A peephole optimiser for SimpleScalar DLX assembly code

J. Stork, L. Swartsenburg and J. Zuiddam

January 2, 2012

Contents

1. Introduction

In this report, we describe our implementation of a "peephole" code optimiser. Such an optimiser scans and performs transformations on a "peephole" of source code, i.e. on a sliding sub-sequence of instructions in that source code. The function of a peephole optimiser is to improve a computer programme at compile-time by reducing its typical resource footprint in terms of CPU time, memory size, or both CPU and memory usage. The peephole optimiser assignment concludes our undergraduate course on compilers at the University of Amsterdam.

2. Implementation

In this section we discuss our peephole optimiser. We first give an overview of the general design of the optimiser. We then discuss the implementation of the parts of the optimiser.

2.1. Design

A peephole optimiser rewrites assembly code to make ik more efficient, in terms of processor cycles and memory usage. The general procedure is:

1. assembly parsing;
2. division in basic blocks;
3. optimisation;
4. assembly generation.

See also figure 2.1. We discuss stages 1 and 4 first and then stage 2 and stage 3.

Assembly parsing and generation

As processing an assembly program in the form of a set of text strings is difficult, the optimiser translates the input to an intermediate representation (ir) in the first stage. This translation is called *parsing*. The resulting representation is a list of Python objects, where each object represents an assembly instruction. After optimising, the resulting intermediate representation is translated back to assembly code. Ideally, when no optimisation is done between parsing assembly and regenerating assembly, the composite function $4 \circ 1$ is the identity function on the set of all assembly programs. (In our implementation, comments are left out in some cases, and spacing is sometimes changed.)

Division in basic blocks

Before optimising, the ir is split in *basic blocks*. A basic block is a set of consecutive instructions having a single entry point and a single exit point. By definition, only the first line i_1 of a basic block $(i_1, \ldots, i_k)$ can be a jump destination and only the last line i_k can be a jump. Therefore, the execution flow is always $i_1 \to i_2 \to \cdots \to i_k$. This property reduces the complexity of doing certain optimisations on instructions in a basic block.

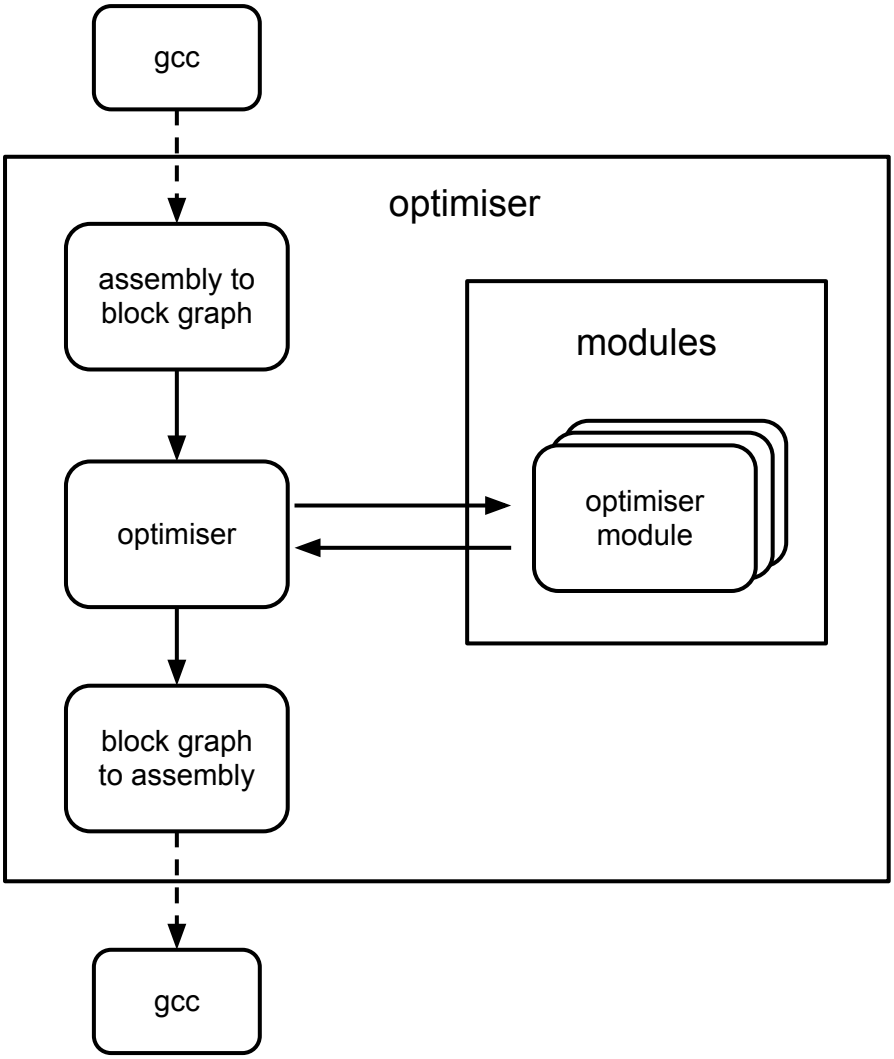

Figure 2.1.: Optimiser design diagram.

Basic blocks can in turn be divided in functions, linked together by *jump and link* instructions, `jal`. We call the set of instructions in a function a *frame*. Frames have a single entry point (on frame level), but not necessarily a single exit point.

Optimisation

In the optimisation stage, the original intermediate representation A is transformed into a new ir B, such that

1. the new representation B has the same *meaning* as A, in terms of program output and behaviour;

2. the execution of the program corresponding to B takes less processor cycles and uses less memory.

Optimisation types

There are two main types of optimisations: graph level optimisations and block level optimisations. Our optimiser performs the following graph optimisations:

- unused block removal,
- graph flattening, and
- global branch optimisations,

and the following block optimisations:

- dead code elimination,
- copy propagation, and
- constant folding.

The optimisations are explained in the implementation sections that follow.

Scheduling

As there are multiple optimisations to apply and a single optimisation may be applied multiple times, we must decide how to schedule them. In this sections we will regard graph and block optimisations as functions on the set of control flow graphs and ir blocks, respectively.

Let f be an optimisation on blocks. It is possible that an application of f on a block B yields a block that in turn can be optimised. That is, applying f on $f(B)$ will yield a different result than $f(B)$. Therefore, f is applied to B until the result is constant, i.e. n times, with

$$f^{(n)}(B) = f^{(n-1)}(B).$$

We denote the function that applies f until the result is stable by f^*. Note that $f^{(k)}(B)$ for some k need not necessarily be more efficient than B to be useful, because other optimisations can take advantage of the changes made. For example, copy propagation only pays off after dead code elimination.

Let f and g be two different optimisations on blocks. We could apply f^* to a block B and then g^*. But, as f^* and g^* may be mutually beneficial, the composite application $g^* \circ f^*$ must be repeated until the result is constant.

The same properties hold when f and g are graph level optimisations, *mutatis mutandis*. We could even mix graph and block optimisations, where block optimisations are applied to all blocks in the graph concerned. Let $F_1, \ldots, F_m$ be such optimisations.

Then, every optimisation must be repeated until constant, which we called F_i^*, and the composite optimisation must also be repeated until constant, giving

$$(F_m^* \circ \cdots \circ F_1^*)^*.$$

This is the general scheduling recipe.

We did not consider the order of applying optimisations. A particular order could be optimal. However, as the composite optimisation is repeated until the result is constant, the precise order probably is not decisive.
In the following sections we will look at the implementation of the four stages

2.2. Instruction parsing

For instruction parsing we use Python Lex-Yacc (PLY), an implementation of Lex and Yacc for Python. The Lex part of PLY splits the source in *tokens*, e.g. a `COMMENT`, or a comma. These tokens are passed to Yacc, which finds hierarchical structure in them. When Yacc recognizes a certain pattern, a corresponding action is taken.

2.2.1. Lex

Our lexer splits the source in six tokens: `COMMENT` for comments, `RAW` for compiler directives (lines starting with a dot), `HEX` for hexadecimal values, `INT` for integer values (unsigned), `ID` for label names and instruction names, and `REGISTER` for register identifiers (`$1`, `$2`, etc.) The following regular expressions specify the tokens.

```
COMMENT  : #.*
RAW      : \..+
HEX      : 0x[0-9a-f]+
INT      : [0-9]+
ID       : ($L){0,1}[_a-zA-Z0-9][_a-zA-Z0-9\.]*
REGISTER : $[a-z0-9]+
```

In addition, we consider the characters ‘`,`’, ‘`(`’, ‘`)`’, ‘`:`’, ‘`.`’, ‘`+`’, and ‘`-`’ as individual tokens, called *literals*.

2.2.2. Yacc

We specified the structure of the assembly language according to the following grammar rules.

⟨*expr*⟩ := ⟨*raw*⟩
| ⟨*label*⟩
| ⟨*comment*⟩
| ⟨*instr*⟩
| ⟨*instr*⟩ ⟨*comment*⟩

⟨*raw*⟩ := `RAW`

⟨*label*⟩ := `ID` ‘`:`’

⟨*comment*⟩ := `COMMENT`

⟨*instr*⟩ := `ID`
| `ID` ⟨*arg*⟩
| `ID` ⟨*arg*⟩ ‘`,`’ ⟨*arg*⟩
| `ID` ⟨*arg*⟩ ‘`,`’ ⟨*arg*⟩ ‘`,`’ ⟨*arg*⟩

⟨*arg*⟩ := ⟨*int*⟩
| ⟨*hex*⟩
| ⟨*register*⟩
| `ID`
| `ID` ‘`+`’ ⟨*arg*⟩
| `ID` ‘`-`’ ⟨*arg*⟩
| ⟨*int*⟩ ‘`(`’ ⟨*register*⟩ ‘`)`’
| `ID` ‘`(`’ ⟨*register*⟩ ‘`)`’

⟨*int*⟩ := `INT`
| ‘`-`’ `INT`

⟨*hex*⟩ := `HEX`

⟨*register*⟩ := `REGISTER`

Each expression is stored in a Python object of type `Raw`, `Label`, `Comment`, `Instr` or `Register`.

2.2.3. Registers

To perform dataflow analysis and liveness analysis, we need to parse each instruction to determine which registers are set and which registers are used. All instructions have a different format, so they need to be parsed seperatly. This process is done in the `parse_instr` module. When we run `parse_instr.parse`, it parses each Instr object from the `ir` module and sets the register that the instructoin writes to, the registers that it needs, the offset for writing to memory, and immidiate values. An example:

```
1  if ins.instr == 'add':
2      if len(ins.args) == 3:
3          self.gen = [ins.args[0]]
4          if self.is_reg(ins.args[2]):
5              self.need = [ins.args[1], ins.args[2]]
6          else:
7              self.need = [ins.args[1]]
8              self.ival = ins.args[2]
9      else:
10         raise Exception("Invalid number of args for ins: ", ins.instr)
```

2.3. Division of instructions into basic blocks

As mentioned in section 2.1 we split the source code on two levels.

- The code is split in functions.
- Each function is split in *basic blocks*.

The blocks of each frame are stored in a *control flow graph* (cfg). The edges in this graph correspond to possible branches.

Splitting in blocks is implemented in `cfg.py`.

2.4. Control Flow Graph optimisations

In the section 2.3 the code is divided in frames and after that in basic blocks. It is also possible to create a graph from the whole code. If this is done, it is possible to perform optimalisations on the graph. We implemented two optimalisations in the module `optimise_tree`.

Remove not used

A program has only one entry point: the beginning of the instruction list. The first block should be the only block with no incoming edges. If there are more blocks with no incoming edges, then these can be removed since they will never be reached. The blocks with no edges are the blocks that are between a jump and a label in the assembly code:

```
1       addu    $2,$3,4
2       j       $L6
3   |   bne     $2,$0,$L5
4   |   mtc1    $2,$f0
5   |   lw      $31,52($sp)
6   |   lw      $fp,48($sp)
7   |   .loc    1 5
8   |   .ent    main
9   $L6:
10      div.d   $f0,$f0,$f2
```

In this case, there are two lines that contain raw code. This code is used by gcc and can therefore not be removed. So we keep `.loc` and `.end` and we remove `bne`, `mtc1` and the `lw` instructions. If the raw types wouldn't be there, the jump would now be optimised in 2.5.

Flatten the graph

For all blocks that have one incoming edge (now: "block two") and a block on the other side that has only one outgoing edge (now: "block one"), the blocks can be placed after eachother in the code. This can work out well when the only edge to a block is caused by a jump. This jump will become redundant if the blocks are repositioned. We implented this by appending all instructions from block two to the instructions from block one and by removing block two as a block object.

Result

We made a code that generates a diagram from our graph when the verbosity level is three. We generate a diagram image before and after optimalisation. The result is displayed for pi.s in a 2.2 and 2.3.

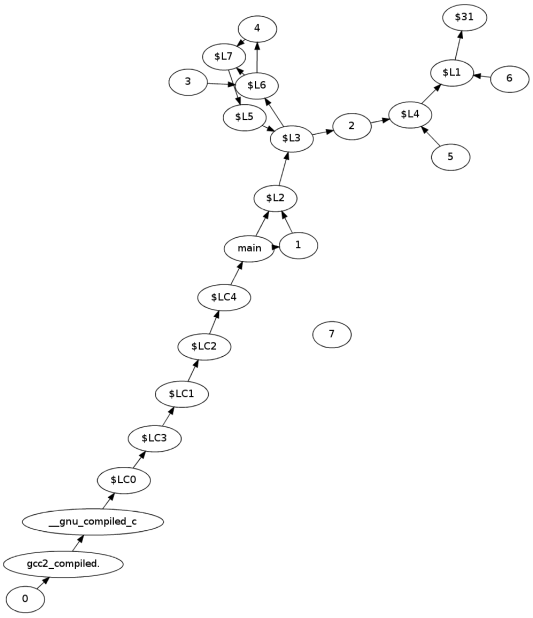

Figure 2.2.: CFG for pi.s before optimalisation

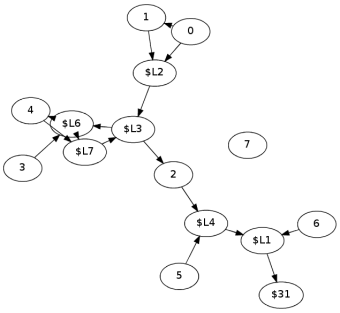

Figure 2.3.: CFG for pi.s after optimalisation

After the graph optimalisations we need to perform global jump optimalisations to actually gain performance.

2.5. Global branch optimisations

The flat module contains five different optimalisation functions that remove redundant code on a global basis. In this subsection we will elaborate on those functions. These functions perform better when comments are removed so there is a function that removes all comments from the code. We decided to not lose information in the assembly code.

Jump - code - Label

When there is a jump, every code that is between the jump and the label is never reached. Thus: the instruction between a jump and a label can be removed. This function probably will not speed up the code, but can reduce lines in the assembly code. Example:

```
1       addu    $2,$3,4
2       lw      $3,60($fp)
3       j       $L4
4       mtc1    $2,$f0
5       div.d   $f0,$f0,$f2
6   $L6:
7       jal     random
```

Becomes:

```
1       addu    $2,$3,4
2       lw      $3,60($fp)
3       j       $L4
4   $L6:
5       jal     random
```

Label - Label

When two labels placed directly after eachother, one of the two is redundant. The second label is removed and each jump in the code that jumps to the second label is changed to jump to the first label. This function can reduce lines and basicblocks for later on in optimalisation. In combination with the jump optimalisation "useless branches" this can potentially speed up the program. Example:

```
1       addu    $2,$3,4
2       lw      $3,60($fp)
3       bne     $2,$0,$L6
4       mtc1    $2,$f0
5       div.d   $f0,$f0,$f2
6   $L5:
7   $L6:
8       jal     random
```

Becomes:

```
1       addu    $2,$3,4
2       lw      $3,60($fp)
3       bne     $2,$0,$L5
```

```
4       mtc1    $2,$f0
5       div.d   $f0,$f0,$f2
6   $L5:
7       jal     random
```

Label - Jump

If a jump comes directly after a label, both are redundant. We can remove both lines and adjust all control instructions that point to the label so that they point to the label the jump was pointing to. If this situation occurs, two lines of assembly can be removed and the program is optimised by removing one jump. Example:

```
1       addu    $2,$3,4
2       bne     $2,$0,$L5
3       mtc1    $2,$f0
4   $L5:
5       j       $L8
6   $L6:
7       div.d   $f0,$f0,$f2
```

Becomes:

```
1       addu    $2,$3,4
2       bne     $2,$0,$L8
3       mtc1    $2,$f0
4   $L6:
5       div.d   $f0,$f0,$f2
```

Useless branch/jump

If a control function jumps to a label that comes directly after the jump/branch , the instruction can be removed. This results in a smaller assembly file and a faster program. Example:

```
1       addu    $2,$3,4
2       bne     $2,$0,$L6
3   $L6:
4       div.d   $f0,$f0,$f2
```

Becomes:

```
1       addu    $2,$3,4
2   $L6:
3       div.d   $f0,$f0,$f2
```

Branch - Jump - Label

This function removes a jump when we find a branch - jump - label combination where the branch points to the label. We can optimise this by negating the brench, making it point to the jump adres and removing the label. This results in a smaller assembly file and a faster program. Example:

```
1       slt     $2,$2,$3
2       bne     $2,$0,$L6
3       j       $L4
4   $L6:
5       jal     random
```

Becomes:

```
1       slt     $2,$2,$3
2       beq     $2,$0,$L4
3   $L6:
4       jal     random
```

2.6. Peephole optimisation on basic blocks

This section describes the framework for the basic-block optimisers in our peephole optimiser, before discussing the different optimisers as we have implemented them.

