modify the memory. The following example classifies the store instructions `st32` and `st32i` as `MEMORY` and `SIDE_EFFECT`.

```
MEMORY SIDE_EFFECT:

st32(reg rm, reg rn)
{ emit "STR" CC() " " rn ", [" rm "]"; }

st32i(imm i, reg rm, reg rn)
{ emit "STR" CC() " " rn ", [" rm ", #" i "]"; }
```

I have made a machine description also for an imaginary processor, in order to evaluate the fitness for retargetability. The processor I implemented does not use condition codes, but instructions to evaluate specific conditions the result of which is put into a register. The rest of the machine description is a lot like that of the ARM7. I chose to change the implementation of conditional execution, since that is one of the things that varies greatly among RISC processors, and is one drastically different from the ARM implementation.

There were little problems in creating this machine description. The greatest problem is that using this machine description all instructions get an extra argument for the condition, an argument the input instruction set does not have. This is not a big problem, only a lot of type work in the instruction selection, you get a lot of rules like "`add(a,b) -> ADD(ALWAYS(),a,b)`", where `ALWAYS` is the condition. There is also a small part of C-code that had to be changed, the part that emits the condition. In order to eliminate this mishap, the machine description language may have to be extended, this is part of future work.

Appendix A gives the complete ARM7 machine description file.

Chapter 9

Benchmarks

In order to evaluate the effectiveness of the scheduling heuristics, presented in chapter 5, we measure the number of stores (S), loads (L), and duplicates (D) added during register allocation. Figure 9.1 presents the results on the SPECint95 benchmark set. This benchmark set is a collection of "real life" applications that stress the processor:

compress	a Lempel-Ziv data compression utility
gcc	The GNU C compiler
li	a Lisp interpreter
go	a Go game with artificial opponent
ijpeg	JPEG compression and decompression
m88ksim	a Motorola 88k processor simulator
vortex	an object oriented database management system

The three heuristics presented in the table are described in section 5.3. The Warren heuristic was presented in [War90], and the Haifa heuristic in [Tie89].

program	Warren				Haifa				our heuristic			
	S	L	D	total	S	L	D	total	S	L	D	total
compress	1	1	1	3	1	2	1	4	1	1	0	2
gcc	88	91	595	774	73	76	567	716	8	9	75	92
li	0	0	2	2	0	0	2	2	0	0	0	0
go	72	101	203	376	57	97	160	314	30	42	10	82
ijpeg	334	385	527	1246	243	282	387	912	113	136	80	329
m88ksim	73	88	225	386	56	66	196	318	6	6	10	22
perl	2	2	38	42	2	3	33	38	2	2	3	7
vortex	84	99	751	934	87	103	682	872	25	26	61	112

Figure 9.1: Number of instructions added due to spilling. The S column gives the number of stores, L the number of loads, and D the number of duplicates.

The improvement caused by using our heuristic is rather dramatic. This can partly be explained with the aim of the heuristic. The Warren and Haifa heuristic are not only aimed at reducing register pressure, but also at preventing interlocking. Scheduling to prevent interlocking tends to increase register pressure. I have ignored interlocking for the moment since the current target, and ARM7 processor, does not suffer from interlocking. I am aware that this makes the benchmark slightly biased, but the Haifa heuristic is used in the GNU C compiler for the ARM7 as well.

The following table compares the speed of the SPECint95 programs when compiled with the ARM compiler and with our compiler. Unfortunately due to library issues *perl* did not compile and due to compiler issued *ijpeg*, and *gcc* did not compile for the ARM. Nevertheless, the benchmarks that could compile paint a disappointing picture.

The slow down is mainly caused by the Global Register Allocator introducing spill instructions

program	armcc	our cc	%
compress	17.8M	20M	89%
li	337.1M	456M	73.9%
go	483.7M	630M	76.7%
m88ksim	875.6M	1087.7M	80.5%
vortex	2136.0M	3524.3M	60.6%

Figure 9.2: Speed of compiled programs. Column 2 and 3 give the absolute cycle count, column 4 gives the relative speed up of using our compiler.

into critical parts of the code. The ARM compiler is more effective in avoiding adding spill code in these critical section. The ARM compiler is also more effective in its use of instructions with side effects. This can be improved in our compiler by the add support for instructions with more than one result.

Chapter 10

Conclusions

The aim of this exercise was to produce an easy to retarget, industrial strength code generator for the ARM7 RISC processor. The code generator I have made can be retargeted using two files, one for instruction selection, and another for assembly file emission and remaining configuration. Although the code generator I have created can currently not compete with the ARM compiler, it has shown to be more stable.

During the creation of the code generator, the following has been achieved.

- A scheduling heuristic for reducing register pressure for non-interlocking RISC processors has been created. This heuristic shows a dramatic improvement in register pressure over present heuristics that do take interlocking into account.
- A method has been developed for solving register conflicts in the presence of 2-address instructions.
- A machine description language for assembly file emission has been developed.

The road to completion of the code generator and this report has been an arduous one. Many parts of the code generator have been written several times over. Although this method gives good insight into, or at least appreciation of, the problem, it is rather lengthy, and not very constructive. A systematic approach is more effective and even unavoidable for the more complex problems, like local register allocation, and the solving of register conflicts. Had I started using a systematic approach, the road to completion might have been shorter, and easier to travel.

I have also found that, finding the wheel is sometimes as hard or harder than reinventing it. It took me a while to find the paper of Farach [FL98], discussing local register allocation. By that time I had already made one myself, which gave similar results.

The code generator has also reaffirmed my belief that good design of the data structures is of the utmost importance. Not only the fact that you use a DAG to represent data flow, but also all the pesky little details not worth mentioning that come to haunt you later on if you don't pay them the proper attention. Alas, it is hard to think of everything beforehand.

I am particularly pleased with the scheduling heuristic I have found, for the simple reason that it seems to be a very good one. I am also pleased with the register conflict solver, this was not a problem common in code generators, and forced me to really think. All in all, I have learned a lot during the creation of the code generator, and this report.

10.1 Future work

The project in which this code generator has been made has ended. However, future directions could include the following.