In order to focus our optimisation efforts, we drew up a ranking of instructions in the benchmark suite (see table 2.1 on page 15) using an instruction ranker we made for the purpose (source in appendix section A.14 on page 94). Based on the incidence of certain instructions, and on the fact that we wished to develop basic-block optimisers in parallel with both global and "advanced" (dataflow and liveness analysis based) optimisations, we decided to focus on: copy propagation; dead code removal; constant folding; and algebraic transformations. Subsections 2.6.2 through 2.6.5 below describe these basic-block optimisations before discussing the choices and considerations involved in their respective implementations.

2.6.1. Framework

The `optimise` module's `Optimiser` class contains the `optimise()` function that serves, amongst other things, to instantiate and run the basic-block optimisers. After instantiating the optimisers, `optimise()` runs a loop that calls a sequence of basic-block sub-optimisations until this sequence stops altering the source code.

The `BlockOptimiser` class in the `block_optimise` module contains common attributes and tools needed for the different basic-block optimisations implemented in `BlockOptimiser`'s subclasses (also defined in `block_optimise`. Notable amongst these tools are:

- `block`: This is the `BasicBlock` object assigned to this optimiser for optimisation. See the `cfg` module for the definition of a `BasicBlock`
- `optimise()`: The top-level `Optimiser` object mentioned earlier calls this function to execute the sub-optimisation. This function in turn enters a loop of calls to the

opcode	occurences	opcode	occurences	opcode	occurences
lw	1885	sub.d	67	bgtz	7
addu	1017	mov.d	60	c.eq.d	6
l.d	789	beq	48	srl	6
move	772	cvt.d.w	39	sltu	5
sw	651	mtc1	39	lbu	4
sll	561	add.s	32	mfhi	4
j	372	lb	24	blez	4
s.d	347	dsz	22	dmfc1	4
jal	297	bc1f	22	trunc.w.d	4
la	239	cvt.d.s	21	cvt.s.d	4
mul.d	199	nop	21	c.le.d	4
subu	193	c.lt.d	20	div	4
l.s	163	abs.d	19	neg.s	3
li	150	sra	17	dsw	3
add.d	150	mul.s	13	cvt.s.w	3
bne	140	dlw	12	xori	2
slt	130	sb	12	div.s	2
s.s	78	neg.d	11	mov.s	2
mflo	77	bgez	11	bltz	1
mult	77	sub.s	11	mfc1	1
div.d	71	bc1t	8	or	1

Table 2.1.: Ranking of instructions

`suboptimisation` function, which ends when the suboptimisation reports that a pass failed to change the code further.

- `suboptimise()`: Implemented in each subclass of `BlockOptimiser`, this function scans the peephole until it finds a trigger instruction, at which point it calls the relevant suboptimisation function, which in turn scans the rest of the peephole for optimisation opportunities. Note that from this function up to the top `Optimiser`, an `opt` or `optimised` boolean is return to indicate whether an optimisation was executed at a lower level. This enables higher-level functions move on to the next optimisation pass.

- `reg_indexes_in()`: This function serves to obtain a list of indexes at which a register was found an instruction's arguments list. Since registers are sometimes used with an offset, e.g. `16($fp)`, this function resorts to a regular expression matcher to find reliably find instances of the register.

- `reg_in()`: Returns a boolean to indicate whether a register is in an instruction's arguments.

- `replace_reg()`: Replaces a reference to one register with a reference to another, in an instruction's arguments list. Like `reg_indexes_in()`, this function uses regex matching to deal with offset plus register value addressing.
- `find_constants()`: This function searches code above the trigger instruction (in the case of constant folding) for `li` instructions, and extracts the compile-time constants and their corresponding registers.

The `peephole` module's `Peephole` and `Peeper` classes lend our optimiser its famous "peephole" quality. The `Peeper` class generates the successive peepholes required to produce the sliding peephole functionality for our optimisers. Like `BasicBlock` objects, `Peephole` objects emulate Python's built-in iterable containers, making for easy referencing, iteration and assignments of instructions in the peephole. The overall object-oriented structure of our peepholes and basic blocks allows for a more elegant handling of the peephole concept, where instructions are handled through various layers of abstraction.

The `uic` module contains "useful instruction categories", notably: `copy_prop_targets`, which lists instructions that may be targetted for copy propagation; and `assign_to`, which is a dictionary listing instructions that assign a value to a register, together with corresponding assigned-to register's index in the instruction's arguments list.

Though not a part of our optimiser's basic block component, the unittests in `test_block_optimise` were essential to the development process. They served to test new functionality as it was added; to replicate bugs and test fixes; and to ensure that functionality did not break after code changes.

Finally, it is worth mentioning that the block optimisers write statistics to the logger regarding the numbers of the respective optimisations carried out. These are printed at the top level when it has finished cycling the basic-block optimisations. Additionally, messages a logged for each instance of optimisation at verbosity level 4 ("debug").

2.6.2. Copy propagation

Copy propagation paves the way for dead code removal (see section 2.6.3 below). It does so by finding instructions that copy the value in one register (let's call this the "original" register) to another register (the "copy" register). The copy propagation optimiser then scans subsequent code to substitute references to the "copy" register with references to the "original" register, where this does not alter the overall semantics of the code.

The implementation in the `block_optimise` module's `CopyPropagation` class triggers a copy propagation attempt every time it encounters a `move` instruction. Copy propagation is only performed on instructions in the `copy_prop_targets` category, defined in the `uic` module. The propagation attempt ends when either: the "original" or "copy" register are altered in a way that makes propagation unsafe further down; or the "copy"

or "original" register are found in an instruction classified as `copy_prop_unsafe` (also defined in `uic`). Subsection 2.6.1 above describes the register matching and substitution tools used in the course of copy propagation.

2.6.3. Dead code removal

Dead code removal is the process of searching peepholes for registers that, after being assigned a value in the peephole, are then not used before being assigned another value in the same peephole. Copy propagation makes more such registers apparent to the dead code removal optimiser. The dead code optimiser triggers a sub-scan whenever it encounters an instruction classified in `uic.py` as `assign_to`, i.e. as assigning a value to a register. The register that this instruction assigns to is taken via the index number stored in the same dict in `uic.py`. If the register is written to again within the same peephole without being used, the triggering instruction encountered earlier is removed from the basic block. It is worth mentioning here that the optimiser must take account of both (offset plus register value) references to normal registers, and (offset plus register value) references to registers used as function parameters for subsequent jump-and-link instructions. These appear to be registers `$f12` through `$f15` and `$4` through `$7`.

2.6.4. Constant folding

Constant folding replaces arithmetic operations involving compile-time constants with a load-immediate operation for the result of the arithmetic operation (to the same destination register). Keeping in mind the ranking of instructions in the benchmark suite, and the fact that our implementation needs an extra line of code or two for each type of arithmetic operation, we have only implemented constant folding for `addu` and `subu` instructions. These avoid the complication of bit manipulation in Python for non-unsigned instructions. They further avoid arithmetic instructions that write their results to the `hi` and `lo` registers, again due to the extra effort - albeit little more than a trivial one.

2.6.5. Algebraic transformations

The `block_opt_lab.py` file contains our work to-date on an algebraic transformations sub-optimiser, which we have chosen not to include in our final version of the optimiser. This optimiser replaces arithmetic operations with semantically equivalent operations that are more efficient. Our work to-date was on a transformation of `divd` instructions to `sra` instructions. We chose `divd` over `div` due to it high incidence (71) in the benchmark suite compared to the latter (4), and because the "manual" provided for the SimpleScalar instruction set listed `div` as writing its results to `hi` and `lo` registers, a more complicated scenario to code for. It later transpired that the occurences of `div` in the benchmark suite write their results to a `$fd` register. Whilst our implementation passed our initial unit tests, no suitable targets for optimisation were found in the benchmark suite: only operations where the denominator was a multiple of 2 could be transformed. Our only route to establishing this at compile-time are the `li` instructions in our peepholes. Yet

the `li` instructions only use the non-floating-point registers, whereas `divd` obviously uses floating-point registers.

2.7. Advanced peephole optimisations using dataflow and liveness analysis

In instruction parsing all instructions are parsed to find out to which register they write and which registers they need. With this information, dataflow analysis and liveness analysis can be done. Noteworthy is that both analysis implementations take the double precision floating point operations into account. This means, that two consecutive registers are used when appropriate.

Dataflow analysis

With a control flow graph (CFG) object we can create a dataflow object. When a dataflow object is created, first the gen set of each block is determined and set in the basicblock. After this, the reach of each block has to be calculated (which other blocks can be reached using the directed edges of the graph). With this information, we can generate the kill set for each block.

Now that we have the gen- and kill-set of each block, we can calculate the in and out sets using the reaching definitions algorithm.

An overview of the sets and the number of iterations needed for the reaching definitions algorithm can be printed by calling a print function on the dataflow object.

Liveness analysis

To optimise our assembly further, it is important to know which variables are alive at which point. That is: if a instruction writes to a register, the register is alive between the write and the last read (without writing in between). Wikipedia: "a variable is live if it holds a value that may be needed in the future".
First, we start by determining the $live_{out}$ set of each block. These are the registers that are needed somewhere in the code after the block. It is therefore crucial to examine all edges.
The $live_{in}$ sets are all variables that are needed in the block or further along in the graph, but are not set before they are needed in the block. For each instruction in the block we determine which registers are needed. This is done backwards. We can search in the block which instruction writes the last to these registers. This instruction is saved in a set called *kill*. The analysis takes into account that some jump functions have a elaborate set of registers they need. This is the case for the `jal`, `jal` and `j $31` instructions. These instructions are function calls. In the case of a `j $31` instruction,
$fp, $sp, $16-$23, $f20, $f22, $f24, $f26, $f28, $f30, $2, $3, $f0, $f1, $f2, $f3
are alive.

In the case of a `jal` or a `jalr` instruction,
$4-$7 en $f12-$f15
are alive and
$31
is set.

Liveness optimalisation

A small optimalisation is done based on our liveness analysis. We examine each instruction that writes for liveness of the written register. If the register is never used (in the graph), it can be removed.

3. Benchmarks

3.0.1. Results

Note that, for all tests, we entered 100 at both prompts for values in `slalom`. Use command line option `-v` to see more statistics.

3.0.2. Test 1

Our first benchmark test. A number of the files had errors.

file	original cycles	lines	optimised cycles	lines
pi	12988		9125	
acron	4435765		segfault(7370)!	
clinpack	1546336		segfault(7406)!	
dhrys.	2887741		compile error!	
whet	2864067		never ending!	
slalom	27254		27161	

Table 3.1.: Results of test 1

3.0.3. Test 2

Our second benchmark test. `dhrystone.s` failed to cross compile.

file	original cycles	lines	optimised cycles	lines
pi	12988	137	9125	128
acron	4435765	440	4332868	413
clinpack	1546336	3774	942222	3498
dhrys.	2887741	923	compile error!	855
whet	2864067	1058	2810428	1030
slalom	27254	4675	26950	4466

Table 3.2.: Results of test 2

3.0.4. Discussion

While we performed dataflow analysis and liveness analysis, we could only implement a basic optimalisation based on the liveness information. With more time, we could have gotten a huge performance boost using the gained data. We produced a very nice framework for optimalisations: all essential data is gathered and optimalisations are done in a way that we can expand very easy. Unfortunatly the optimalisations we implemented still need some finetuning.

A. Source

A.1. asmlex.py

```
1  #
2  # assembly lexer
3  #
4  # Run this module without arguments to use the interactive
5  # lexer. Give a filename as argument to tokenise a file.
6  #
7
8  import lex
9  import sys
10
11 tokens = (
12     'COMMENT',
13     'RAW',
14     'HEX',
15     'INT',
16     'REGISTER',
17     'ID',
18     )
19
20 def t_COMMENT(t):
21     r'\#.*'
22     return t
23
24 # We dont touch lines starting with '.'.
25 def t_RAW(t):
26     r'\..+'
27     return t
28
29 def t_HEX(t):
30     r'0x[0-9a-f]+'
31     return t
32
33 # Dont put '-' in regex. Yacc handles '-'.
34 def t_INT(t):
35     r'[0-9]+'
36     t.value = int(t.value)
37     return t
38
39 # An ID token is either a label or an instruction.
40 def t_ID(t):
41     r'(\$L){0,1}[_a-zA-Z0-9][_a-zA-Z0-9\.]*'
42     return t
43
```

```
44 # Registers are expressions $[a-z0-9]+ .
45 def t_REGISTER(t):
46     r'\$[a-z0-9]+'
47     return t
48 
49 literals = [ ',', '(', ')', ':', '.', '+', '-' ]
50 
51 # Define a rule so we can track line numbers
52 def t_newline(t):
53     r'\n+'
54     t.lexer.lineno += len(t.value)
55 
56 # A string containing ignored characters (spaces and tabs)
57 t_ignore  = ' \t'
58 
59 error_count = 0
60 # Error handling rule
61 def t_error(t):
62     print "Illegal character '%s'" % t.value[0]
63     t.lexer.skip(1)
64     global error_count
65     error_count += 1
66 
67 # Build the lexer
68 lexer = lex.lex()
69 
70 if __name__ == '__main__':
71     # file
72     if len(sys.argv) > 1:
73         lex.input(''.join(open(sys.argv[1], 'r').readlines()))
74         while 1:
75             tok = lex.token()
76             if not tok: break
77             print tok
78         print 'errors: %d\n' % error_count
79     # interactive
80     else:
81         while True:
82            try:
83                s = raw_input('asmlex > ')
84            except EOFError:
85                break
86            if not s: continue
87            lex.input(s)
88            while 1:
89                tok = lex.token()
90                if not tok: break
91                print tok
```

A.2. asmyacc.py

```
1 #
2 # assembly parser
```

```
3  #
4  # Run this module without arguments to use the interactive
5  # parser. Give a filename as argument to parse a file.
6  #
7
8  import yacc
9  import sys
10
11 from ir import *
12
13 # Get the token map from the lexer.  This is required.
14 from asmlex import tokens
15
16 raise_on_error = False
17
18 def p_expr(p):
19     '''expr : raw
20             | label
21             | comment
22             | instr
23             | instr comment
24             '''
25     p[0] = p[1]
26
27
28 def p_raw(p):
29     'raw : RAW'
30     p[0] = Raw(p[1])
31
32
33 def p_label(p):
34     '''label : ID ':' '''
35     p[0] = Label(p[1])
36
37
38 def p_comment(p):
39     'comment : COMMENT'
40     p[0] = Comment(p[1])
41
42
43 # for nop
44 def p_instr_no_arg(p):
45     '''instr : ID '''
46     p[0] = Instr(p[1], [])
47
48
49 def p_instr_one_arg(p):
50     '''instr : ID arg'''
51     p[0] = Instr(p[1], [p[2]])
52
53
54 def p_instr_two_arg(p):
55     '''instr : ID arg ',' arg'''
56     p[0] = Instr(p[1], [p[2], p[4]])
```

```
57
58
59  def p_instr_three_arg(p):
60      '''instr : ID arg ',' arg ',' arg'''
61      p[0] = Instr(p[1], [p[2], p[4], p[6]])
62
63
64  # loop.17+4($3)
65
66  def p_arg(p):
67      '''arg : int
68             | hex
69             | register
70             | ID
71             '''
72      p[0] = p[1]
73
74
75  def p_arg_ext(p):
76      '''arg : ID '+' arg
77             | ID '-' arg
78             '''
79      p[0] = '%s%s%s' %(p[1],p[2],p[3])
80
81
82  def p_arg_brackets(p):
83      '''arg : int '(' register ')'
84             | ID  '(' register ')'
85             '''
86      p[0] = '%s(%s)' %(p[1], p[3].expr)
87
88
89  def p_int_minus(p):
90      '''int : '-' INT'''
91      p[0] = -1*p[2]
92
93
94  def p_int(p):
95      'int : INT'
96      p[0] = p[1]
97
98
99  def p_hex(p):
100     'hex : HEX'
101     p[0] = p[1]
102
103
104 def p_register(p):
105     'register : REGISTER'
106     p[0] = Register(p[1])
107
108
109 error_count = 0
110 # Error rule for syntax errors
```

```
111 def p_error(p):
112     print "Syntax error in input!"
113     global raise_on_error
114     global error_count
115     error_count +=1
116     if raise_on_error: raise Exception()
117 
118 
119 # Build the parser
120 parser = yacc.yacc()
121 
122 
123 if __name__ == '__main__':
124     # file
125     if len(sys.argv) > 1:
126         raise_on_error = True
127         counter = 1
128         for line in open(sys.argv[1], 'r').readlines():
129            if not line.strip(): continue
130            result = parser.parse(line)
131            print counter, result
132            counter +=1
133            #if counter % 50 == 0: raw_input('pres key')
134         print 'errors: %d\n' % error_count
135     # interactive
136     else:
137         while True:
138            try:
139                s = raw_input('asmyacc > ')
140            except EOFError:
141                break
142            if not s: continue
143            result = parser.parse(s)
144            print repr(result)
```

A.3. block_optimise.py

```
1  """
2  File:          block_optimise.py
3  Course:        Compilerbouw 2011
4  Author:        Joris Stork, Lucas Swartsenburg, Jeroen Zuiddam
5  
6  
7  Description:
8      Defines the various subclasses of block optimiser.
9      A block optimiser carries out optimisations through a Peephole
           object on
10     BasicBlock objects (cfg.py), which represent basic blocks in the
           source
11     code.
12 
13     The folllowing types of block optimisers are currently implemented
           :
```

```
14          ConstantFold
15          DeadCode
16          CopyPropagation
17
18      Other potential bb-optimisations to be found in block_optimise_lab
            .py
19
20  """
21
22
23  from cfg import BasicBlock
24  import re
25  import ir
26  from ir import Instr
27  import math
28  from peephole import Peephole, Peeper
29  from uic import copy_prop_targets, copy_prop_unsafe, assign_to
30  import logging
31  from urc import j_thirty_one_regs, jal_regs
32
33
34  class BlockOptimiser(object):
35      """ parent class for the various block optimisations """
36
37
38      def __init__(self, block = None, peephole_size = None):
39          """ by default the peephole size is that of the basic block
                """
40
41          self.block = block
42          self.stats = {'dc':0,'cp':0,'cf':0}
43
44          if not peephole_size:
45              self.p_size = len(block)
46          else:
47              self.p_size = peephole_size
48
49
50      def set_block(self, block):
51          """ re-assigns a new block to the optimiser """
52
53          self.block = block
54
55
56      def set_peephole_size(self, size):
57          """ changes optimiser's peephole size setting """
58
59          self.p_size = size
60
61
62      def reg_indexes_in(self, subject, args):
63          """ returns indexes of occurences of subject in args """
64
65          indexes = []
```

```
66
67          for i, arg in enumerate(args):
68              if isinstance(arg, str):
69                  m = re.search('\$\w*', arg)
70                  if m:
71                      arg_reg = m.group(0)
72                  else: continue
73                  if (arg_reg == subject):
74                      indexes.append(i)
75
76          return indexes
77
78
79      def reg_in(self, subject, args):
80          """ returns true if the string is a substring of one of args[i
                ] """
81
82          return (len(self.reg_indexes_in(subject, args)) > 0)
83
84
85      def replace_reg(self, subject, arg):
86          """ replaces register in arg with subject register """
87
88          return ir.Register(re.sub('\$\w*', str(subject), str(arg)))
89
90
91      def find_constants(self, before = 0):
92          """
93          compiles a dict of (register:constant) pairs, with the
                constant
94          corresponding to the last value in the given register prior to
                 the
95          peephole[before] instruction
96
97          """
98
99          consts = {}
100
101         for ins in self.peephole[0:before]:
102             if isinstance(ins,Instr):
103                 if ins.instr == 'li':
104                     consts[ins.args[0].expr] = ins.args[1]
105
106         return consts
107
108
109     def suboptimisation(self):
110         """ defined in relevant optimisation subclass """
111
112         pass
113
114
115     def optimise(self):
```

```
116         """ if block is present: runs sub-optimisation until no
                changes left """
117
118         changed = False
119
120         if not self.block:
121             return changed
122
123         peeper = Peeper(self.block, self.p_size)
124         optimised = False
125
126         for peephole in peeper:
127             self.peephole = peephole
128             optimised = True
129             while optimised:
130                 optimised = self.suboptimisation()
131                 changed = changed | optimised
132
133         return changed
134
135
136
137 class CopyPropagation(BlockOptimiser):
138     """ after copy, propagate original variable where copy unaltered
            """
139
140     def propagate_from_move(self, i, ins, opt):
141         """
142         triggered by a move instruction; ignores non-instructions and
143         instructions not containing the relevant registers;
                substitutes
144         subsequent uses of unaltered copied-to (copy) register with
                the original
145         (orig) register, until it finds an unsafe instruction with the
                 orig or
146         copy register, or until the orig or copy register are altered
147         (substitution can still be done in the altering instruction)
148
149         """
150
151         optimised = opt
152
153         if ins.instr == 'move':
154             orig = ins.args[1]
155             copy = ins.args[0] # nb: move not explicity defined for
                    simplescalar
156
157             for i2, ins2 in enumerate(self.peephole[i+1:len(self.
                    peephole)]):
158
159                 if not isinstance(ins2,Instr):
160                     continue
161
162                 args = []
```

```
163                 for arg in ins2.args:
164                     if isinstance(arg,str) | isinstance(arg,int) :
165                         args.append(arg)
166                     else:
167                         args.append(arg.expr)
168 
169                 copy_in_args = self.reg_in(copy.expr, args)
170                 orig_in_args = self.reg_in(orig.expr, args)
171                 ins2_in_cp_targets = str(ins2.instr) in
                        copy_prop_targets
172                 ins2_in_assignments = str(ins2.instr) in assign_to
173                 assigns_to = None
174 
175                 if (ins2 in copy_prop_unsafe) & (copy_in_args |
                        orig_in_args):
176                     return optimised
177 
178                 if (not ins2_in_cp_targets) & (not ins2_in_assignments
                        ):
179                     continue
180                 elif not (copy_in_args | orig_in_args):
181                     continue
182 
183                 elif ins2_in_cp_targets:
184 
185                     if ins2_in_assignments:
186                         assigns_to = assign_to[str(ins2.instr)]
187                         wrote_copy = self.reg_in(copy.expr,[args[
                                assigns_to]])
188                         wrote_orig = self.reg_in(orig.expr,[args[
                                assigns_to]])
189                         if copy_in_args:
190                             for k in self.reg_indexes_in(copy.expr,
                                    args):
191                                 if not (k == assigns_to):
192                                     msg = 'editing: '+str(self.
                                            peephole[i+1+i2])
193                                     self.logger.debug(msg)
194                                     old = ins2.args[k]
195                                     ins2.args[k] = self.replace_reg(
                                            orig, old)
196                                     optimised = True
197                             if optimised:
198                                 self.peephole[i+1+i2] = ins2
199                                 msg = 'edited to: '+str(self.peephole[
                                        i+1+i2])
200                                 self.logger.debug(msg)
201                                 self.stats['cp'] += 1
202                         if wrote_copy | wrote_orig:
203                             return optimised
204 
205                     elif copy_in_args:
206                         for k in self.reg_indexes_in(copy.expr, args):
```

```
207                             msg = 'editing: '+str(self.peephole[i+1+i2
                                    ])
208                             self.logger.debug(msg)
209                             old = ins2.args[k]
210                             ins2.args[k] = self.replace_reg(orig, old)
211                             optimised = True
212                         self.peephole[i+1+i2] = ins2
213                         msg = 'edited to: '+str(self.peephole[i+1+i2])
214                         self.logger.debug(msg)
215                         self.stats['cp'] += 1
216
217                 elif ins2_in_assignments:
218                     assigns_to = assign_to[str(ins2.instr)]
219                     wrote_copy = self.reg_in(copy.expr,[args[
                            assigns_to]])
220                     wrote_orig = self.reg_in(orig.expr,[args[
                            assigns_to]])
221                     if wrote_copy | wrote_orig:
222                         return optimised
223
224         return optimised
225
226
227     def suboptimisation(self):
228         """ triggers propagate_from_move() when move instruction found
                """
229
230         self.logger = logging.getLogger('CopyPropagation')
231         optimised = False
232         for i, ins in enumerate(self.peephole):
233             if isinstance(ins,Instr):
234                 optimised = self.propagate_from_move(i, ins, optimised
                        )
235         return optimised
236
237
238
239 class DeadCode(BlockOptimiser):
240     """
241     finds and removes instructions that assign a value that is then
            never used.
242
243     """
244
245     def j_thirty_one_present(self):
246         """
247         (unimplemented) returns true if a j$31 instruction is present
                in
248         block
249
250         """
251
252         for instruction in self.block:
253
```

```
254             if not isinstance(instruction,Instr):
255                 continue
256             elif str(instruction.instr) == 'j':
257                 arg = instruction.args[0]
258                 if isinstance(arg, str):
259                     if arg == '$31':
260                         return True
261                 else:
262                     if arg == '$31':
263                         return True
264 
265         return False
266 
267 
268     def jal_present(self):
269         """
270         (unimplemented) returns true if a j$31 instruction is present
                in
271         block
272 
273         """
274 
275         for instruction in self.block:
276             if not isinstance(instruction,Instr):
277                 continue
278             elif str(instruction.instr) == 'jal':
279                 return True
280         return False
281 
282 
283     def subscan(self, i, ins, opt, cand_reg_index):
284         """
285         searches subsequent instructions, and: stops if candidate
                register is
286         used; or removes the candidate instruction if register is
                overwritten;
287         note: the only instruction object without an args attribute is
                 the nop
288         instruction, and such instructions are ignored; note also,
                certain
289         registers not candidates due to being used for function
                parameters or
290         other more permanent values
291 
292         """
293 
294         optimised = opt
295 
296         for i2, ins2 in enumerate(self.peephole[i+1:len(self.peephole)
                ]):
297 
298             candidate_reg = ins.args[cand_reg_index]
299             ins2_is_instruction = isinstance(ins,Instr)
300             args = []
```

```
301
302             if not ins2_is_instruction:
303                 continue
304
305             try:
306                 for arg in ins2.args:
307                     if isinstance(arg,str) | isinstance(arg,int) :
308                         args.append(arg)
309                     else:
310                         args.append(arg.expr)
311             except AttributeError:
312                 continue
313
314             if not self.reg_in(candidate_reg.expr, args):
315                 continue
316
317             elif str(ins2.instr) in assign_to:
318                 assigned_to_index = assign_to[str(ins2.instr)]
319                 pre_args = args[0:assigned_to_index]
320                 post_args = args[assigned_to_index+1:len(args)]
321                 if self.reg_in(candidate_reg.expr,pre_args+post_args):
322                     return optimised
323                 else:
324                     self.logger.debug(' being removed... '+str(self.
                        peephole[i]))
325                     del self.peephole[i]
326                     optimised = True
327                     self.logger.debug('instruction removed')
328                     self.stats['dc'] += 1
329                     return optimised
330
331             else: return optimised
332
333         return optimised
334
335
336     def suboptimisation(self):
337         """
338         triggers subscan() for every peephole instruction in assign_to
339         category; aborts if jal or j $31 instruction in block
340
341         """
342
343         optimised = False
344         if (self.j_thirty_one_present()) | (self.jal_present()):
345             return optimised
346         self.logger = logging.getLogger('DeadCode')
347         candidate = None
348         for i, ins in enumerate(self.peephole):
349             if not isinstance(ins,Instr):
350                 continue
351             elif str(ins.instr) in assign_to:
352                 cand_reg_index = assign_to[str(ins.instr)]
```

```
353                 optimised = self.subscan(i, ins, optimised,
                        cand_reg_index)
354         return optimised
355
356
357
358 class ConstantFold(BlockOptimiser):
359     """
360     replaces arithmetic instructions with only compile-time constants
            in
361     arguments, with a load immediate instruction to assign the
            corresponding
362     value to the same target register
363
364     """
365
366
367     def addu(self, i, ins, opt, consts):
368         """
369         carries out constant folding on addu instructions; though not
                guaranteed
370         to replicate unsigned behaviour, this should be ok for our
                benchmarks;
371         note that addu's, as manifest in benchmark suite, seem to
                incorporate
372         addiu functionality since no addiu's are present, and at least
                 one
373         instance of a compile-time value was found in the arguments of
                 an addu
374         instruction, in the benchmark suite
375
376         """
377
378         optimised = opt
379
380         if ins.instr == 'addu':
381
382             arg1_is_reg = isinstance(ins.args[1],ir.Register)
383             arg2_is_reg = isinstance(ins.args[2],ir.Register)
384             none_reg = (not arg1_is_reg) & (not arg2_is_reg)
385             arg1_known_reg = False
386             arg2_known_reg = False
387
388             if arg1_is_reg:
389                 arg1_known_reg = (self.reg_in(ins.args[1].expr, consts
                        ))
390             if arg2_is_reg:
391                 arg2_known_reg = (self.reg_in(ins.args[2].expr, consts
                        ))
392
393             both_known_regs = arg1_known_reg & arg2_known_reg
394             c1 = None
395             c2 = None
396             optimised_this_time = True
```

```
397
398             if both_known_regs:
399                 c1 = consts[ins.args[1].expr]
400                 c2 = consts[ins.args[2].expr]
401             elif none_reg:
402                 c1 = ins.args[1]
403                 c2 = ins.args[2]
404             elif arg1_known_reg & (not arg2_is_reg):
405                 c1 = consts[ins.args[1].expr]
406                 c2 = ins.args[2]
407             elif arg2_known_reg & (not arg1_is_reg):
408                 c1 = ins.args[1]
409                 c2 = consts[ins.args[2].expr]
410             else:
411                 optimised_this_time = False
412
413             if optimised_this_time:
414                 if isinstance(c1,str):
415                     c1 = int(c1,0)
416                 if isinstance(c2,str):
417                     c2 = int(c2,0)
418                 fold = c1 + c2
419                 self.logger.debug(' being replaced... '+str(self.
                        peephole[i]))
420                 self.peephole[i] = Instr('li',[ins.args[0], hex(fold)
                        ])
421                 self.logger.debug('new instruction: '+str(self.
                        peephole[i]))
422                 consts[self.peephole[i].args[0].expr] = self.peephole[
                        i].args[1]
423                 optimised = True
424                 self.logger.debug('constant folded')
425                 self.stats['cf'] += 1
426
427         return optimised
428
429
430     def suboptimisation(self):
431         """ tries to constant-fold if 2+ compile time constants
                present """
432
433         self.logger = logging.getLogger('ConstantFold')
434         optimised = False
435
436         for i, ins in enumerate(self.peephole):
437             if isinstance(ins,Instr):
438                 consts = self.find_constants(i)
439                 if len(consts) > 1:
440                     optimised = self.addu(i, ins, optimised, consts)
441         return optimised
```

A.4. block_opt_lab.py

```
"""
File:          block_opt_lab.py
Course:        Compilerbouw 2011
Author:        Joris Stork, Lucas Swartsenburg, Jeroen Zuiddam


Description:
    Block optimisers not ready for prime-time.

"""

from cfg import BasicBlock
import ir
from ir import Instr
import math
from peephole import Peephole, Peeper
from uic import copy_prop_targets, copy_prop_unsafe, assign_to
import logging


class AlgebraicTransformations(BlockOptimiser):
    """ contains various algebraic transformation optimisations  """


    def divd_to_sra(self, i, ins, opt, consts):
        """
        shift-right arithmetic is faster than division...
        Needs fixing: div.d only ever uses $fn (floating point)
            registers. li
        instructions (used to find constants) only use $n registers.
            Note that
        manual states that div instructions write to hi and lo
            registers,
        whereas benchmark code makes use of an fd register.

       """
        optimised = opt
        if ins.instr == 'div.d':
            if (ins.args[1].expr in consts) & (ins.args[2].expr in
                consts):
                n = math.log(consts[ins.args[2].expr], 2)
                if n % 1 == 0:
                    newins = Instr('sra', [ins.args[0], consts[ins.
                        args[1].expr], n])
                    self.peephole[i] = newins
                    self.logger.info('algebraic transformed (div->sra)
                        ')
                    optimised = True
        return optimised


    def suboptimisation(self):
        """    """