- Perform dtree flattening first instead of last. This will increase the number of possible schedules for a basic block with guarded instructions, and hence may reduce register pressure.

- Improve the performance of the compiler.
- Add a scheduling heuristic for interlocking processors.
- Adapt the generator to be retargetable for other processor kinds, eg. DSPs, CISC machines, and perhaps VLIWs.
- Allow the use of pre-assigned register in local register allocation.
- Improve the machine description further, mainly to eliminate the last bits of machine dependant source code.

10.2 Acknowledgments

I'd like to thank the following people, for their help during the creation of the back-end and the report, and for keeping me going on this report.

Lex Augusteijn, Cees Hemerik, Rik van de Wiel, Frits Schalij, Tom de Vries, Paul Hoogendijk, Arjan Bink, Hervé Soulard, the other people of the CT cluster, and my parents.

Appendix A

Machine Description file

A.1 Assembly emission file

The following is the assembly emission part of the ARM7 machine description.

```
#define CC()  emit_condition()

TYPES

reg = int;
label = string;
imm = expr;
opr = operator;

CONSTANTS

NumRegisters = 16;
RegisterPC   = 15;
RegisterLR   = 14;
RegisterSP   = 13;
LiteralPoolMaxOffset = 514;
LiteralPoolMinOffset = -510;
CalleeSavedRegs = "0FF0";

OPERATIONS

ident(reg rm) : reg rd
{  emit "MOV" CC() " " rd ", " rm; }

addsp(imm i, reg rm) : reg rd
{ require rm==rd;
  emit "ADD" CC() " " rd ", " rm ", #" i;
}
subsp(imm i, reg rm) : reg rd
{ require rm==rd;
  emit "SUB" CC() "S " rd ", " rm ", #" i;
}

iadd(reg rm, reg rn) : reg rd
{ emit "ADD" CC() " " rd ", " rm ", " rn; }
iaddi(imm i, reg rm) : reg rd
```

```
{ emit "ADD" CC() " " rd ", " rm ", #" i; }
isub(reg rm, reg rn) : reg rd
{ emit "SUB" CC() " " rd ", " rm ", " rn; }
isubi(imm i, reg rm) : reg rd
{ emit "SUB" CC() " " rd ", " rm ", #" i; }
imul(reg rm, reg rn) : reg rd
{ require rm!=rd;
  emit "MUL" CC() " " rd ", " rm ", " rn;
}
umul(reg rm, reg rn) : reg rd
{ require rm!=rd;
  emit "MUL" CC() " " rd ", " rm ", " rn;
}

asr(reg rm, reg rn) : reg rd
{ mne = "ASR";
  emit "MOV" CC() " " rd ", " rm ", " mne " " rn;
}
asri(imm i, reg rm) : reg rd
{ mne = "ASR";
  emit "MOV" CC() " " rd ", " rm ", " mne " #" i;
}
lsr(reg rm, reg rn) : reg rd
{ mne = "LSR";
  emit "MOV" CC() " " rd ", " rm ", " mne " " rn;
}
lsri(imm i, reg rm) : reg rd
{ mne = "LSR";
  emit "MOV" CC() " " rd ", " rm ", " mne " #" i;
}
lsl(reg rm, reg rn) : reg rd
{ mne = "LSL";
  emit "MOV" CC() " " rd ", " rm ", " mne " " rn;
}
lsli(imm i, reg rm) : reg rd
{ mne = "LSL";
  emit "MOV" CC() " " rd ", " rm ", " mne " #" i;
}

irsb(reg rm, reg rn) : reg rd
{ emit "RSB" CC() " " rd ", " rm ", " rn; }
irsbi(imm i, reg rm) : reg rd
{ emit "RSB" CC() " " rd ", " rm ", #" i; }


and(reg rm, reg rn) : reg rd
{ emit "AND" CC() " " rd ", " rm ", " rn; }
or(reg rm, reg rn) : reg rd
{ emit "ORR" CC() " " rd ", " rm ", " rn; }
xor(reg rm, reg rn) : reg rd
{ emit "EOR" CC() " " rd ", " rm ", " rn; }
not(reg rm) : reg rd
{ emit "MVN" CC() " " rd ", " rm; }
```

```
andn(reg rm, reg rn) : reg rd
{ emit "BIC" CC() " " rd ", " rm ", " rn; }

addiA(imm i, imm j, reg rm) : reg rd
{ emit "ADD" CC() " " rd ", " rm ", #" i ", " j; }
subiA(imm i, imm j, reg rm) : reg rd
{ emit "SUB" CC() " " rd ", " rm ", #" i ", " j; }
rsbiA(imm i, imm j, reg rm) : reg rd
{ emit "RSB" CC() " " rd ", " rm ", #" i ", " j; }
andiA(imm i, imm j, reg rm) : reg rd
{ emit "AND" CC() " " rd ", " rm ", #" i ", " j; }
xoriA(imm i, imm j, reg rm) : reg rd
{ emit "EOR" CC() " " rd ", " rm ", #" i ", " j; }
oriA(imm i, imm j, reg rm) : reg rd
{ emit "ORR" CC() " " rd ", " rm ", #" i ", " j; }
andniA(imm i, imm j, reg rm) : reg rd
{ emit "BIC" CC() " " rd ", " rm ", #" i ", " j; }

addx(reg i, reg rm, reg rn, opr op) : reg rd
{ emit "ADD" CC() " " rd ", " rm ", " rn ", " op " " i; }
subx(reg i, reg rm, reg rn, opr op) : reg rd
{ emit "SUB" CC() " " rd ", " rm ", " rn ", " op " " i; }
rsbx(reg i, reg rm, reg rn, opr op) : reg rd
{ emit "RSB" CC() " " rd ", " rm ", " rn ", " op " " i; }