```

```
49          self.logger = logging.getLogger('AlgebraicTransformations')
50          optimised = False
51          for i, ins in enumerate(self.peephole):
52              if isinstance(ins,Instr):
53                  consts = self.find_constants(i)
54                  #optimised=self.divd_to_sra(i, ins, optimised, consts)
                        #cf shelf
55                  #TODO: any other algebraic opts? (sla not available)
56          return optimised
57
58
59
60  class CommonSubexpressions(BlockOptimiser):
61      """ Block optimisation: duplicate subexpressions -> variables"""
62
63      def suboptimisation(self):
64          """ If duplicate subexpression found within peephole,  """
65
66          optimised = False
67
68          return optimised
69
70
71
72  class TempVarRename(BlockOptimiser):
73      """    """
74
75      def suboptimisation(self):
76          """    """
77
78          optimised = False
79
80          return optimised
81
82
83
84  class ExchangeIndependentStatements(BlockOptimiser):
85      """    """
86
87      def suboptimisation(self):
88          """    """
89
90          optimised = False
91
92          return optimised
93
94
95
96  class MachineDependentTransformations(BlockOptimiser):
97      """    """
98
99      def suboptimisation(self):
100         """    """
101
```

```
102         optimised = False
103 
104         return optimised
```

A.5. cfg.py

```
1  #!/usr/bin/env python
2  
3  #
4  # cfg.py
5  #
6  """
7  
8  Control Flow Graph and Basic Block
9  
10 """
11 
12 from ir import Instr, Label, control_instructions
13 
14 
15 
16 class BasicBlock(object):
17     """
18     Basic Block
19 
20     """
21 
22     def __init__(self, instr = None, name = None):
23         self.genset = {}
24         self.killset = {}
25         self.inset = {}
26         self.outset = {}
27 
28         self.killown = {}
29         self.reach = {}
30         self.liveout = []
31         self.livein = []
32         self.live_in_node = []
33         self.live_in_node_reg = []
34 
35         if not instr:
36             self.instructions = []
37         else:
38             self.instructions = instr
39         if name:
40             self.name = name
41         else:
42             self.name = "Nameless"
43         self.next = []
44 
45 
46     def __getitem__(self, index):
47         return self.instructions[index]
```

```
48
49
50      def __setitem__(self, index, value):
51          self.instructions[index] = value
52
53
54      def __len__(self):
55          "Return number of instructions in this block."
56          return len(self.instructions)
57
58      def append(self, value):
59          "Append instruction to block."
60          self.instructions.append(value)
61
62      def __str__(self):
63          return str(self.instructions)
64
65      def print_block(self):
66          print "\nBlockname: ", self.name
67          for ins in self.instructions:
68              print ins
69
70
71
72  class CFG(object):
73      """
74      Control Flow Graph
75      """
76
77      def __init__(self, flat_ir):
78          """
79          Sets the blocks and edges, by executing the load_flat function
                . The
80          load_flat function gets of (flat) list of instructions.
81          After the edges and blocks are found a png image is generated
                that
82          displays the cfg.
83          """
84          #The edges of the graph are tuples, where the first value is
                the
85          #name of the source node and the second value is the name of
                the
86          #destination node.
87          self.edges = []
88          self.blocks = []
89          self.load_flat(flat_ir)
90
91
92
93      def load_flat(self, flat_ir):
94          """
95          Load list of expression objects in Control Flow Graph.
96          """
97          j = 0
```

```
98          self.blocks.append(BasicBlock(name=str(j)))
99          j += 1
100         brenches = [
101             'beq',   #- branch == 0
102             'bne',   #- branch != 0
103             'blez',  #- branch <= 0
104             'bgtz',  #- branch > 0
105             'bltz',  #- branch < 0
106             'bgez',  #- branch >= 0
107             'bct',   #- branch FCC TRUE
108             'bcf',    #- branch FCC FALSE
109             'bc1f',   #- Branch on floating point compare false.
110             'bc1t',   #- Branch on floating point compare true.
111         ]
112
113         for i,expr in enumerate(flat_ir):
114             if (type(expr) == Instr
115                 and expr.instr in control_instructions
116                 and not expr.instr in ['jal', 'jalr']):
117                 self.blocks[-1].append(expr)
118
119                 # Add edge from this block to the destination of the
                        jump
120                 try:
121                     self.edges.append((self.blocks[-1].name, expr.
                            jump_dest()))
122                 except exception:
123                     print "Adding edge failed because of jump"
124
125                 # If previous instruction was a brench, add a edge
                        from the
126                 # previous block to the new block.
127                 if (i > 0 and type(flat_ir[i-1]) == Instr
128                     and flat_ir[i-1].instr in brenches
129                     and len(self.blocks) > 1):
130                     self.edges.append((self.blocks[-2].name, self.
                            blocks[-1].name))
131
132                 self.blocks.append(BasicBlock(name=str(j)))
133                 j += 1
134             elif type(expr) == Label:
135                 # If previous instruction wasn't a jump, add an edge
                        between
136                 # the previous block and the new one.
137                 if (i > 0
138                     and flat_ir[i-1] not in ['j', 'jr']
139                     and len(self.blocks) > 0):
140                     self.edges.append((self.blocks[-1].name, expr.expr
                            ))
141
142                 self.blocks.append(BasicBlock([expr], expr.expr))
143             else:
144                 # If previous instruction was a brench, add a edge
                        from the
```

```
145                 # previous block to the new block.
146                 if (i > 0 and type(flat_ir[i-1]) == Instr
147                     and flat_ir[i-1].instr in brenches
148                     and len(self.blocks) > 1):
149                     self.edges.append((self.blocks[-2].name, self.
                            blocks[-1].name))
150 
151                 self.blocks[-1].append(expr)
152 
153     def cfg_to_diagram(self, name="CFG.png"):
154         """
155         Generates a diagram using the edges that were found when
                load_flat
156         was executed. The result is saved in a png image file.
157         """
158         import pygraphviz as pgv
159         A = pgv.AGraph(directed=True)
160         for edge in self.edges:
161             A.add_edge(edge[0], edge[1])
162         for block in self.blocks:
163             if len(self.get_out_edges(block)) + len(self.get_in_edges(
                    block)) == 0:
164                 A.add_node(block.name)
165         A.layout()
166         A.draw(name)
167 
168     def print_cfg(self):
169         """
170         Prints the name of each block, its instructions and its edges.
171 
172         NOTE: This function is not to be used to create assembly
                output.
173         """
174         for block in self.blocks:
175             block.print_block()
176             print "Edges: (out)", self.get_out_edges(block),\
177             " (in)", self.get_out_edges(block)
178 
179     def remove_block(self, name):
180         """
181         Removes the block corresponding to the name and all its edges
182         """
183         removed = False
184         for i,block in enumerate(self.blocks):
185             if block.name == name:
186                 del self.blocks[i]
187                 removed = True
188                 break
189 
190         rm = []
191         for edge in self.edges:
192             if name == edge[0] or name == edge[1]:
193                 rm.append(edge)
194         for r in rm:
```

```
195             self.edges.remove(r)
196         return removed
197 
198     def get_out_edges(self, block):
199         """
200         Returns all (a list) edges that come out of the given block.
201         You can pass the name of a block or the
202         """
203         _out = []
204         if type(block) == BasicBlock:
205             block = block.name
206         for edge in self.edges:
207             if (edge[0] == block):
208                 _out.append(edge)
209         return _out
210 
211     def get_in_edges(self, block):
212         """
213         Returns all (a list) edges that go into the given block.
214         """
215         _in = []
216         if type(block) == BasicBlock:
217             block = block.name
218         for edge in self.edges:
219             if (edge[1] == block):
220                 _in.append(edge)
221         return _in
222 
223     def get_blockname(self, block):
224         """
225         Returns the name of a given block
226         """
227         return block.name
228 
229     def get_block(self, name):
230         """
231         Given a name, the corresponding block object is returned.
232         """
233         for block in self.blocks:
234             if block.name == name:
235                 return block
236         return None
237 
238     def cfg_to_flat(self):
239         """
240         Convert Control Flow Graph to list of expression objects.
241         """
242 
243         return sum((list(block) for block in self.blocks), [])
244 
245 def main():
246     # test code
247     from asmyacc import parser
248 
```

```
249     flat = []
250     for line in open('../benchmarks/pi.s', 'r').readlines():
251         if not line.strip(): continue
252         flat.append(parser.parse(line))
253     c = CFG(flat)
254     return c
255 if __name__ == '__main__':
256     main()
257     pass
```

A.6. dataflow.py

```
1  from cfg import CFG, BasicBlock
2  from ir import *
3  import parse_instr
4
5  class Dataflow(object):
6      def __init__(self, graph):
7          self.graph = graph
8          self.iterations = 0
9          self.set_ins_names()
10         self.create_sets()
11
12
13     def set_ins_names(self):
14         """
15         Creates names for instructions, so that they can easily be
               identified
16         during optimisation. Using indexes would cause problems.
17         """
18         for block in self.graph.blocks:
19             for i,instr in enumerate(block.instructions):
20                 instr.id = block.name + "_" + str(i)
21
22
23     def create_sets(self):
24         """
25         Calls all subroutines to create the different sets in the
               right order.
26         """
27         self.create_gen()
28         self.create_kill()
29         self.create_inout()
30
31     def create_gen(self):
32         """
33         Determines which instructions are in the gen set of each block
34         """
35         for block in self.graph.blocks:
36             # {"",[]}
37             gen = {}
38             killown = {}
39             for i,instr in enumerate(block.instructions):
```

```
40                  if type(instr) == Instr:
41                      kill = []
42                      if len(instr.gen) > 0:
43                          for reg in instr.gen:
44                              kill += self.check_regs(gen, reg)
45                          for key in kill:
46                              if key not in killown:
47                                  killown[key] = gen[key]
48                              if key in gen:
49                                  del gen[key]
50                          gen[instr.id] = instr.gen
51              block.genset = gen
52              block.killown = killown
53
54      def create_kill(self):
55          """
56          Determines which instructions are in the kill set of each
                block
57          """
58          for block in self.graph.blocks:
59              self.get_reach(block)
60
61          #For each blokc in the graph
62          for block in self.graph.blocks:
63              #Check all nodes that can be reached
64              for targetname in block.reach:
65                  target = self.graph.get_block(targetname)
66                  if target:
67                      kills = []
68
69                      #For all instruction in the target block
70                      for ins in target.instructions:
71                          if type(ins)==Instr:
72                              #Check if they write to the same
73                              #registers as a instruction in the
74                              #origional block
75                              for g in ins.gen:
76                                  kills += self.check_regs(block.genset,
                                        g)
77                                  kills += self.check_regs(block.killown
                                        , g)
78
79
80                      #For all instructions that are overwritten (killed
                        ) in the
81                      #origional block, find the corresponding registers
                         and add
82                      #the killed instruction to the killset of the
                        target block.
83                      for kill in kills:
84                          if kill not in target.killset:
85                              if kill in block.genset:
86                                  target.killset[kill] = block.genset[
                                        kill]
```

```
87                              elif kill in block.killset:
88                                  target.killset[kill] = block.killown[
                                        kill]
89      def create_inout(self):
90          """
91          Determines the in and out sets for blocks using the Iterative
                algorithm
92          for reaching definitions.
93          """
94          for block in self.graph.blocks:
95              for key in block.genset:
96                  block.outset[key] = block.genset[key]
97
98          change = True
99          while change:
100             self.iterations += 1
101             change = False
102             for i,block in enumerate(self.graph.blocks):
103
104                 #Set inset
105                 oldin = dict(block.inset)
106                 block.inset = {}
107                 pred = self.graph.get_in_edges(block)
108                 for pre in pred:
109                     p = self.graph.get_block(pre[0])
110
111                     for key in p.outset:
112                         block.inset[key] = p.outset[key]
113                 #Set outset
114                 oldout = dict(block.outset)
115                 block.outset = dict(block.genset)
116                 for key in block.inset:
117                     if key not in block.killset:
118                         block.outset[key] = block.inset[key]
119                 #test for change
120                 #if i == 19:
121                 #    self.graph.blocks[i].print_block()
122                 #    print oldout, "\n"
123                 #    print block.outset, "\n"
124                 #    print oldin, "\n"
125                 #    print block.inset, "\n"
126                 #    print block.killset
127                 if not (self.compare_dict(oldout, block.outset) and
                        self.compare_dict(block.inset, oldin)):
128                     change = True
129
130     def compare_dict(self,a,b):
131         if len(a) != len(b):
132             return False
133         for key in a:
134             if key not in b:
135                 return False
136         return True
137
```

```
138     def print_sets(self):
139         """
140         If all sets are determined, they can be printed in a overview
                for
141         analysis by hand.
142         """
143         for block in self.graph.blocks:
144             print "
                    ----------------------------------------------------------
                    "
145             block.print_block()
146             print "\nGenset:"
147             for i in block.genset:
148                 print str(i) + ": " + str(block.genset[i])
149             print "\nKillownset:"
150             for i in block.killown:
151                 print str(i) + ": " + str(block.killown[i])
152             print "\nKillset:"
153             for i in block.killset:
154                 print str(i) + ": " + str(block.killset[i])
155             print "\nInset:"
156             for i in block.inset:
157                 print str(i) + ": " + str(block.inset[i])
158             print "\nOutset:"
159             for i in block.outset:
160                 print str(i) + ": " + str(block.outset[i])
161         print "Iterations: ", self.iterations
162
163
164     def remove_duplicates(self, l):
165         """
166         Removes all duplicate values in a list (with hashable values)
167         """
168         d = {}
169         for x in l:
170             d[x] = 1
171         l = list(d.keys())
172         return l
173
174
175     def check_regs(self, dic, reg):
176         """
177         Checks if the register that is given, is in the gen dict of
178         a block. If
179         so, the key (instruction name) is added to a list and
180         returned.
181         """
182         kill = []
183         if type(reg) == Register:
184             reg = reg.expr
185         for key in dic:
186             for dicreg in dic[key]:
187                 if type(dicreg) == Register:
188                     if reg == dicreg.expr:
```

```
189                         kill.append(key)
190                 elif type(dicreg) == str:
191                     if reg == dicreg:
192                         kill.append(key)
193         return kill
194 
195     def get_reach(self,block):
196         """
197         Returns a list that contains all names of the blocks that can
198         be reached from the given block.
199 
200         Example: Graph = b1 -> b2 -> b3 Reach(b1) = [b2,b3]
201         """
202         lengthold = 0
203         reach = [block.name]
204         lengthnew = 1
205         while lengthold != lengthnew:
206             lengthold = len(reach)
207             for bl in reach:
208                 for edge in self.graph.get_out_edges(str(bl)):
209                     if edge[1] not in reach:
210                         reach.append(edge[1])
211 
212             lengthnew = len(reach)
213         reach.remove(block.name)
214         block.reach = reach
215         #print reach
216         return reach
217 
218 
219 def main():
220     # test code
221     from asmyacc import parser
222 
223     flat = []
224     for line in open('../benchmarks/pi.s', 'r').readlines():
225         if not line.strip(): continue
226         flat.append(parser.parse(line))
227     flat = parse_instr.parse(flat)
228     c = CFG(flat)
229     d = Dataflow(c)
230     d.print_sets()
231     #d.get_reach(c.get_block("$L7"))
232 
233 
234     return c
235 if __name__ == '__main__':
236     main()
237     pass
```