andx(reg i, reg rm, reg rn, opr op) : reg rd
{ emit "AND" CC() " " rd ", " rm ", " rn ", " op " " i; }
xorx(reg i, reg rm, reg rn, opr op) : reg rd
{ emit "EOR" CC() " " rd ", " rm ", " rn ", " op " " i; }
orx(reg i, reg rm, reg rn, opr op) : reg rd
{ emit "ORR" CC() " " rd ", " rm ", " rn ", " op " " i; }
andnx(reg i, reg rm, reg rn, opr op) : reg rd
{ emit "BIC" CC() " " rd ", " rm ", " rn ", " op " " i; }

addxi(imm i, reg rm, reg rn, opr op) : reg rd
{ emit "ADD" CC() " " rd ", " rm ", " rn ", " op " #" i; }
subxi(imm i, reg rm, reg rn, opr op) : reg rd
{ emit "SUB" CC() " " rd ", " rm ", " rn ", " op " #" i; }
rsbxi(imm i, reg rm, reg rn, opr op) : reg rd
{ emit "RSB" CC() " " rd ", " rm ", " rn ", " op " #" i; }
andxi(imm i, reg rm, reg rn, opr op) : reg rd
{ emit "AND" CC() " " rd ", " rm ", " rn ", " op " #" i; }
xorxi(imm i, reg rm, reg rn, opr op) : reg rd
{ emit "EOR" CC() " " rd ", " rm ", " rn ", " op " #" i; }
orxi(imm i, reg rm, reg rn, opr op) : reg rd
{ emit "ORR" CC() " " rd ", " rm ", " rn ", " op " #" i; }
andnxi(imm i, reg rm, reg rn, opr op) : reg rd
{ emit "BIC" CC() " " rd ", " rm ", " rn ", " op " #" i; }

mla(reg ra, reg rb, reg rc) : reg rd
{
  require rd!=ra;
  emit "MLA" CC() " " rd ", " ra ", " rb ", " rc;
}
sh_imm(imm a, imm b) : reg ra
```

```
{ emit "MOV" CC() " " ra ", #" a ", " b; }
neg_sh_imm(imm a, imm b) : reg ra
{ emit "MVN" CC() " " ra ", #" a ", " b; }

COMPARISON SIDE_EFFECT:
isubs(reg ra, reg rb) : reg rd
{ emit "SUB" CC() "S " rd ", " ra ", " rb ; }
isubsi(imm i, reg ra) : reg rd
{ emit "SUB" CC() "S " rd ", " ra ", #" i ; }
icmp(reg ra, reg rb)
{ emit "CMP" CC() " " ra ", " rb ; }
icmpi(imm i, reg ra)
{ emit "CMP" CC() " " ra ", #" i ; }

iles(reg ra, reg rb)  { mne = "LT"; emit "CMP " ra ", " rb; }
ileq(reg ra, reg rb)  { mne = "LE"; emit "CMP " ra ", " rb; }
ieql(reg ra, reg rb)  { mne = "EQ"; emit "CMP " ra ", " rb; }
igeq(reg ra, reg rb)  { mne = "GE"; emit "CMP " ra ", " rb; }
igtr(reg ra, reg rb)  { mne = "GT"; emit "CMP " ra ", " rb; }
ineq(reg ra, reg rb)  { mne = "NE"; emit "CMP " ra ", " rb; }
ilesi(imm i, reg rb)  { mne = "LT"; emit "CMP " rb ", #" i; }
ileqi(imm i, reg rb)  { mne = "LE"; emit "CMP " rb ", #" i; }
ieqli(imm i, reg rb)  { mne = "EQ"; emit "CMP " rb ", #" i; }
igeqi(imm i, reg rb)  { mne = "GE"; emit "CMP " rb ", #" i; }
igtri(imm i, reg rb)  { mne = "GT"; emit "CMP " rb ", #" i; }
ineqi(imm i, reg rb)  { mne = "NE"; emit "CMP " rb ", #" i; }
ules(reg ra, reg rb)  { mne = "LO"; emit "CMP " ra ", " rb; }
uleq(reg ra, reg rb)  { mne = "LS"; emit "CMP " ra ", " rb; }
ueql(reg ra, reg rb)  { mne = "EQ"; emit "CMP " ra ", " rb; }
ugeq(reg ra, reg rb)  { mne = "HS"; emit "CMP " ra ", " rb; }
ugtr(reg ra, reg rb)  { mne = "HI"; emit "CMP " ra ", " rb; }
uneq(reg ra, reg rb)  { mne = "NE"; emit "CMP " ra ", " rb; }
ulesi(imm i, reg rb)  { mne = "LO"; emit "CMP " rb ", #" i; }
uleqi(imm i, reg rb)  { mne = "LS"; emit "CMP " rb ", #" i; }
ueqli(imm i, reg rb)  { mne = "EQ"; emit "CMP " rb ", #" i; }
ugeqi(imm i, reg rb)  { mne = "HS"; emit "CMP " rb ", #" i; }
ugtri(imm i, reg rb)  { mne = "HI"; emit "CMP " rb ", #" i; }
uneqi(imm i, reg rb)  { mne = "NE"; emit "CMP " rb ", #" i; }

IMMLOAD:

imm(imm i) : reg rd
{ emit "LDR" CC() " " rd ", [pc, #((" i "-2)*4)]"; }

MEMORY:

push(regmask mask, reg rm) : reg rd
{ require rm==rd, rd==RegisterSP;
  emit "STM" CC() "DB " rm "!, " mask;
}
pop(regmask mask, reg rm) : reg rd
{ require rm==rd, rd==RegisterSP;
  emit "LDM" CC() "IA " rm "!, " mask;
}
```

```
ld32(reg rm) : reg rd
{ emit "LDR" CC() " " rd ", [" rm "]"; }
ld32i(imm i, reg rm) : reg rd
{ emit "LDR" CC() " " rd ", [" rm ", #" i "]"; }
ld32r(reg rm, reg rn) : reg rd
{ emit "LDR" CC() " " rd ", [" rm ", " rn "]"; }
ld32x(reg rm, reg rn) : reg rd
{ emit "LDR" CC() " " rd ", [" rm ", " rn ", LSL #2 ]"; }

ild16(reg rm) : reg rd
{ emit "LDR" CC() "SH " rd ", [" rm "]"; }
ild16r(reg rm, reg rn) : reg rd
{ emit "LDR" CC() "SH " rd ", [" rm ", " rn "]"; }
ild16i(imm i, reg rm) : reg rd
{ emit "LDR" CC() "SH " rd ", [" rm ", #" i "]"; }

uld16(reg rm) : reg rd
{ emit "LDR" CC() "H " rd ", [" rm "]"; }
uld16r(reg rm, reg rn) : reg rd
{ emit "LDR" CC() "H " rd ", [" rm ", " rn "]"; }
uld16i(imm i, reg rm) : reg rd
{ emit "LDR" CC() "H " rd ", [" rm ", #" i "]"; }

ild8(reg rm) : reg rd
{ emit "LDR" CC() "SB " rd ", [" rm "]"; }
ild8r(reg rm, reg rn) : reg rd
{ emit "LDR" CC() "SB " rd ", [" rm ", " rn "]"; }
ild8i(imm i, reg rm) : reg rd
{ emit "LDR" CC() "SB " rd ", [" rm ", #" i "]"; }

uld8(reg rm) : reg rd
{ emit "LDR" CC() "B " rd ", [" rm "]"; }
uld8r(reg rm, reg rn) : reg rd
{ emit "LDR" CC() "B " rd ", [" rm ", " rn "]"; }
uld8i(imm i, reg rm) : reg rd
{ emit "LDR" CC() "B " rd ", [" rm ", #" i "]"; }

ld32_xi(reg rm, reg rn, opr op, imm i) : reg rd
{ emit "LDR" CC() " "  rd ", [" rm ", " rn ", " op " #" i " ]"; }
uld8_xi(reg rm, reg rn, opr op, imm i) : reg rd
{ emit "LDR" CC() "B " rd ", [" rm ", " rn ", " op " #" i " ]"; }

MEMORY SIDE_EFFECT:

st32(reg rm, reg rn)
{ emit "STR" CC() " " rn ", [" rm "]"; }
st32i(imm i, reg rm, reg rn)
{ emit "STR" CC() " " rn ", [" rm ", #" i"]"; }
st32r(reg ra, reg rb, reg rc)
{ emit "STR" CC() " " ra ", [" rb ", " rc "]"; }
st32x(reg ra, reg rb, reg rc)
{ emit "STR" CC() " " ra ", [ " rb ", " rc ", LSL #2 ]"; }
```

```
st16(reg rm, reg rn)
{ emit "STR" CC() "H " rn ", [" rm "]"; }
st16i(imm i, reg rm, reg rn)
{ emit "STR" CC() "H " rn ", [" rm ", #" i"]"; }
st16r(reg ra, reg rb, reg rc)
{ emit "STR" CC() "H " ra ", [" rb ", " rc "]"; }

st8(reg rm, reg rn)
{ emit "STR" CC() "B " rn ", [" rm "]"; }
st8i(imm i, reg rm, reg rn)
{ emit "STR" CC() "B " rn ", [" rm ", #" i"]"; }
st8r(reg ra, reg rb, reg rc)
{ emit "STR" CC() "B " ra ", [" rb ", " rc "]"; }

st32_xi(reg rv, reg ra, reg rb, opr op, imm i)
{ emit "STR" CC() " " rv ", [ " ra ", " rb ", " op " #" i " ]"; }
st8_xi(reg rv, reg ra, reg rb, opr op, imm i)
{ emit "STR" CC() "B " rv ", [ " ra ", " rb ", " op " #" i " ]"; }

LINK SIDE_EFFECT:

link(imm i)
{ emit "MOV" CC() " lr, pc"; }

FLOW SIDE_EFFECT:

jmp(reg rm)
{ emit "MOV" CC() " pc, " rm; }
ijmp(imm i)
{ emit "B" CC() " "  i; }
ajmp(imm i)
{ emit "B" CC() " " i; }
ret()
{ emit "MOV" CC() " pc, lr"; }

FLOW CONDITIONAL SIDE_EFFECT:

call(imm i)
{ emit "BL" CC() " " i; }
jz(imm i)
{ emit "BEQ " i; }
jnz(imm i)
{ emit "BNE " i; }
jgt(imm i)
{ emit "BGT " i; }
jgtu(imm i)
{ emit "BHI " i; }
jle(imm i)
{ emit "BLE " i; }
jleu(imm i)
{ emit "BLS " i; }
jlt(imm i)
{ emit "BLT " i; }
jltu(imm i)
```

```
{ emit "BLO " i; }
jge(imm i)
{ emit "BGE " i; }
jgeu(imm i)
{ emit "BHS " i; }

jc(reg ra, imm i)

PSEUDO:

rdreg(imm i) : reg rd

SIDE_EFFECT PSEUDO:

wrreg(imm i, reg rm)

OUTPUT FUNCTIONS

emit (reg i)
{
  assert 0<=i, i<=16;
  emit IF    (i==RegisterPC) THEN "pc"
       ELSIF (i==RegisterLR) THEN "lr"
       ELSIF (i==RegisterSP) THEN "sp"
       ELSE                       "r" i
       FI ;
}

emit (imm e)
{ emit e; }

emit (label s)
{ emit s; }

emit (HEADER)
{ emit ""; }

emit (FOOTER)
{ emit ""; }

emit (LABELDECL string s)
{ emit s ":"; }

emit (EXPORT string s)
{ emit "\t.global " s ; }

emit (IMPORT string s)
{ emit ""; }

emit (SECTION string s)
{ emit
  IF    (s== "text")       THEN "\t.text"
  ELSIF (s== "jumptable")  THEN "\tAREA |C$$jumptable|, CODE"
  ELSIF (s== "data")       THEN "\t.section .data"
```

```
  ELSIF (s== "bss")        THEN "\tAREA |C$$bss|, DATA, NOINIT"
  ELSIF (s== "data1")      THEN "\t.section .rodata"
  ELSIF (s== "string")     THEN "\t.section .rodata"
  ELSIF (s== "constpool")  THEN "\tAREA |C$$cpool|, READONLY, DATA"
  ELSE                          ""
  FI ;
}

emit (COMMENT string s)
{ emit "@ " s; }

emit (BYTE imm i)
{ emit "\t.byte " i; }

emit (HALFWORD imm i)
{ emit "\t.half " i; }

emit (WORD imm i)
{ emit "\t.word " i; }

emit (STRING string s)
{ emit "\t.ascii \"" s "\""; }
emit (SKIP int i)
{ emit "\t.skip " i; }
emit (ALIGN int i)
{ emit "\t.align " i; }
emit (ZERO int i)
{ emit "\t.zero " i; }

emit (COMMON string s, int i, int j)
{ emit
  IF( j==1 ) THEN "\tEXPORT |" s "|\n\tAREA |" s "|, COMMON, NOINIT\n\t% " i "\n"
  ELSE            "\tEXPORT |" s "|\n\tAREA |" s "|, ALIGN=" j ", COMMON, NOINIT\n\t% " i "\n"
  FI;
}
```