A.7. flat.py

```
1 #!/usr/bin/env python
```

```
2
3  from asmyacc import parser
4  import sys
5  from ir import *
6  import re
7
8  #test with: optimize([Instr("beq", ["2"]), Label("1"),Label("2"),Instr
       ("j",["3"]),Instr("beq", ["2"]),Instr("beq", ["2"]),Instr("beq",
       ["2"]),Instr("beq", ["2"]),Instr("beq", ["2"]),Label("3")])
9
10
11 def optimize_jump(instruction_list):
12     """
13     This function scans the instruction list to make improvements by
           adjusting
14     and removing jumps and labels.
15     Situations:     Label, Jump
16                     Jump, NOT Label
17                     label, label
18     First we need to scan the list for jump_not_label. After this has
19     been done, we can start on label_label and label_jump imporvements
           .
20     """
21     #instruction_list = remove_comments(instruction_list)
22     clean = False
23     while(not clean):
24         clean = True
25         for i,instruction in enumerate(instruction_list):
26             clean = jump_not_label(instruction, i, instruction_list)
27             if not clean:
28                 break
29     clean = False
30     while(not clean):
31         clean = True
32         for i,instruction in enumerate(instruction_list):
33             clean = label_label(instruction, i, instruction_list)
34             if not clean:
35                 break
36             clean = label_jump(instruction, i, instruction_list)
37             if not clean:
38                 break
39
40     #Needs liveness analysis
41     #clean = False
42     #while(not clean):
43     #    clean = True
44     #    for i,instruction in enumerate(instruction_list):
45     #        clean = cvt(instruction, i, instruction_list)
46     #        if not clean:
47     #            break
48
49     return instruction_list
50
51 def remove_comments(il):
```

```
52      """
53      Not used
54      """
55      new = []
56      for ins in il:
57          if type(ins) != Comment:
58              new.append(ins)
59      return new
60
61  def cvt(ins, i, il):
62      """
63      Optimises redundant conversion.
64      """
65      clean = True
66      if i + 1 < len(il) -1 and \
67      type(ins) == Instr and type(il[i+1])== Instr and 'cvt' in ins.
            instr and 'cvt' in il[i+1].instr:
68          if ins.args[0].expr == il[i + 1].args[1].expr:
69
70              l = re.compile("^cvt\.([a-z])\.([a-z])$")
71              van = re.match(l,ins.instr).group(2)
72              naar = re.match(l,ins.instr).group(1)
73              new = Instr("cvt."+naar+"."+van,[ins.args[1],il[i+1].args
                    [0]])
74              il[i + 1] = new
75              del il[i]
76              clean = False
77
78      return clean
79
80
81  def jump_not_label(ins, i, il):
82      """
83      Every instruction that is between a jump and a label will never be
84      executed. This function removes all the instructions that never
            will be
85      executed.
86      """
87      clean = True
88      if (type(ins) == Instr and
89          ins.instr.lower() == 'j'):
90              while (i < len(il) - 1 and
91                     type(il[i + 1]) != Label and type(il[i + 1]) != Raw
                           ):
92                  del il[i + 1]
93                  if clean:
94                      clean = False
95      return clean
96
97  def label_label(label, i , il):
98      """
99      This function searches for labels, and removes all labels that
            come
```

```
100     directly after it. The instructions that point to one of the
            removed
101     labels are adjusted so that they point to the first label, that
            isn't
102     removed.
103     """
104     clean = True
105     if (type(label) == Label):
106         if(i < len(il) - 1 and
107                 type(il[i + 1]) == Label) and not\
108                 (len(il[i + 1].expr) > 2 and
109                 il[i + 1].expr[0:2]=="__"):
110             replace_label(label.expr, il[i + 1].expr, il)
111             del il[i + 1]
112             clean = False
113     return clean
114
115 def label_jump(ins, i, il):
116     """
117     If a jump comes directly after a label, the label as well as the
            jump are
118     unneccesery. All instructions point to the label are adjusted to
            point to
119     the label the jump was pointing at.
120     """
121     clean = True
122     if i < len(il) - 1:
123         next_i = il[i + 1]
124         if (type(ins) == Label and
125             type(il[i + 1]) == Instr and
126             il[i + 1].instr == 'j'):
127                 replace_label(il[i + 1].args[0], ins.expr, il)
128                 del il[i:i + 2]
129                 clean = False
130     return clean
131
132
133 def replace_label(new_label, old_label, il):
134     """
135     This helper function adjusts all instructions that point to the
            old_label
136     so that they point to the new label.
137
138     IMPORTANT: Make sure new_label and old_label are strings. For
            instructions
139     the correct string is Instr.args[i] for some i and for Labels the
            correct
140     string is Label.expr .
141     """
142     for ins in il:
143         if (type(ins) == Instr):
144             while (old_label in ins.args):
145                 ins.args[ins.args.index(old_label)] = new_label
146
```

```
147 def optimize_brench(instruction_list):
148     instruction_list = brench_jump(instruction_list)
149     instruction_list = useless_brench(instruction_list)
150     return instruction_list
151 
152 def useless_brench(il):
153     """
154     If a jump or brench points to a label that comes directly after it
            , it can be
155     removed.
156     """
157     clean = False
158     while(not clean):
159         clean = True
160         for i,ins in enumerate(il):
161             if type(ins) == Instr and i < len(il) - 1 and \
162                type(il[i + 1]) == Label and \
163                ins.instr in control_instructions:
164                 if ins.jump_dest() == il[i + 1].expr:
165                     del il[i]
166                     clean = False
167                     break
168 
169     return il
170 
171 
172 def brench_jump(il):
173     """
174     This function removes a jump when we find a branch - jump - label
175     combination where the branch points to the label. We can optimise
            this
176     by negating the brench, making it point to the jump adres and
            removing the
177     label.
178     """
179     negation = {'beq':'bne','bne':'beq','blez':'bgtz','bgtz':
180         'blez','bltz':'bgez','bgez':'bltz','bc1t':'bc1f','bc1f':'bc1t'
                }
181     clean = False
182     while(not clean):
183         clean = True
184         for i,ins in enumerate(il):
185             if type(ins) == Instr and ins.instr in negation \
186                and i + 2 < len(il) - 1 and type(il[i+1]) == Instr \
187                and type(il[i+2]) == Label and \
188                ins.jump_dest() == il[i+2].expr and il[i+1] == 'j':
189                 il[i].instr = negation[il[i].instr]
190                 il[i].args[-1] = il[i+1].jump_dest()
191                 del il[i+1]
192                 clean = False
193                 break
194 
195 
196     return il
```

```
197
198
199 def optimise(instruction_list):
200     """
201     This functions calls a number of flat optimalization routines and
202     returns the improved list.
203     """
204     old_len = 1
205     new_len = 0
206     while old_len != new_len:
207         old_len = len(instruction_list)
208         instruction_list = optimize_jump(instruction_list)
209         instruction_list = optimize_brench(instruction_list)
210         new_len = len(instruction_list)
211     return instruction_list
212
213 if __name__ == '__main__':
214     if len(sys.argv) > 1:
215         raise_on_error = True
216         instruction_list = []
217         for line in open(sys.argv[1], 'r').readlines():
218            if not line.strip(): continue
219            instruction_list.append(parser.parse(line))
220
221         instruction_list = optimise(instruction_list)
222         for ins in instruction_list:
223             if type(ins) == Label:
224                 print ins
225             elif type(ins) != Comment:
226                 print "\t" + str(ins)
```

A.8. ir.py

```
1  #
2  # IR objects
3  #
4  # Reg $31 is a return register
5  control_instructions = [
6      'j',      #- jump.
                                                                    Used:
           j {label}. Needs: [], Gen: []
7      'jal',    #- jump and link.
                                                          Used: jal {
          function}. Needs: [], Gen: [$31]
8      'jr',     #- jump register.
                                                          Used: jr $x.
          Needs: [$x], Gen: []
9      'jalr',   #- jump and link register.
                                                  Not used
10     'beq',    #- branch == 0.
                                                            Used: beq $a
           $b {label}. Needs: [$a,$b], Gen: []
```

```
11      'bne',    #- branch != 0.
                                                        Used: bne $a
            $b {label}. Needs: [$a,$b], Gen: []
12      'blez',   #- branch <= 0.
                                                        Used: blez
            $a {label}. Needs: [$a], Gen: []
13      'bgtz',   #- branch > 0.
                                                         Used: bgtz
            $a {label}. Needs: [$a], Gen: []
14      'bltz',   #- branch < 0.
                                                         Used: bltz
            $a {label}. Needs: [$a], Gen: []
15      'bgez',   #- branch >= 0.
                                                        Used: bgez
            $a {label}. Needs: [$a], Gen: []
16      'bct',    #- branch FCC TRUE.
                                                       Used: bct {label
            }. Needs: [$fcc], Gen: []
17      'bcf',    #- branch FCC FALSE.
                                                      Used: bcf {label
            }. Needs: [$fcc], Gen: []
18      'bc1f',   #- Branch on floating point compare false.
                                  Used: bc1t {label}. Needs: [$fcc], Gen:
             []
19      'bc1t'    #- Branch on floating point compare true.
                                   Used: bc1t {label}. Needs: [$fcc], Gen
            : []
20      ]
21
22  loadstore_instructions = [
23      'lb',     #- load byte.
                                                            Used: lb
            $a C($b). Needs: [$b], Gen: [$a]
24      'lbu',    #- load byte unsigned.
                                                     Used: lbu $a C($b)
            . Needs: [$b], Gen: [$a]
25      'lh',     #- load half (short).
                                                      Used: lh $a C($b)
            . Needs: [$b], Gen: [$a]
26      'lhu',    #- load half (short) unsigned.
                                              Used: lhu $a C($b). Needs:
             [$b], Gen: [$a]
27      'lw',     #- load word.
                                                            Used: lw
            $a C($b). Needs: [$b], Gen: [$a]
28      'dlw',    #- load double word .
                                                      Used: dlw $a C($b
            ). Needs: [$b], Gen: [$a, $(a+1)]
29      'dmfc1',#- Doubleword move from floating point.
                                        Used: dmfc1 $a $b. Needs: [$b], Gen
            : [$a, $(a+1)]
30      'l.s',    #- load single-precision FP.
                                                Used: l.s $a C($b).
            Needs: [$b], Gen: [$a]
```

```
31      'l.d',  #- load double-precision FP.
                                                Used: l.d $a C($b).
            Needs: [$b], Gen: [$a, $(a+1)]
32      'sb',   #- store byte.
                                                        Used: sb
            $a C($b). Needs: [$a,$b], Gen: []
33      'sbu',  #- store byte unsigned.
                                                Used: sbu $a C($b).
             Needs: [$a,$b], Gen: []
34      'sh',   #- store half (short).
                                                 Used: sh $a C($b).
             Needs: [$a,$b], Gen: []
35      'shu',  #- store half (short) unsigned.
                                            Used: shu $a C($b). Needs:
            [$a,$b], Gen: []
36      'sw',   #- store word.
                                                        Used: sw
            $a C($b). Needs: [$a,$b], Gen: []
37      'dsw',  #- store double word.
                                                  Used: dsw $a C($b
            ). Needs: [$b,$a, $(a+1)], Gen: []
38      'dsz',  #- Double store zero.
                                                  Used: dsz C($a).
            Needs: [$a], Gen: []
39      's.s',  #- store single-precision FP.
                                                Used: s.s $a C($b). Needs
            : [$a,$b], Gen: []
40      's.d',  #- store double-precision FP.
                                                Used: s.d $a C($b). Needs
            : [$a, $(a+1),$b], Gen: []
41      'move', #- Move register value.
                                                Used: move $a $b.
            Used: move $a $b. Needs: [$b], Gen: [$a]
42      'mov.d',#- Move floating point value. mov.d $a $b.
                                    Used: mov.d $a $b. Needs: [$b, $(b+1)
            ], Gen: [$a, $(a+1)]
43      'mov.s',#- Move floating point value. mov.s $a $b.
                                    Used: mov.s $a $b. Needs: [$b], Gen: [
            $a]
44      'li'    #- Load immidiate.
                                                        Used: li $a {
            val}. Needs: [], Gen: [$a]
45      ]
46
47  intarithm_instructions = [
48      'add',  #- integer add.
                                                        Used: add
            $a $b $c or add $a $b val. Needs: [$b(,$c)], Gen: [$a]
49      'addi', #- integer add.
                                                        Used: addi
            $a $b val. Needs: [$b], Gen: [$a]
50      'addu', #- integer add unsigned.
                                                Used: addu $a $b $c
            or addu $a $b val. Needs: [$b(,$c)], Gen: [$a]
```

```
51      'addiu',#- integer add.
                                                        Used: addiu
            $a $b val. Needs: [$b], Gen: [$a]
52      'sub',  #- subtract.
                                                              Used:
            sub $a $b $c or sub $a $b val. Needs: [$b(,$c)], Gen: [$a]
53      'subu', #- integer subtract unsigned.
                                                    Used: subu $a $b $c or
            subu $a $b val. Needs: [$b(,$c)], Gen: [$a]
54      'mult', #- integer multiply.
                                                          Used: mult $a $b
            . Needs: [$a,$b], Gen: [$hi,$lo]
55      'multu',#- integer multiply unsigned.
                                                    Used: multu $a $b . Needs
            : [$a,$b], Gen: [$hi,$lo]
56      'div',  #- integer divide.
                                                            Used: div $a
            $b . Needs: [$a,$b], Gen: [$hi,$lo]
57      'divu', #- integer divide unsigned.
                                                      Used: div $a $b . Needs
            : [$a,$b], Gen: [$hi,$lo]
58      'and',  #- logical AND.
                                                        Used: and
            $a $b $c or and $a $b val. Needs: [$b(,$c)], Gen: [$a]
59      'andi', #- logical AND.
                                                        Used: andi
            $a $b val. Needs: [$b], Gen: [$a]
60      'or',   #- logical OR.
                                                         Used: or
            $a $b $c or or $a $b val. Needs: [$b(,$c)], Gen: [$a]
61      'ori',  #- logical OR.
                                                         Used: ori
            $a $b val. Needs: [$b], Gen: [$a]
62      'xor',  #- logical XOR.
                                                        Used: xor
            $a $b $c or xor $a $b val. Needs: [$b(,$c)], Gen: [$a]
63      'xori', #- logical XOR.
                                                        Used: xori
            $a $b val. Needs: [$b], Gen: [$a]
64      'nor',  #- logical NOR.
                                                        Used: nor
            $a $b $c. Needs: [$b,$c], Gen: [$a]
65      'sll',  #- shift left logical.
                                                    Used: sll $a $b {
            val}. Needs: [$b], Gen: [$a]
66      'sllv', #- Not used, but in user guide.
                                                  Used: sllv $a $b {val}.
            Needs: [$b], Gen: [$a]
67      'srl',  #- shift right logical.
                                                    Used: srl $a $b {
            val}. Needs: [$b], Gen: [$a]
68      'srlv', #- Not used, but in user guide.
                                                  Used: srlv $a $b {val}.
            Needs: [$b], Gen: [$a]
```

```
69      'sra',  #- shift right arithmetic.
                                                    Used: sra $a $b {val}.
            Needs: [$b], Gen: [$a]
70      'srav', #- Not used, but in user guide.
                                                  Used: srav $a $b {val}.
           Needs: [$b], Gen: [$a]
71      'slt',  #- set less than.
                                                      Used: slt $a
           $b $c or slt $a $b val. Needs: [$b(,$c)], Gen: [$a]
72      'slti', #- set less than.
                                                      Used: slti $a
            $b val. Needs: [$b], Gen: [$a]
73      'sltu', #- set less than unsigned.
                                                    Used: sltu $a $b $c or
            sltu $a $b val. Needs: [$b(,$c)], Gen: [$a]
74      'sltiu' #- set less than unsigned.
                                                    Used: sltiu $a $b val.
            Needs: [$b], Gen: [$a]
75      ]
76
77  floatarithm_instructions = [
78      'add.s',    #- single-precision (SP) add.
                                                  Used: add.s $a $b $c. Needs:
           [$b,$c], Gen: [$a]
79      'add.d',    #- double-precision (DP) add.
                                                  Used: add.d $a $b $c. Needs:
           [$b, $(b+1),$c, $(c+1)], Gen: [$a, $(a+1)]
80      'sub.s',    #- SP subtract.
                                                      Used: sub.s $a
           $b $c. Needs: [$b,$c], Gen: [$a]
81      'sub.d',    #- DP subtract.
                                                      Used: sub.d $a
           $b $c. Needs: [$b, $(b+1),$c, $(c+1)], Gen: [$a, $(a+1)]
82      'mul.s',    #- SP multiply.
                                                      Used: mul.s $a
           $b $c. Needs: [$b,$c], Gen: [$a]
83      'mul.d',    #- DP multiply.
                                                      Used: mul.d $a
           $b $c. Needs: [$b, $(b+1),$c, $(c+1)], Gen: [$a, $(a+1)]
84      'div.s',    #- SP divide.
                                                        Used: div.s
           $a $b $c. Needs: [$b,$c], Gen: [$a]
85      'div.d',    #- DP divide.
                                                        Used: div.d
           $a $b $c. Needs: [$b, $(b+1),$c, $(c+1)], Gen: [$a, $(a+1)]
86      'abs.s',    #- SP absolute value.
                                                    Used: abs.s $a $b.
           Needs: [$b], Gen: [$a]
87      'abs.d',    #- DP absolute value.
                                                    Used: abs.d $a $b.
           Needs: [$b, $(b+1)], Gen: [$a, $(a+1)]
88      'neg.s',    #- SP negation.
                                                      Used: neg.s $a
           $b. Needs: [$b], Gen: [$a]
```

```
89      'neg.d',    #- DP negation.
                                                    Used: neg.d $a
           $b. Needs: [$b, $(b+1)], Gen: [$a, $(a+1)]
90      'sqrt.s',   #- SP square root.
                                                    Used: sqrt.s $a $b
           . Needs: [$b], Gen: [$a]
91      'sqrt.d',   #- DP square root.
                                                    Used: sqrt.d $a $b
           . Needs: [$b, $(b+1)], Gen: [$a, $(a+1)]
92      'cvt',      #- int., single, double conversion.
                                      Used: cvt $a $b. Needs: [$b], Gen:
           [$a]
93      'cvt.d.w',  #- Integer to float.
                                                    Used: cvt.d.w $a $b.
            Needs: [$b], Gen: [$a, $(a+1)]
94      'cvt.s.d',  #- Float: double to single precision.
                                       Used: cvt.s.d $a $b. Needs: [$b, $(b
           +1)], Gen: [$a]
95      'cvt.d.s',  #- Float: single to double precision.
                                       Used: cvt.d.s $a $b. Needs: [$b], Gen
           : [$a, $(a+1)]
96      'cvt.s.w',  #- Float: integer to single precision.
                                     Used: cvt.s.w $a $b. Needs: [$b], Gen:
            [$a]
97      'cvt.w.s',  #- Float: single precision to integer.
                                     Used: cvt.w.s $a $b. Needs: [$b], Gen:
            [$a]
98      'cvt.w.d',  #- Float: double precision to integer.
                                     Used: cvt.w.d $a $b. Needs: [$b, $(b
           +1)], Gen: [$a]
99      'c.eq.s',   #- SP compare.
                                                    Used: c.eq.s
           $a $b. Needs: [$a, $(a+1),$b, $(b+1)], Gen: [$fcc]
100     'c.eq.d',   #- DP compare.
                                                    Used: c.eq.d
           $a $b. Needs: [$a,$b], Gen: [$fcc]
101     'c.lt.s',   #- SP compare.
                                                    Used: c.lt.s
           $a $b. Needs: [$a,$b], Gen: [$fcc]
102     'c.lt.d',   #- DP compare.
                                                    Used: c.lt.d
           $a $b. Needs: [$a, $(a+1),$b, $(b+1)], Gen: [$fcc]
103     'c.le.s',   #- SP compare.
                                                    Used: c.le.s
           $a $b. Needs: [$a,$b], Gen: [$fcc]
104     'c.le.d',   #- DP compare.
                                                    Used: c.le.d
           $a $b. Needs: [$a, $(a+1),$b, $(b+1)], Gen: [$fcc]
105     'trunc.l.d',#- Convert FP.
                                                    Used: trunc.l.
           d $a $b $c. Needs: [$b, $(b+1),$c], Gen:[$a]
106     'trunc.l.s',#- Convert FP.
                                                    Used: trunc.l.
           s $a $b $c. Needs: [$b,$c], Gen: [$a]
```

```
107     'trunc.w.d',#- Convert FP.
                                                        Used: trunc.w.
            d $a $b $c. Needs: [$b, $(b+1),$c], Gen:[$a]
108     'trunc.w.s' #- Convert FP.
                                                        Used: trunc.w.
            s $a $b $c. Needs: [$b,$c], Gen: [$a]
109 ]
110
111 misc_instructions = [
112     'nop',      #- no operation.
                                                      Used: nop
113     'syscall',  #- system call,
                                                       Used: syscall
114     'break',    #- declare program error.
                                                 Used: break
115
116     #extra
117     'mflo',     #- Move from lo.
                                                      Used: mflo $a.
            Needs: [$lo], Gen: [$a]
118     'mtlo',     #- Move to lo.
                                                        Used: mtlo $a.
             Needs: [$a], Gen: [$lo]
119     'mfhi',     #- Move from hi.
                                                      Used: mfhi $a.
            Needs: [$hi], Gen: [$a]
120     'mthi',     #- Move to hi.
                                                        Used: mthi $a.
             Needs: [$a], Gen: [$hi]
121     'mtc1',     #- From int reg to float reg.
                                              Used: mtc1 $a $b. Needs: [$a
            ], Gen: [$b]
122     'mfc1',     #- From float reg to int reg.
                                              Used: mfc1 $a $b. Needs: [$b
            ], Gen: [$a]
123     'la',       #- Load address.
                                                      Used: la $a {
            label}. Needs: [], Gen: [$a]
124     'lui'       #- Load upper immidiate:
                                               Used: lui $a {val}.
            Needs: [], Gen: [$a]
125 ]
126
127
128
129 registers = [
130     r'\$zero' #zero-valued source/sink
131     r'\$at' #reserved by assembler
132     r'\$v[0-1]', #fn return result regs
133     r'\$a[0-3]', #fn argument value regs
134     r'\$t[0-7]', #temp regs, caller saved
135     r'\$s[0-7]', #saved regs, callee saved
136     r'\$t[8-9]', #temp regs, caller saved
137     r'\$k[0-1]', #reserved by OS
```

```
138     r'\$gp', #global pointer
139     r'\$sp', #stack pointer
140     r'\$s8', #saved regs, callee saved
141     r'\$ra', #return address reg
142     r'\$hi', #high result register
143     r'\$lo', #low result register
144     r'\$f([1-2][0-9]|3[0-1]|[0-9])', #floating point registers $f0 -
            $f31
145     r'\$fcc', #floating point condition code
146     r'\$([1-2][0-9]|3[0-1]|[0-9])', #extra registers $0-$31 (not in
            spec! :s)
147     r'\$fp' #Frame pointer
148     ]
149
150
151 class Expr(object):
152     obtype = 'expr'
153     def __init__(self, expr):
154         self.expr = expr
155
156     def __repr__(self):
157         return '<Expr %r>' % (self.expr,)
158
159     def __str__(self):
160         return self.expr
161
162     def pattern(self):
163         return self.obtype
164
165 class Instr(Expr):
166     obtype = 'instr'
167
168     def __init__(self, instr, args):
169         self.id = None
170         self.instr = instr
171         self.args = args
172         self.gen = []
173         self.need = []
174         self.c = None
175         self.label = None
176         self.ival = None
177
178     def jump_dest(self):
179         if self.instr in control_instructions:
180             return self.args[-1]
181         else:
182             raise Exception('this is not a jump/branch: ', self.instr)
183
184     def __repr__(self):
185         return '<Instr %r %r>' % (self.instr, self.args)
186
187     def __str__(self):
188         return '\t%s\t' % self.instr + ','.join((str(arg) for arg in
                self.args))
```

```
189
190     def pattern(self):
191         return '%s\t' % self.instr + ','.join((arg.pattern() for arg
                in self.args))
192
193 class Register(Expr):
194     obtype = 'reg'
195     def __repr__(self):
196         return '<Register %r>' % self.expr
197
198 class Raw(Expr):
199     obtype = 'raw'
200     def __repr__(self):
201         return '<Raw %r>' % self.expr
202     def __str__(self):
203         return '\t%s' % self.expr
204
205
206 class Comment(Expr):
207     obtype = 'comment'
208     def __repr__(self):
209         return '<Comment %r>' % self.expr
210     def __str__(self):
211         return '\t%s' % self.expr
212
213
214 class Label(Expr):
215     obtype = 'label'
216     def __repr__(self):
217         return '<Label %r>' % self.expr
218
219     def __str__(self):
220         return '%s:' % self.expr
```