A.2 Instruction selection file

The following is the instruction selection part of the ARM7 machine description.

```
TYPES

/*
name    C-name          abbreviation    default-value
*/
dep     "DEP"           "D"             "NULL"
int     "INT"           "I"             "0"
String  "char *"        "S"             "NULL"
regmask "REGMASK"       "R"             "NULL"
Bool    "BOOL"          "BOOL"          "FALSE"
expr    "EXPR"          "E"             "NULL"


#define SHIFT_OPS       [lsl, lsr, asr]
```

```
#define SHIFT_OPSI      [lsli, lsri, asri]


MACHINE DESCRIPTION arm

/* Machine independent instructions */

ident(dep)                      ident(1)
wrreg(int, dep)                 wrreg(1, 2)
and (dep, dep)                  and (1, 2)              COMMUTATIVE
or (dep, dep)                   or (1, 2)               COMMUTATIVE
xor (dep, dep)                  xor (1, 2)              COMMUTATIVE
andn (dep, dep)                 andn (1, 2)
bitinv (dep)                    not (1)

imul (dep, dep)                 imul (1, 2)             COMMUTATIVE
umul (dep, dep)                 umul (1, 2)             COMMUTATIVE
mla  (dep, dep, dep)            mla(1,2,3)
sub  (dep, dep)                 isub (1, 2)
subi (dep, int)                 isubi (2, 1)
rsb  (dep, dep)                 irsb (1, 2)
rsbi (dep, int)                 irsbi (2, 1)
cmp  (dep, dep)                 icmp (1, 2)
cmpi (dep, int)                 icmpi (2, 1)
subs (dep, dep)                 isubs (1, 2)
subsi(dep, int)                 isubsi (2, 1)
add  (dep, dep)                 iadd (1, 2)             COMMUTATIVE
addi (dep, int)                 iaddi (2, 1)
lsl  (dep, dep)                 lsl (1, 2)
lsli (dep, int)                 lsli (2, 1)
lsr  (dep, dep)                 lsr (1, 2)
lsri (dep, int)                 lsri (2, 1)
asr  (dep, dep)                 asr (1, 2)
asri (dep, int)                 asri (2, 1)

push(dep, regmask)              push(2, 1)
pop(dep, regmask)               pop(2, 1)

ld32 (dep)                      ld32(1)
ld32i(dep, int)                 ld32i(2,1)
ld32r(dep, dep)                 ld32r(1,2)              COMMUTATIVE
ld32x(dep, dep)                 ld32x(1,2)

ild16 (dep)                     ild16(1)
ild16i(dep, int)                ild16i(2,1)
ild16r(dep, dep)                ild16r(1,2)              COMMUTATIVE

ild8 (dep)                      ild8(1)
ild8i(dep, int)                 ild8i(2,1)
ild8r(dep, dep)                 ild8r(1,2)              COMMUTATIVE

uld16 (dep)                     uld16(1)
uld16i(dep, int)                uld16i(2,1)
uld16r(dep, dep)                uld16r(1,2)              COMMUTATIVE
```

```
uld8 (dep)                      uld8(1)
uld8i(dep, int)                 uld8i(2,1)
uld8r(dep, dep)                 uld8r(1,2)              COMMUTATIVE

st32 (dep, dep)                 st32i(1,2)
st32i(dep, dep, int)            st32i(3,2,1)
st32r(dep, dep, dep)            st32r(1,2,3)
st32x(dep, dep, dep)            st32x(1,2,3)

st16 (dep, dep)                 st16i(1,2)
st16i(dep, dep, int)            st16i(3,2,1)
st16r(dep, dep, dep)            st16r(1,2,3)

st8 (dep, dep)                  st8i(1,2)
st8i(dep, dep, int)             st8i(3,2,1)
st8r(dep, dep, dep)             st8r(1,2,3)

jz(expr)                        jz(1)
jnz(expr)                       jnz(1)
jgt(expr)                       jgt(1)
jgtu(expr)                      jgtu(1)
jle(expr)                       jle(1)
jleu(expr)                      jleu(1)
jlt(expr)                       jlt(1)
jltu(expr)                      jltu(1)
jge(expr)                       jge(1)
jgeu(expr)                      jgeu(1)
jc(dep,expr)                    jc(1,2)
ajmp(expr)                      ajmp(1)
call(expr)                      call(1)

iles(dep, dep)    iles(1,2)
ileq(dep, dep)    ileq(1,2)
ieql(dep, dep)    ieql(1,2)     COMMUTATIVE
igeq(dep, dep)    igeq(1,2)
igtr(dep, dep)    igtr(1,2)
ineq(dep, dep)    ineq(1,2)     COMMUTATIVE
ilesi(dep, int)   ilesi(2,1)
ileqi(dep, int)   ileqi(2,1)
ieqli(dep, int)   ieqli(2,1)
igeqi(dep, int)   igeqi(2,1)
igtri(dep, int)   igtri(2,1)
ineqi(dep, int)   ineqi(2,1)
ules(dep, dep)    ules(1,2)
uleq(dep, dep)    uleq(1,2)
ueql(dep, dep)    ueql(1,2)     COMMUTATIVE
ugeq(dep, dep)    ugeq(1,2)
ugtr(dep, dep)    ugtr(1,2)
uneq(dep, dep)    uneq(1,2)     COMMUTATIVE
ulesi(dep, int)   ulesi(2,1)
uleqi(dep, int)   uleqi(2,1)
ueqli(dep, int)   ueqli(2,1)
ugeqi(dep, int)   ugeqi(2,1)
```

```
ugtri(dep, int)   ugtri(2,1)
uneqi(dep, int)   uneqi(2,1)


/* Arm instructions */

imm(int|expr)                   imm(1)
sh_imm(int, int)                sh_imm(1,2)
neg_sh_imm(int, int)            neg_sh_imm(1,2)

addiA(dep, int, int)            addiA (3, 1, 2)
subiA(dep, int, int)            subiA (3, 1, 2)
rsbiA(dep, int, int)            rsbiA (3, 1, 2)
andiA(dep, int, int)            andiA (3, 1, 2)
xoriA(dep, int, int)            xoriA (3, 1, 2)
oriA(dep, int, int)             oriA (3, 1, 2)
andniA(dep, int, int)           andniA (3, 1, 2)

addx(dep, dep, OPR, dep)        addx (2, 3, 4, 1)
subx(dep, dep, OPR, dep)        subx (2, 3, 4, 1)
rsbx(dep, dep, OPR, dep)        rsbx (2, 3, 4, 1)
andx(dep, dep, OPR, dep)        andx (2, 3, 4, 1)
xorx(dep, dep, OPR, dep)        xorx (2, 3, 4, 1)
orx(dep, dep, OPR, dep)         orx (2, 3, 4, 1)
andnx(dep, dep, OPR, dep)       andnx (2, 3, 4, 1)

addxi(dep, dep, OPR, int)       addxi (2, 3, 4, 1)
subxi(dep, dep, OPR, int)       subxi (2, 3, 4, 1)
rsbxi(dep, dep, OPR, int)       rsbxi (2, 3, 4, 1)
andxi(dep, dep, OPR, int)       andxi (2, 3, 4, 1)
xorxi(dep, dep, OPR, int)       xorxi (2, 3, 4, 1)
orxi(dep, dep, OPR, int)        orxi (2, 3, 4, 1)
andnxi(dep, dep, OPR, int)      andnxi (2, 3, 4, 1)

ld32_xi(dep,dep,OPR,int)        ld32_xi(1,2,3,4)
st32_xi(dep,dep,dep,OPR,int)    st32_xi(1,2,3,4,5)

uld8_xi(dep,dep,OPR,int)        uld8_xi(1,2,3,4)
st8_xi(dep,dep,dep,OPR,int)     st8_xi(1,2,3,4,5)


SUPPORTS

wrreg
and or xor andn bitinv
imul umul mla
sub subi rsb rsbi
cmp cmpi subs subsi
add addi
lsl lsli lsr lsri asr asri
push pop
ld32 ld32i ld32r ld32x
ild16 ild16i ild16r
ild8 ild8i ild8r
```

```
uld16 uld16i uld16r
uld8 uld8i uld8r
st32 st32i st32r st32x
st16 st16i st16r
st8 st8i st8r
jz jnz jgt jgtu jle jleu jlt jltu jge jgeu jc ajmp call
iles ileq ieql igeq igtr ineq
ilesi ileqi ieqli igeqi igtri ineqi
ules uleq ueql ugeq ugtr uneq
ulesi uleqi ueqli ugeqi ugtri uneqi
imm sh_imm neg_sh_imm
addiA subiA rsbiA andiA xoriA oriA andniA
addx subx rsbx andx xorx orx andnx
addxi subxi rsbxi andxi xorxi orxi andnxi
ld32_xi st32_xi uld8_xi st8_xi

EXTERNAL
extern Bool     IS_NULL(dep);
extern Bool     IS_NULL_OPR(OPR);
extern OPR      OPERATOR (dep);
extern Bool     CAN_BE_ROTATER_IMM (int);
extern int      GET_IMMEDIATE_BASE(int);
extern int      GET_IMMEDIATE_ROTATE(int);
extern Bool     CAN_BE_2_ROTATER_IMMS (int);
extern int      GET_2_ROTATER_IMM_BASE1(int);
extern int      GET_2_ROTATER_IMM_ROTATE1(int);
extern int      GET_2_ROTATER_IMM_BASE2(int);
extern int      GET_2_ROTATER_IMM_ROTATE2(int);
extern Bool     IS_CONSTANT(dep|int);
extern int      GET_CONSTANT(dep|int);
extern dep      FIND_LINK(dep);
extern dep      FIND_RET(dep);
extern int      KILL(dep);
extern dep      GET_SUBSIO_SUCC(dep, dep);
extern regmask  BV_INVERT_PC_LR(regmask);
extern Bool     LR_IS_POPPED(regmask);
extern dep      COPY_AFTERS(dep,dep);
extern dep      SET_LEAVE(dep, dep);
extern dep      REPLACE(dep,dep,int);
extern dep      SET_REPL(dep,dep);
extern dep      ADD_AFTER(dep,dep);
extern dep      ADD_BEFORE_TREE_CMP(dep);

extern Bool     IS_POWER_OF_TWO(int);
extern int      GET_POWER_OF_TWO(int);
extern dep      ADD_CONDITION(dep, dep, Bool);
extern dep      BIND(dep, dep, dep);

extern Bool     PHASE_OPT();
extern Bool     PHASE_NORM();

FUNCTIONS

OPR get_iopA
```

```
    (op=[add,addi])  -> addiA,
    (op=[sub,subi])  -> subiA,
    (op=[rsb,rsbi])  -> rsbiA,
    (op=and)  -> andiA,
    (op=xor)  -> xoriA,
    (op=or)   -> oriA,
    (op=andn) -> andniA;

OPR get_reverse_op
  (x=ieql)      -> ineq,
  (x=ieqli)     -> ineqi,
  (x=igeq)      -> iles,
  (x=igeqi)     -> ilesi,
  (x=igtr)      -> ileq,
  (x=igtri)     -> ileqi,
  (x=ileq)      -> igtr,
  (x=ileqi)     -> igtri,
  (x=iles)      -> igeq,
  (x=ilesi)     -> igeqi,
  (x=ineq)      -> ieql,
  (x=ineqi)     -> ieqli,

  (x=ueql)      -> uneq,
  (x=ueqli)     -> uneqi,
  (x=ugeq)      -> ules,
  (x=ugeqi)     -> ulesi,
  (x=ugtr)      -> uleq,
  (x=ugtri)     -> uleqi,
  (x=uleq)      -> ugtr,
  (x=uleqi)     -> ugtri,
  (x=ules)      -> ugeq,
  (x=ulesi)     -> ugeqi,
  (x=uneq)      -> ueql,
  (x=uneqi)     -> ueqli;

RULES

x=[addi,subi](a, b)  | CAN_BE_ROTATER_IMM(b)
  -> ia(val_b, val_s, a)
WHERE
  int val_b = GET_IMMEDIATE_BASE(b)
  int val_s = GET_IMMEDIATE_ROTATE(b)
  OPR(int,int,dep) ia = get_iopA (OPERATOR(x))
  ;

x=[add,sub,and,xor,or,andn](a, b)
  | IS_CONSTANT(b) && CAN_BE_ROTATER_IMM(b)
  -> ia(val_b, val_s, a)
WHERE
  int val_b = GET_IMMEDIATE_BASE(b)
  int val_s = GET_IMMEDIATE_ROTATE(b)
  OPR(int,int,dep) ia = get_iopA (OPERATOR(x))
  ;
```

```
create_short_immediate:
imm(int a) | CAN_BE_ROTATER_IMM(a) && PHASE_NORM()
  -> sh_imm(base,rot)
WHERE
  int base = GET_IMMEDIATE_BASE(a)
  int rot  = GET_IMMEDIATE_ROTATE(a)
  ;

create_negative_short_immediate:
imm(int a) | CAN_BE_ROTATER_IMM(~a) && PHASE_NORM()
  -> neg_sh_imm(base,rot)
WHERE
  int base = GET_IMMEDIATE_BASE(~a)
  int rot  = GET_IMMEDIATE_ROTATE(~a)
  ;

introduce_immediate_variants:
op=ANY (a, sh_imm(b,c)) | !IS_NULL_OPR(iopA)
 -> iopA (b,c,a)
WHERE
  OPR(int, int, dep) iopA = get_iopA(OPERATOR(op));

sub (a,b) | IS_CONSTANT(a) -> rsb (b, a) ;

add (a,x=[lsl,lsr,asr](b, c)) -> addx(a,b,OPERATOR(x),c);
sub (a,x=[lsl,lsr,asr](b, c)) -> subx(a,b,OPERATOR(x),c);
rsb (a,x=[lsl,lsr,asr](b, c)) -> rsbx(a,b,OPERATOR(x),c);
and (a,x=[lsl,lsr,asr](b, c)) -> andx(a,b,OPERATOR(x),c);
xor (a,x=[lsl,lsr,asr](b, c)) -> xorx(a,b,OPERATOR(x),c);
or  (a,x=[lsl,lsr,asr](b, c)) -> orx(a,b,OPERATOR(x),c);
andn(a,x=[lsl,lsr,asr](b, c)) -> andnx(a,b,OPERATOR(x),c);

add (a,x=[lsli,lsri,asri](b, c)) -> addxi(a,b,OPERATOR(x),c);
sub (a,x=[lsli,lsri,asri](b, c)) -> subxi(a,b,OPERATOR(x),c);
rsb (a,x=[lsli,lsri,asri](b, c)) -> rsbxi(a,b,OPERATOR(x),c);
and (a,x=[lsli,lsri,asri](b, c)) -> andxi(a,b,OPERATOR(x),c);
xor (a,x=[lsli,lsri,asri](b, c)) -> xorxi(a,b,OPERATOR(x),c);
or  (a,x=[lsli,lsri,asri](b, c)) -> orxi(a,b,OPERATOR(x),c);
andn(a,x=[lsli,lsri,asri](b, c)) -> andnxi(a,b,OPERATOR(x),c);

x=ajmp(l) | !IS_NULL(link)
-> c
WHERE dep link=FIND_LINK(x)
      dep c=COPY_AFTERS(call(l),link)
      int kill=KILL(link);

cmpi(r,0) -> subsi(r,0);

w=wrreg(r, x) | !IS_NULL(c)
-> wrreg(r, c)
WHERE dep c=GET_SUBSIO_SUCC(x,w);

pop_return:
wrreg(r,x=pop(sp,mask))
```

```
  | r==13 && !IS_NULL(ret) && LR_IS_POPPED(mask)
-> wrreg(r,p)
WHERE dep ret=FIND_RET(x)
      regmask m=BV_INVERT_PC_LR(mask)
      dep p = REPLACE(x,pop(sp,m),0)
      dep kill=SET_REPL(ret,p);

/* addressing modes */
ld32(a)   -> ld32i  (a,0) ;
ild16(a)  -> ild16i (a,0) ;
uld16(a)  -> uld16i (a,0) ;
ild8(a)   -> ild8i (a,0) ;
uld8(a)   -> uld8i (a,0) ;
st32(v,a) -> st32i (v,a,0) ;

ld32i (add(a,b),0)   -> ld32r(a,b);
ild16i(add(a,b),0)   -> ild16r(a,b);
uld16i(add(a,b),0)   -> uld16r(a,b);
ild8i (add(a,b),0)   -> ild8r(a,b);
uld8i (add(a,b),0)   -> uld8r(a,b);
st32i (v,add(a,b),0) -> st32r(v,a,b);
st16i (v,add(a,b),0) -> st16r(v,a,b);
st8i  (v,add(a,b),0) -> st8r(v,a,b);

ld32i (addx(a,b,op,c),0)   -> ld32_xi(a,b,op,c);
uld8i (addx(a,b,op,c),0)   -> uld8_xi(a,b,op,c);
st32i (v,addx(a,b,op,c),0) -> st32_xi(v,a,b,op,c);
st8i  (v,addx(a,b,op,c),0) -> st8_xi(v,a,b,op,c);

ld32r (a,x=[lsli,lsri,asri](b,c))   -> ld32_xi(a,b,OPERATOR(x),c);
uld8r (a,x=[lsli,lsri,asri](b,c))   -> uld8_xi(a,b,OPERATOR(x),c);
st32r (v,a,x=[lsli,lsri,asri](b,c)) -> st32_xi(v,a,b,OPERATOR(x),c);
st8r  (v,a,x=[lsli,lsri,asri](b,c)) -> st8_xi(v,a,b,OPERATOR(x),c);

[imul,umul](x,c) | IS_CONSTANT(c) && IS_POWER_OF_TWO(c)
    -> lsli(x,n)
    WHERE int n = GET_POWER_OF_TWO (c);

[imul,umul](x,c) | IS_CONSTANT(c) && IS_POWER_OF_TWO(c-1)
    -> add (lsli(x,n), x)
    WHERE int n = GET_POWER_OF_TWO (c-1);

[imul,umul](x,c) | IS_CONSTANT(c) && IS_POWER_OF_TWO(c+1)
    -> sub (lsli(x,n), x)
    WHERE int n = GET_POWER_OF_TWO (c+1);

add([imul,umul](a,b),c)          -> mla(a,b,c);
sub(a,[imul,umul](b,imm(int c))) -> mla(b,imm(-c),a);

/* control flow */
jc(x=iles (a,b),c) -> ADD_AFTER(jlt(c),REPLACE(x,cmp(a,b),0));
jc(x=ileq (a,b),c) -> ADD_AFTER(jle(c),REPLACE(x,cmp(a,b),0));
jc(x=ieql (a,b),c) -> ADD_AFTER(jz (c),REPLACE(x,cmp(a,b),0));
jc(x=igeq (a,b),c) -> ADD_AFTER(jge(c),REPLACE(x,cmp(a,b),0));
```

```
jc(x=igtr (a,b),c) -> ADD_AFTER(jgt(c),REPLACE(x,cmp(a,b),0));
jc(x=ineq (a,b),c) -> ADD_AFTER(jnz(c),REPLACE(x,cmp(a,b),0));

jc(x=ilesi(a,b),c) -> ADD_AFTER(jlt(c),REPLACE(x,cmpi(a,b),0));
jc(x=ileqi(a,b),c) -> ADD_AFTER(jle(c),REPLACE(x,cmpi(a,b),0));
jc(x=ieqli(a,b),c) -> ADD_AFTER(jz (c),REPLACE(x,cmpi(a,b),0));
jc(x=igeqi(a,b),c) -> ADD_AFTER(jge(c),REPLACE(x,cmpi(a,b),0));
jc(x=igtri(a,b),c) -> ADD_AFTER(jgt(c),REPLACE(x,cmpi(a,b),0));
jc(x=ineqi(a,b),c) -> ADD_AFTER(jnz(c),REPLACE(x,cmpi(a,b),0));

jc(x=ules (a,b),c) -> ADD_AFTER(jltu(c),REPLACE(x,cmp(a,b),0));
jc(x=uleq (a,b),c) -> ADD_AFTER(jleu(c),REPLACE(x,cmp(a,b),0));
jc(x=ueql (a,b),c) -> ADD_AFTER(jz  (c),REPLACE(x,cmp(a,b),0));
jc(x=ugeq (a,b),c) -> ADD_AFTER(jgeu(c),REPLACE(x,cmp(a,b),0));
jc(x=ugtr (a,b),c) -> ADD_AFTER(jgtu(c),REPLACE(x,cmp(a,b),0));
jc(x=uneq (a,b),c) -> ADD_AFTER(jnz (c),REPLACE(x,cmp(a,b),0));

jc(x=ulesi(a,b),c) -> ADD_AFTER(jltu(c),REPLACE(x,cmpi(a,b),0));
jc(x=uleqi(a,b),c) -> ADD_AFTER(jleu(c),REPLACE(x,cmpi(a,b),0));
jc(x=ueqli(a,b),c) -> ADD_AFTER(jz  (c),REPLACE(x,cmpi(a,b),0));
jc(x=ugeqi(a,b),c) -> ADD_AFTER(jgeu(c),REPLACE(x,cmpi(a,b),0));
jc(x=ugtri(a,b),c) -> ADD_AFTER(jgtu(c),REPLACE(x,cmpi(a,b),0));
jc(x=uneqi(a,b),c) -> ADD_AFTER(jnz (c),REPLACE(x,cmpi(a,b),0));
```