A.9. liveness.py

```
1  from cfg import CFG, BasicBlock
2  from ir import *
3  import parse_instr
4  from dataflow import Dataflow
5
6  class Liveness(object):
7      def __init__(self, graph, verbosity=2):
8          self.graph = graph
9          self.verbosity = verbosity
10         #self.dataflow = False
11         #self.dataflow = dataflow_done()
12         #if self.dataflow:
13         #change = True
14         #while change:
15         #    self.analyse()
16         #    change = self.optimise()
17         #    print change
```

```
18          #self.analyse()
19
20      def dataflow_done(self):
21          result = False
22          for block in self.graph.blocks:
23              if len(block.inset) > 0:
24                  result = True
25                  break
26          return result
27
28      def analyse(self):
29          atnode = []
30          blocknames = []
31          for block in self.graph.blocks:
32              blocknames.append(block.name)
33              #block.print_block()
34              for i in xrange(1,len(block.instructions)):
35                  if type(block.instructions[-i])==Instr:
36                      #print "1, ", block.instructions[-i]
37                      needs = []
38                      for reg in block.instructions[-i].need:
39                          if type(reg)==Register:
40                              needs.append(reg)
41                      #print "2, ", needs
42                      for j in xrange(i + 1, len(block.instructions) +
                            1):
43
44                          if type(block.instructions[-j])==Instr:
45                              #print "3, ", block.instructions[-j]
46                              for n in needs:
47                                  for g in block.instructions[-j].gen:
48                                      #print "4, ", repr(g)
49                                      if type(g)==Register and n.expr ==
                                            g.expr:
50                                          #print "5, ", n, g, block.
                                                instructions[-j].id
51                                          if block.instructions[-j].id
                                                not in block.live_in_node:
52                                              block.live_in_node.append(
                                                    block.instructions[-j].
                                                    id)
53                                              block.live_in_node_reg +=
                                                    block.instructions[-j].
                                                    gen
54                                          needs.remove(n)
55                                          #Register that is needed is
                                                set in
56                                          #this last instruction. No
                                                need to
57                                          #look further.
58                                          break
59                              #Registers that are still in the needs
                                    list, are
```

```
60                          #registers that are not set in the block.
                                They
61                          #are in the in set of
62                          #the block.
63                  #print "6, ", block.live_in_node
64                  #print "7 (needs), ", needs
65                  for n in needs:
66                      if not self.reg_in_reglist(n, block.livein): #
                            n.expr not in block.livein
67                          block.livein.append(n)
68                  #print "8 (needs), ", block.livein
69
70
71
72              if len(self.graph.get_out_edges(block)) == 0 and len(self.
                    graph.get_in_edges(block)) > 0:
73                  atnode.append(block)
74                  block.liveout = []
75
76
77          for edge in self.graph.edges:
78              if edge[1] not in blocknames:
79                  for e in self.graph.get_in_edges(edge[1]):
80                      b = self.graph.get_block(e[0])
81                      if b:
82                          atnode.append(b)
83          passednodes = []
84          while len(atnode) != 0:
85
86              copyatnode = atnode[:]
87              for block in copyatnode:
88                  self.remove_block_list(block,atnode)
89                  for edge in self.graph.get_in_edges(block):
90                      #print "test"
91                      succ = self.graph.get_block(edge[0])
92                      oldout = succ.liveout[:]
93                      oldin = succ.livein[:]
94                      for inreg in block.livein:
95                          if not self.reg_in_reglist(inreg, succ.liveout
                                ):
96                              succ.liveout.append(inreg)
97                          if (not self.reg_in_reglist(inreg, succ.
                                live_in_node_reg)) and (not self.
                                reg_in_reglist(inreg, succ.livein)):
98                              succ.livein.append(inreg)
99
100                     if not (self.comp_reglist(oldin, succ.livein) and
                            self.comp_reglist(oldout, succ.liveout) and
                            succ.name in passednodes):
101                         atnode.append(succ)
102                         passednodes.append(succ.name)
103
104     def comp_reglist(self,a,b):
105         if len(a)!=len(b):
```

```
106             return False
107
108         for i in a:
109             found = False
110             for j in b:
111                 if i.expr == j.expr:
112                     found = True
113                     break
114             if not found:
115                 return False
116         return True
117
118     def remove_block_list(self,b,l):
119         for block in l:
120             if block.name == b.name:
121                 l.remove(block)
122
123     def reg_in_reglist(self,reg,reglist):
124         #print reg, reglist
125         for r in reglist:
126             if r.expr == reg.expr:
127                 return True
128         return False
129
130     def print_live(self):
131         for block in self.graph.blocks:
132             print "
                ------------------------------------------------------------
                "
133             block.print_block()
134             print "\nLive in:"
135             for i in block.livein:
136                 print i
137
138             print "\nLive out:"
139             for i in block.liveout:
140                 print i
141
142             print "\nKill instructions:"
143             for i in block.live_in_node:
144                 print i
145
146     def optimise(self):
147         change = False
148         for block in self.graph.blocks:
149             for ins in block.instructions:
150                 if type(ins) == Instr and len(ins.gen) > 0 \
151                     and ins.id not in block.live_in_node \
152                     and not (self.comp_regs(ins.gen, block.liveout)) \
153                     and ins.instr not in ['jal','jalr']:
154                     block.instructions.remove(ins)
155                     change = True
156         return change
157
```

```
158     def comp_regs(self,a,b):
159         for reg in a:
160             for r in b:
161                 if reg.expr == r.expr:
162                     return True
163         return False
164 
165 
166 def main():
167     # test code
168     from asmyacc import parser
169 
170     flat = []
171     for line in open('../benchmarks/pi.s', 'r').readlines():
172         if not line.strip(): continue
173         flat.append(parser.parse(line))
174     flat = parse_instr.parse(flat)
175     c = CFG(flat)
176     d = Dataflow(c)
177     l = Liveness(c)
178     change = True
179     while change:
180         l.analyse()
181         change = l.optimise()
182     l.print_live()
183     #1d.print_sets()
184     #d.get_reach(c.get_block("$L7"))
185 
186 
187     return c
188 if __name__ == '__main__':
189     main()
190     pass
```

A.10. optimise.py

```
1  #!/usr/bin/env python
2  
3  #
4  # optimise.py
5  #
6  
7  from optparse import OptionParser
8  import logging
9  
10 import parse_instr
11 import optimise_tree
12 import flat as flat_opt
13 
14 from asmyacc import parser
15 from ir import Raw
16 from cfg import CFG
17 from dataflow import Dataflow
```

```
18 from liveness import Liveness
19
20 import block_optimise as b_opt
21
22
23
24 def split_frames(flat):
25     """
26     Split list of expression objects in 'frames'.
27
28     """
29
30     frames = [[]]
31     for expr in flat:
32         # A new frame starts with a .loc expression.
33         if type(expr) == Raw and expr.expr[:4] == '.loc':
34             frames.append([expr])
35         else:
36             frames[-1].append(expr)
37     return frames
38
39
40
41 class Optimiser(object):
42     """
43     Main Optimiser.
44
45     Tasks:
46      1. convert source to IR;
47      2. optimise IR;
48      3. convert IR back to source.
49
50     """
51
52     def __init__(self, lines, verbosity = 0):
53         """
54         Convert expressions to IR.
55         """
56
57         self.verbosity = verbosity
58         self.stats = {'cp':0,'cf':0,'dc':0}
59         self.logger = logging.getLogger('Optimiser')
60
61         self.logger.info('parsing assembly')
62         # Parse assembly and store in flat.
63         self.flat = []
64         for line in lines:
65             if not line.strip():
66                 # We skip empty lines. We could also tell yacc to put
                       them in a Raw.
67                 continue
68             self.flat.append(parser.parse(line))
69         self.flat = parse_instr.parse(self.flat)
70
```

```
71
72      def optimise(self):
73          """
74          Optimise the IR.
75
76          Procedure:
77           1. split in frames
78           2. convert frames to graphs
79           3. optimise graphs
80           4. convert graphs to (flat) frames
81           5. concatenate frames to get optimised program.
82
83          Store result in flat.
84
85          """
86
87          self.logger.info('optimising global control flow graph')
88
89          cfg = CFG(self.flat)
90          if self.verbosity > 2:
91              cfg.cfg_to_diagram("allinstr_graph_before.png")
92          optimise_tree.optimise(cfg)
93          if self.verbosity > 2:
94              cfg.cfg_to_diagram("allinstr_graph_after.png")
95          self.flat = cfg.cfg_to_flat()
96
97          self.logger.info('optimising flat (jumps and branches)')
98          self.flat = flat_opt.optimise(self.flat)
99
100
101
102
103
104         self.logger.info('splitting flat in frames')
105         frames = split_frames(self.flat)
106         self.logger.info('creating graph for each frame')
107         graphs = [CFG(frame) for frame in frames]
108
109         self.logger.info('optimising blocks')
110
111         for graphnr, graph in enumerate(graphs):
112             self.logger.info('graph %d of %d' % (graphnr + 1, len(
                    graphs)))
113
114             Dataflow(graph)
115             l = Liveness(graph,self.verbosity)
116
117             #self.logger.info('Performing liveness optimalisation on
                    graph')
118             #change = True
119             #while change:
120             #    l.analyse()
121             #    change = l.optimise()
122
```

```
123             for blocknr, block in enumerate(graph.blocks):
124
125                 self.logger.debug('block %d of %d' % (blocknr + 1, len
                        (graph.blocks)))
126
127                 cf_opt = b_opt.ConstantFold(block)
128                 cp_opt = b_opt.CopyPropagation(block)
129                 dc_opt = b_opt.DeadCode(block)
130
131                 done = False
132                 subopt_changes = False
133                 i = 0
134
135                 while (not done):
136                     done = True
137                     i += 1
138                     self.logger.debug('pass '+str(i))
139
140                     subopt_changes = cf_opt.optimise()
141                     if subopt_changes:self.stats['cf'] += cf_opt.stats
                            ['cf']
142                     done = done & (not subopt_changes)
143
144                     subopt_changes = cp_opt.optimise()
145                     if subopt_changes:self.stats['cp'] += cp_opt.stats
                            ['cp']
146                     done = done & (not subopt_changes)
147
148                     subopt_changes = dc_opt.optimise()
149                     if subopt_changes:self.stats['dc'] += dc_opt.stats
                            ['dc']
150                     done = done & (not subopt_changes)
151
152         self.logger.info('basic-block peephole optimisations done:')
153         self.logger.info('\t\tconstant folds: %d' % (self.stats['cf'])
                )
154         self.logger.info('\t\tcopy propagations: %d' % (self.stats['cp
                '']))
155         self.logger.info('\t\tdead code removes: %d' % (self.stats['dc
                '']))
156         self.logger.info('joining graphs to frames')
157         frames = [graph.cfg_to_flat() for graph in graphs]
158         self.logger.info('joining frames to flat')
159         self.flat = sum(frames, [])
160
161
162     def result(self):
163         """
164         Return optimised assembly.
165         """
166
167         self.logger.info('generating assembly')
168         return [str(expr)+'\n' for expr in self.flat]
169
```

```
170
171
172
173 def main():
174     """ Parse command line args, init. optimiser and run optimisations
        . """
175
176
177     usage = "usage: %prog [options] file"
178     parser = OptionParser(usage)
179     parser.add_option("-d", "--dest", dest="filename",
180                       help="save result in FILENAME, overrides -e, --
                              extension")
181     parser.add_option("-v", "--verbosity", dest="verbosity",
182             help="set verbosity (0: critical, 1: error, 2: warning, 3:
                  info, 4: debug)")
183     parser.add_option("-e", "--extension", dest="extension",
184             help="save result in source filename + EXTENSION")
185
186     (options, args) = parser.parse_args()
187     if len(args) != 1:
188         parser.error('incorrect number of arguments')
189
190     if not options.verbosity:
191         options.verbosity = 2
192
193     if not options.extension:
194         options.extension = '.opt'
195
196     logging_levels = {0: logging.CRITICAL,
197                       1: logging.ERROR,
198                       2: logging.WARNING,
199                       3: logging.INFO,
200                       4: logging.DEBUG}
201
202     logging.basicConfig(format='%(asctime)s %(levelname)-7s %(name)-14
        s %(message)s',
203                         level=4,
204                         filename='log',
205                         filemode='w'
206                         )
207     console = logging.StreamHandler()
208     console.setLevel(logging_levels[int(options.verbosity)])
209     formatter = logging.Formatter(fmt='%(asctime)s %(levelname)-7s %(
        name)-14s %(message)s')
210     console.setFormatter(formatter)
211     logging.getLogger('').addHandler(console)
212
213
214     logger = logging.getLogger('main')
215     logger.info('opening sourcefile')
216     try:
217         sourcefile = open(args[0], 'r')
218     except IOError:
```

```
219         print('error: file not found: %s' % args[0])
220         exit(1)
221     opt = Optimiser(sourcefile.readlines(), options.verbosity)
222     sourcefile.close()
223     logger.info('sourcefile closed')
224 
225     opt.optimise()
226 
227     if options.filename:
228         target_filename = options.filename
229     else:
230         target_filename = args[0] + options.extension
231 
232     targetfile = open(target_filename, 'w')
233     logging.info('writing optimised assembly to file')
234     targetfile.writelines(opt.result())
235     targetfile.close()
236 
237 if __name__ == '__main__':
238     main()
```

A.11. optimise_tree.py

```
1  import cfg
2  from ir import *
3  
4  
5  def optimise(graph):
6      """
7      Runs various optimisation schemes on the cfg
8      """
9      graph = remove_notused(graph)
10     graph = flatten(graph)
11     return graph
12 
13 def remove_notused(graph):
14     """
15     All blocks that have no incoming edges are removed.
16     Ofcourse the starting block is ignored, as well as
17     the block that only contains a raw .end instruction.
18     """
19     for (i, block) in enumerate(graph.blocks):
20         if i != 0 and len(graph.get_in_edges(block)) == 0:
21             for instr in block.instructions:
22                 if type(instr) == Instr:
23                     block.instructions.remove(instr)
24             if len(block.instructions) == 0:
25                 graph.remove_block(block)
26 
27 
28     return graph
29 
30 def flatten(graph):
```

```
31      """
32      If a block has only one incoming edge and the block on the other
            side of the edge
33      has only one outgoing edge, the blocks can be joined into one.
34
35      After this, jump optimalisation is needed.
36      """
37      clean = False
38      while(not clean):
39          clean = True
40          length = len(graph.blocks)
41          for i in xrange(length - 1):
42
43              te = graph.get_in_edges(graph.blocks[i + 1])
44
45              if len(te) == 1:
46                  et = graph.get_out_edges(te[0][0])
47                  if len(et) == 1:
48                      een = graph.get_block(te[0][0])
49                      twee = graph.blocks[i + 1]
50                      td = graph.get_out_edges(twee)
51
52                      een.instructions = een.instructions + twee.
                            instructions
53                      for (fr, to) in td:
54                          graph.edges.append((een.name, to))
55
56                      graph.remove_block(twee.name)
57                      clean = False
58                      break
59
60      return graph
61
62
63  def main():
64      c = cfg.main()
65      c.cfg_to_diagram("cfg_org.png")
66      c = optimise(c)
67      c.cfg_to_diagram("cfg_new.png")
68      return c
69  if __name__ == '__main__':
70      main()
71      pass
```