Appendix B

Definitions and Abbreviations

$a \rightarrow b$	dependence edge, a must be executed after b.
$a \rightsquigarrow b$	data dependence edge, a takes the result of b as an argument.
rdreg	pseudo instruction representing the parent of a globally allocated, inter-dtree data dependence edge, read as "read from register". Normally located at the top of a dtree.
wrreg	pseudo instruction representing the child of a globally allocated, inter-dtree data dependence edge, read as "write to register". Normally located at the bottom of a dtree.
RISC	Reduced Instruction Set Computer
VLIW	Very Large Instruction Word
ARM7	A RISC machine developed by the ARM company
LRA	Local Register Allocation
GRA	Global Register Allocation

Bibliography

[ANS89] ANSI. Programming Language - C (ANSI X3.159-1989). 1989.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[Cha82] Gregory J. Chaitin. Register Allocation and Spilling via Graph Coloring. In *SIGPLAN Symposium on Compiler Construction*, pages 98–105, 1982.

[FL98] M. Farach and V. Liberatore. On local register allocation. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1998.

[Fre74] R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17(11):638–642, 1974.

[HFG89] W. C. Hsu, C. N. Fischer, and J. R. Goodman. On the Minimization of Loads/Stores in Local Register Allocation. *IEEE Transactions on Software Engineering*, 15(10):1252–1260, 1989.

[Int92] UNIX International. DWARF Debugging Information Format. 1992.

[Les] Robert Leslie. MAD homepage: http://www.mars.org/home/rob/proj/mpeg/.

[Tie89] Michael D. Tiemann. The GNU Instruction Scheduler. July 1989.

[War90] Henry S. Warren. Instruction Scheduling for the IBM RISC System/6000 Processor. *IBM Journal Research Development*, 34(1):85–92, January 1990.