A.12. parse_instr.py

```
1  from ir import *
2  import re
3  numreg = re.compile("^(\$[a-z]?)([0-9]+)$")
4  Creg = re.compile("^([0-9]+)\((\$[A-Za-z0-9]+)\)$")
5
6  def parse(flat):
7      for ex in flat:
```

```
8           if type(ex) == Instr:
9               i = Instruction(ex)
10              ex.gen = i.gen
11              ex.need = i.need
12              ex.c = i.c
13              ex.label = i.label
14              ex.ival = i.ival
15      return flat
16
17  class Instruction(object):
18
19      def __init__(self,ins):
20          self.type = None
21          self.ins = ins
22          self.gen = []
23          self.need = []
24          self.c = None
25          self.label = None
26          self.ival = None
27          self.parse(self.ins)
28
29      def __str__(self):
30          string  = ""
31          if self.type:
32              string += self.type
33          if self.need:
34              string += " Needs: " + str(self.need)
35          if self.gen:
36              string += ". Gens: " + str(self.gen)
37
38          if self.label:
39              string += ". Label: " + str(self.label)
40          if self.ival:
41              string += ". iVal: " + str(self.ival)
42          if self.c:
43              string += ". Offset: " + str(self.c)
44          return string
45
46      def parse(self, ins):
47          """
48          Parses the instruction, to determine wich registers are used
                and set.
49          It also sets the offset used for loading an storing, used
                labels and
50          used immidiate values.
51          """
52          if type(ins)!=Instr:
53              raise Exception("You are parsing object that isn't a
                    instruction")
54          self.type = ins.instr
55          if ins.instr in control_instructions:
56              self.parse_control(ins)
57          elif ins.instr in loadstore_instructions:
58              self.parse_ls(ins)
```

```
59          elif ins.instr in intarithm_instructions :
60              self.parse_int(ins)
61          elif ins.instr in floatarithm_instructions:
62              self.parse_float(ins)
63          elif ins.instr in misc_instructions:
64              self.parse_misc(ins)
65          else:
66              self.parse_unknown(ins)
67
68      def parse_control(self,ins):
69          """
70          Parses the control type instructions.
71          """
72          if ins.instr == 'j':
73              if len(ins.args) == 1:
74                  if type(ins.args[0]) == Register:
75                      self.need = [ins.args[0]]
76                      if ins.args[0].expr == "$31":
77                          self.need += [Register("$2"), Register("$3"),
                                Register("$16"),Register("$17"),Register("
                                $18"),Register("$19"),Register("$20"),
                                Register("$21"),Register("$22"),Register("
                                $23"),Register("$f0"),Register("$f1"),
                                Register("$f2"),Register("$f3"),Register("
                                $fp"),Register("$sp"),Register("$f20"),
                                Register("$f22"),Register("$f24"),Register(
                                "$f26"),Register("$f28"),Register("$f30")]
78                  else:
79                      self.label = [ins.args[0]]
80
81
82              else:
83                  raise Exception("Invalid number of args for ins: ",
                        ins.instr)
84
85          elif ins.instr == 'jal':
86              self.gen = [Register("$31"),Register("$2"),Register("$3"),
                    Register("$f0")] #Return address and values
87              #Reg $4,5,6,7 are registers that are used for fuction
                    arguments.
88              # $f12, $f13, $f14, $f15 parameter registers zijn voor
                    functies.
89              # http://msdn.microsoft.com/en-us/library/ms253512%28v=vs
                    .90%29.aspx
90              #it is not clear which one will be used.
91              self.need = [Register("$4"),Register("$5"),Register("$6"),
                    Register("$7"),Register("$fp"), Register("$sp"),
                    Register("$f12"),Register("$f13"),Register("$f14"),
                    Register("$f15")]
92
93          elif ins.instr == 'jr':
94              if len(ins.args) == 1:
95                  self.need = [ins.args[0]]
96                  if ins.args[0].expr == "$31":
```

```
97                      self.need += [Register("$2"), Register("$3"),
                            Register("$16"),Register("$17"),Register("$18")
                            ,Register("$19"),Register("$20"),Register("$21"
                            ),Register("$22"),Register("$23"),Register("$f0
                            "),Register("$f1"),Register("$f2"),Register("
                            $f3"),Register("$fp"),Register("$sp"),Register(
                            "$f20"),Register("$f22"),Register("$f24"),
                            Register("$f26"),Register("$f28"),Register("
                            $f30")]
98              else:
99                  raise Exception("Invalid number of args for ins: ",
                        ins.instr)
100
101         elif ins.instr == 'jalr':
102             if len(ins.args) == 1:
103                 #Reg $4,5,6,7 are registers that are used for fuction
                        arguments.
104                 # $f12, $f13, $f14, $f15 parameter registers zijn voor
                         functies.
105                 # http://msdn.microsoft.com/en-us/library/ms253512%28v
                        =vs.90%29.aspx
106                 #it is not clear which one will be used.
107                 self.need = [ins.args[0],Register("$4"),Register("$5")
                        ,Register("$6"),Register("$7"),Register("$fp"),
                        Register("$sp"),Register("$f12"),Register("$f13"),
                        Register("$f14"),Register("$f15")]
108                 self.gen = [Register("$2"),Register("$3"),Register("
                        $f0")] #Return values
109             else:
110                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
111
112         elif ins.instr == 'beq':
113             if len(ins.args) == 3:
114                 self.need = [ins.args[0],ins.args[1]]
115                 self.label = Label(ins.args[2])
116             else:
117                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
118
119         elif ins.instr == 'bne':
120             if len(ins.args) == 3:
121                 self.need = [ins.args[0],ins.args[1]]
122                 self.label = Label(ins.args[2])
123             else:
124                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
125
126         elif ins.instr == 'blez':
127             if len(ins.args) == 2:
128                 self.need = [ins.args[0]]
129                 self.label = Label(ins.args[1])
130             else:
```

```
131                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
132
133         elif ins.instr == 'bgtz':
134             if len(ins.args) == 2:
135                 self.need = [ins.args[0]]
136                 self.label = Label(ins.args[1])
137             else:
138                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
139
140         elif ins.instr == 'bltz':
141             if len(ins.args) == 2:
142                 self.need = [ins.args[0]]
143                 self.label = Label(ins.args[1])
144             else:
145                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
146
147         elif ins.instr == 'bgez':
148             if len(ins.args) == 2:
149                 self.need = [ins.args[0]]
150                 self.label = Label(ins.args[1])
151             else:
152                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
153
154         elif ins.instr == 'bct':
155             if len(ins.args) == 1:
156                 self.need = [Register("$fcc")]
157                 self.label = Label(ins.args[0])
158             else:
159                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
160
161         elif ins.instr == 'bcf':
162             if len(ins.args) == 1:
163                 self.need = [Register("$fcc")]
164                 self.label = Label(ins.args[0])
165             else:
166                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
167
168         elif ins.instr == 'bc1f':
169             if len(ins.args) == 1:
170                 self.need = [Register("$fcc")]
171                 self.label = Label(ins.args[0])
172             else:
173                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
174
175         elif ins.instr == 'bc1t':
176             if len(ins.args) == 1:
177                 self.need = [Register("$fcc")]
```

```
178                 self.label = Label(ins.args[0])
179             else:
180                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
181
182
183     def parse_ls(self,ins):
184         """
185         Parses the load/store type instructions.
186         """
187         global Creg
188         if ins.instr == 'lb':
189             if len(ins.args) == 2:
190                 g = re.match(Creg, ins.args[1])
191                 if g:
192                     self.c = g.group(1)
193                     self.need = [Register(g.group(2))]
194                 else:
195                     self.need = [ins.args[1]]
196                 self.gen = [ins.args[0]]
197             else:
198                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
199
200         elif ins.instr == 'lbu':
201             if len(ins.args) == 2:
202                 g = re.match(Creg, ins.args[1])
203                 if g:
204                     self.c = g.group(1)
205                     self.need = [Register(g.group(2))]
206                 else:
207                     self.need = [ins.args[1]]
208                 self.gen = [ins.args[0]]
209             else:
210                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
211
212         elif ins.instr == 'lh':
213             if len(ins.args) == 2:
214                 g = re.match(Creg, ins.args[1])
215                 if g:
216                     self.c = g.group(1)
217                     self.need = [Register(g.group(2))]
218                 else:
219                     self.need = [ins.args[1]]
220                 self.gen = [ins.args[0]]
221             else:
222                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
223
224         elif ins.instr == 'lhu':
225             if len(ins.args) == 2:
226                 g = re.match(Creg, ins.args[1])
227                 if g:
```

```
228                     self.c = g.group(1)
229                     self.need = [Register(g.group(2))]
230                 else:
231                     self.need = [ins.args[1]]
232                 self.gen = [ins.args[0]]
233             else:
234                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
235 
236         elif ins.instr == 'lw':
237             if len(ins.args) == 2:
238                 g = re.match(Creg, ins.args[1])
239 
240                 if g:
241                     self.c = g.group(1)
242                     self.need = [Register(g.group(2))]
243                 else:
244                     self.need = [ins.args[1]]
245                 self.gen = [ins.args[0]]
246             else:
247                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
248 
249         elif ins.instr == 'dlw':
250             if len(ins.args) == 2:
251                 g = re.match(Creg, ins.args[1])
252                 if g:
253                     self.c = g.group(1)
254                     self.need = [Register(g.group(2))]
255                 else:
256                     self.need = [ins.args[1]]
257                 self.gen = self.double_reg(ins.args[0])
258             else:
259                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
260 
261         elif ins.instr == 'dmfc1':
262             if len(ins.args) == 2:
263                 self.need = [ins.args[1]]
264                 self.gen = self.double_reg(ins.args[0])
265             else:
266                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
267 
268         elif ins.instr == 'l.s':
269             if len(ins.args) == 2:
270                 g = re.match(Creg, ins.args[1])
271 
272                 if g:
273                     self.c = g.group(1)
274                     self.need = [Register(g.group(2))]
275                 else:
276                     self.need = [ins.args[1]]
277                 self.gen = [ins.args[0]]
```

```
278             else:
279                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
280
281         elif ins.instr == 'l.d':
282             if len(ins.args) == 2:
283                 g = re.match(Creg, ins.args[1])
284
285                 if g:
286                     self.c = g.group(1)
287                     self.need = [Register(g.group(2))]
288                 else:
289                     self.need = [ins.args[1]]
290                 self.gen = [ins.args[0]]
291             else:
292                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
293
294         elif ins.instr == 'sb':
295             if len(ins.args) == 2:
296                 g = re.match(Creg, ins.args[1])
297                 if g:
298                     self.c = g.group(1)
299                     self.need = [Register(g.group(2))]
300                 else:
301                     self.need = [ins.args[1]]
302                 self.need = [ins.args[0]] + self.need
303             else:
304                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
305
306         elif ins.instr == 'sbu':
307             if len(ins.args) == 2:
308                 g = re.match(Creg, ins.args[1])
309                 if g:
310                     self.c = g.group(1)
311                     self.need = [Register(g.group(2))]
312                 else:
313                     self.need = [ins.args[1]]
314                 self.need = [ins.args[0]] + self.need
315             else:
316                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
317
318         elif ins.instr == 'sh':
319             if len(ins.args) == 2:
320                 g = re.match(Creg, ins.args[1])
321                 if g:
322                     self.c = g.group(1)
323                     self.need = [Register(g.group(2))]
324                 else:
325                     self.need = [ins.args[1]]
326                 self.need = [ins.args[0]] + self.need
327             else:
```

```
328                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
329
330         elif ins.instr == 'shu':
331             if len(ins.args) == 2:
332                 g = re.match(Creg, ins.args[1])
333                 if g:
334                     self.c = g.group(1)
335                     self.need = [Register(g.group(2))]
336                 else:
337                     self.need = [ins.args[1]]
338                 self.need = [ins.args[0]] + self.need
339
340             else:
341                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
342
343         elif ins.instr == 'sw':
344             if len(ins.args) == 2:
345                 g = re.match(Creg, ins.args[1])
346                 if g:
347                     self.c = g.group(1)
348                     self.need = [Register(g.group(2))]
349                 else:
350                     self.need = [ins.args[1]]
351                 self.need = [ins.args[0]] + self.need
352             else:
353                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
354
355         elif ins.instr == 'dsw':
356             if len(ins.args) == 2:
357                 g = re.match(Creg, ins.args[1])
358                 if g:
359                     self.c = g.group(1)
360                     self.need = [Register(g.group(2))]
361                 else:
362                     self.need = [ins.args[1]]
363                 self.need = self.double_reg(ins.args[0]) + self.need
364             else:
365                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
366
367         elif ins.instr == 'dsz':
368             if len(ins.args) == 1:
369                 g = re.match(Creg, ins.args[0])
370                 if g:
371                     self.c = g.group(1)
372                     self.need = [Register(g.group(2))]
373                 else:
374                     self.need = [ins.args[0]]
375             else:
376                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
```

```
377
378         elif ins.instr == 's.s':
379             if len(ins.args) == 2:
380                 g = re.match(Creg, ins.args[1])
381                 if g:
382                     self.c = g.group(1)
383                     self.need = [Register(g.group(2))]
384                 else:
385                     self.need = [ins.args[1]]
386                 self.need = [ins.args[0]] + self.need
387             else:
388                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
389
390         elif ins.instr == 's.d':
391             if len(ins.args) == 2:
392                 g = re.match(Creg, ins.args[1])
393                 if g:
394                     self.c = g.group(1)
395                     self.need = [Register(g.group(2))]
396                 else:
397                     self.need = [ins.args[1]]
398                 self.need = self.double_reg(ins.args[0]) + self.need
399             else:
400                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
401
402         elif ins.instr == 'move':
403             if len(ins.args) == 2:
404                 self.need = [ins.args[1]]
405                 self.gen = [ins.args[0]]
406             else:
407                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
408
409         elif ins.instr == 'mov.d':
410             if len(ins.args) == 2:
411                 self.need = self.double_reg(ins.args[1])
412                 self.gen = self.double_reg(ins.args[0])
413             else:
414                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
415
416         elif ins.instr == 'mov.s':
417             if len(ins.args) == 2:
418                 self.need = [ins.args[1]]
419                 self.gen = [ins.args[0]]
420             else:
421                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
422
423         elif ins.instr == 'li':
424             if len(ins.args) == 2:
425                 self.gen = [ins.args[0]]
```

```
426                 self.ival = ins.args[1]
427             else:
428                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
429
430
431     def parse_int(self,ins):
432         """
433         Parses the int type instructions.
434         """
435         if ins.instr == 'add':
436             if len(ins.args) == 3:
437                 self.gen = [ins.args[0]]
438                 if self.is_reg(ins.args[2]):
439                     self.need = [ins.args[1], ins.args[2]]
440                 else:
441                     self.need = [ins.args[1]]
442                     self.ival = ins.args[2]
443             else:
444                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
445
446         elif ins.instr == 'addi':
447             if len(ins.args) == 3:
448                 self.gen = [ins.args[0]]
449                 self.need = [ins.args[1]]
450                 self.ival = ins.args[2]
451             else:
452                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
453
454         elif ins.instr == 'addu':
455             if len(ins.args) == 3:
456                 self.gen = [ins.args[0]]
457                 if self.is_reg(ins.args[2]):
458                     self.need = [ins.args[1], ins.args[2]]
459                 else:
460                     self.need = [ins.args[1]]
461                     self.ival = ins.args[2]
462             else:
463                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
464         elif ins.instr == 'addiu':
465             if len(ins.args) == 3:
466                 self.gen = [ins.args[0]]
467                 self.need = [ins.args[1]]
468                 self.ival = ins.args[2]
469             else:
470                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
471
472         elif ins.instr == 'sub':
473             if len(ins.args) == 3:
474                 self.gen = [ins.args[0]]
```

```
475                 if self.is_reg(ins.args[2]):
476                     self.need = [ins.args[1], ins.args[2]]
477                 else:
478                     self.need = [ins.args[1]]
479                     self.ival = ins.args[2]
480             else:
481                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
482
483         elif ins.instr == 'subu':
484             if len(ins.args) == 3:
485                 self.gen = [ins.args[0]]
486                 if self.is_reg(ins.args[2]):
487                     self.need = [ins.args[1], ins.args[2]]
488                 else:
489                     self.need = [ins.args[1]]
490                     self.ival = ins.args[2]
491             else:
492                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
493
494         elif ins.instr == 'mult':
495             if len(ins.args) == 2:
496                 self.gen = [Register("$hi"),Register("$lo")]
497                 self.need = [ins.args[0], ins.args[1]]
498             else:
499                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
500
501         elif ins.instr == 'multu':
502             if len(ins.args) == 2:
503                 self.gen = [Register("$hi"),Register("$lo")]
504                 self.need = [ins.args[0], ins.args[1]]
505             else:
506                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
507
508         elif ins.instr == 'div':
509             if len(ins.args) == 2:
510                 self.gen = [Register("$hi"),Register("$lo")]
511                 self.need = [ins.args[0], ins.args[1]]
512             elif len(ins.args) == 3:
513                 self.gen = [ins.args[0]]
514                 self.need = [ins.args[1], ins.args[2]]
515             else:
516                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
517
518         elif ins.instr == 'divu':
519             if len(ins.args) == 2:
520                 self.gen = [Register("$hi"),Register("$lo")]
521                 self.need = [ins.args[0], ins.args[1]]
522             else:
```

```
523                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
524
525         elif ins.instr == 'and':
526             if len(ins.args) == 3:
527                 self.gen = [ins.args[0]]
528                 if self.is_reg(ins.args[2]):
529                     self.need = [ins.args[1], ins.args[2]]
530                 else:
531                     self.need = [ins.args[1]]
532                     self.ival = ins.args[2]
533             else:
534                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
535
536         elif ins.instr == 'andi':
537             if len(ins.args) == 3:
538                 self.gen = [ins.args[0]]
539                 self.need = [ins.args[1]]
540                 self.ival = ins.args[2]
541             else:
542                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
543
544         elif ins.instr == 'or':
545             if len(ins.args) == 3:
546                 self.gen = [ins.args[0]]
547                 if self.is_reg(ins.args[2]):
548                     self.need = [ins.args[1], ins.args[2]]
549                 else:
550                     self.need = [ins.args[1]]
551                     self.ival = ins.args[2]
552             else:
553                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
554
555         elif ins.instr == 'ori':
556             if len(ins.args) == 3:
557                 self.gen = [ins.args[0]]
558                 self.need = [ins.args[1]]
559                 self.ival = ins.args[2]
560             else:
561                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
562
563         elif ins.instr == 'xor':
564             if len(ins.args) == 3:
565                 self.gen = [ins.args[0]]
566                 if self.is_reg(ins.args[2]):
567                     self.need = [ins.args[1], ins.args[2]]
568                 else:
569                     self.need = [ins.args[1]]
570                     self.ival = ins.args[2]
571             else:
```

```
572                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
573
574         elif ins.instr == 'xori':
575             if len(ins.args) == 3:
576                 self.gen = [ins.args[0]]
577                 self.need = [ins.args[1]]
578                 self.ival = ins.args[2]
579             else:
580                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
581
582         elif ins.instr == 'nor':
583             if len(ins.args) == 3:
584                 self.gen = [ins.args[0]]
585                 if self.is_reg(ins.args[2]):
586                     self.need = [ins.args[1], ins.args[2]]
587                 else:
588                     self.need = [ins.args[1]]
589                     self.ival = ins.args[2]
590             else:
591                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
592
593         elif ins.instr == 'sll':
594             if len(ins.args) == 3:
595                 self.gen = [ins.args[0]]
596                 self.need = [ins.args[1]]
597                 self.ival = ins.args[2]
598             else:
599                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
600
601         elif ins.instr == 'sllv':
602             if len(ins.args) == 3:
603                 self.gen = [ins.args[0]]
604                 self.need = [ins.args[1]]
605                 self.ival = ins.args[2]
606             else:
607                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
608
609         elif ins.instr == 'srl':
610             if len(ins.args) == 3:
611                 self.gen = [ins.args[0]]
612                 self.need = [ins.args[1]]
613                 self.ival = ins.args[2]
614             else:
615                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
616
617         elif ins.instr == 'srlv':
618             if len(ins.args) == 3:
619                 self.gen = [ins.args[0]]
```

```
620                 self.need = [ins.args[1]]
621                 self.ival = ins.args[2]
622             else:
623                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
624
625         elif ins.instr == 'sra':
626             if len(ins.args) == 3:
627                 self.gen = [ins.args[0]]
628                 self.need = [ins.args[1]]
629                 self.ival = ins.args[2]
630             else:
631                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
632
633         elif ins.instr == 'srav':
634             if len(ins.args) == 3:
635                 self.gen = [ins.args[0]]
636                 self.need = [ins.args[1]]
637                 self.ival = ins.args[2]
638             else:
639                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
640
641         elif ins.instr == 'slt':
642             if len(ins.args) == 3:
643                 self.gen = [ins.args[0]]
644                 if self.is_reg(ins.args[2]):
645                     self.need = [ins.args[1], ins.args[2]]
646                 else:
647                     self.need = [ins.args[1]]
648                     self.ival = ins.args[2]
649             else:
650                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
651
652         elif ins.instr == 'slti':
653             if len(ins.args) == 3:
654                 self.gen = [ins.args[0]]
655                 self.need = [ins.args[1]]
656                 self.ival = ins.args[2]
657             else:
658                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
659
660         elif ins.instr == 'sltu':
661             if len(ins.args) == 3:
662                 self.gen = [ins.args[0]]
663                 if self.is_reg(ins.args[2]):
664                     self.need = [ins.args[1], ins.args[2]]
665                 else:
666                     self.need = [ins.args[1]]
667                     self.ival = ins.args[2]
668             else:
```

```
669                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
670
671         elif ins.instr == 'sltiu':
672             if len(ins.args) == 3:
673                 self.gen = [ins.args[0]]
674                 self.need = [ins.args[1]]
675                 self.ival = ins.args[2]
676             else:
677                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
678
679
680     def parse_float(self,ins):
681         """
682         Parses the float type instructions.
683         """
684         if ins.instr == 'add.s':
685             if len(ins.args) == 3:
686                 self.gen = [ins.args[0]]
687                 self.need = [ins.args[1], ins.args[2]]
688             else:
689                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
690
691         elif ins.instr == 'add.d':
692             if len(ins.args) == 3:
693                 self.gen = self.double_reg(ins.args[0])
694                 self.need = self.double_reg(ins.args[1]) + self.
                        double_reg(ins.args[2])
695             else:
696                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
697
698         elif ins.instr == 'sub.s':
699             if len(ins.args) == 3:
700                 self.gen = [ins.args[0]]
701                 self.need = [ins.args[1], ins.args[2]]
702             else:
703                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
704
705         elif ins.instr == 'sub.d':
706             if len(ins.args) == 3:
707                 self.gen = self.double_reg(ins.args[0])
708                 self.need = self.double_reg(ins.args[1]) + self.
                        double_reg(ins.args[2])
709             else:
710                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
711
712         elif ins.instr == 'mul.s':
713             if len(ins.args) == 3:
714                 self.gen = [ins.args[0]]
```

```
715                 self.need = [ins.args[1], ins.args[2]]
716             else:
717                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
718
719         elif ins.instr == 'mul.d':
720             if len(ins.args) == 3:
721                 self.gen = self.double_reg(ins.args[0])
722                 self.need = self.double_reg(ins.args[1]) + self.
                        double_reg(ins.args[2])
723             else:
724                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
725
726         elif ins.instr == 'div.s':
727             if len(ins.args) == 3:
728                 self.gen = [ins.args[0]]
729                 self.need = [ins.args[1], ins.args[2]]
730             else:
731                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
732
733         elif ins.instr == 'div.d':
734             if len(ins.args) == 3:
735                 self.gen = self.double_reg(ins.args[0])
736                 self.need = self.double_reg(ins.args[1]) + self.
                        double_reg(ins.args[2])
737             else:
738                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
739
740         elif ins.instr == 'abs.s':
741             if len(ins.args) == 2:
742                 self.gen = [ins.args[0]]
743                 self.need = [ins.args[1]]
744             else:
745                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
746
747         elif ins.instr == 'abs.d':
748             if len(ins.args) == 2:
749                 self.gen = self.double_reg(ins.args[0])
750                 self.need = self.double_reg(ins.args[1])
751             else:
752                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
753
754         elif ins.instr == 'neg.s':
755             if len(ins.args) == 2:
756                 self.gen = [ins.args[0]]
757                 self.need = [ins.args[1]]
758             else:
759                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
```

```
760
761         elif ins.instr == 'neg.d':
762             if len(ins.args) == 2:
763                 self.gen = self.double_reg(ins.args[0])
764                 self.need = self.double_reg(ins.args[1])
765             else:
766                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
767
768         elif ins.instr == 'sqrt.s':
769             if len(ins.args) == 2:
770                 self.gen = [ins.args[0]]
771                 self.need = [ins.args[1]]
772             else:
773                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
774
775         elif ins.instr == 'sqrt.d':
776             if len(ins.args) == 2:
777                 self.gen = self.double_reg(ins.args[0])
778                 self.need = self.double_reg(ins.args[1])
779             else:
780                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
781
782         elif ins.instr == 'cvt':
783             if len(ins.args) == 2:
784                 self.gen = [ins.args[0]]
785                 self.need = [ins.args[1]]
786             else:
787                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
788
789         elif ins.instr == 'cvt.d.w':
790             if len(ins.args) == 2:
791                 self.gen = self.double_reg(ins.args[0])
792                 self.need = [ins.args[1]]
793             else:
794                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
795
796         elif ins.instr == 'cvt.s.d':
797             if len(ins.args) == 2:
798                 self.gen = [ins.args[0]]
799                 self.need = self.double_reg(ins.args[1])
800             else:
801                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
802
803         elif ins.instr == 'cvt.d.s':
804             if len(ins.args) == 2:
805                 self.gen = self.double_reg(ins.args[0])
806                 self.need = [ins.args[1]]
807             else:
```

```
808                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
809
810         elif ins.instr == 'cvt.s.w':
811             if len(ins.args) == 2:
812                 self.gen = [ins.args[0]]
813                 self.need = [ins.args[1]]
814             else:
815                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
816
817         elif ins.instr == 'cvt.w.s':
818             if len(ins.args) == 2:
819                 self.gen = [ins.args[0]]
820                 self.need = [ins.args[1]]
821             else:
822                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
823
824         elif ins.instr == 'cvt.w.d':
825             if len(ins.args) == 2:
826                 self.gen = [ins.args[0]]
827                 self.need = self.double_reg(ins.args[1])
828             else:
829                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
830
831         elif ins.instr == 'c.eq.s':
832             if len(ins.args) == 2:
833                 self.gen = [Register("$fcc")]
834                 self.need = [ins.args[0],ins.args[1]]
835             else:
836                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
837
838         elif ins.instr == 'c.eq.d':
839             if len(ins.args) == 2:
840                 self.gen = [Register("$fcc")]
841                 self.need = self.double_reg(ins.args[0]) + self.
                        double_reg(ins.args[1])
842             else:
843                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
844
845         elif ins.instr == 'c.lt.s':
846             if len(ins.args) == 2:
847                 self.gen = [Register("$fcc")]
848                 self.need = [ins.args[0],ins.args[1]]
849             else:
850                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
851
852         elif ins.instr == 'c.lt.d':
853             if len(ins.args) == 2:
```

```
854                 self.gen = [Register("$fcc")]
855                 self.need = self.double_reg(ins.args[0]) + self.
                        double_reg(ins.args[1])
856             else:
857                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
858 
859         elif ins.instr == 'c.le.s':
860             if len(ins.args) == 2:
861                 self.gen = [Register("$fcc")]
862                 self.need = [ins.args[0],ins.args[1]]
863             else:
864                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
865 
866         elif ins.instr == 'c.le.d':
867             if len(ins.args) == 2:
868                 self.gen = [Register("$fcc")]
869                 self.need = self.double_reg(ins.args[0]) + self.
                        double_reg(ins.args[1])
870             else:
871                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
872 
873         elif ins.instr == 'trunc.l.d':
874             if len(ins.args) == 3:
875                 self.gen = [ins.args[0]]
876                 self.need = self.double_reg(ins.args[1]) + [ins.args
                        [2]]
877             else:
878                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
879 
880         elif ins.instr == 'trunc.l.s':
881             if len(ins.args) == 3:
882                 self.gen = [ins.args[0]]
883                 self.need = [ins.args[1],ins.args[2]]
884             else:
885                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
886 
887         elif ins.instr == 'trunc.w.d':
888             if len(ins.args) == 3:
889                 self.gen = [ins.args[0]]
890                 self.need = self.double_reg(ins.args[1]) + [ins.args
                        [2]]
891             else:
892                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
893 
894         elif ins.instr == 'trunc.w.s':
895             if len(ins.args) == 3:
896                 self.gen = [ins.args[0]]
897                 self.need = [ins.args[1],ins.args[2]]
```

```
898             else:
899                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
900
901
902     def parse_misc(self,ins):
903         """
904         Parses the misc type instructions.
905         """
906         if ins.instr == 'nop':
907             if len(ins.args) != 0:
908                 raise Exception("Invalid number of args for ins: ",
                            ins.instr)
909         elif ins.instr == 'syscall':
910             if len(ins.args) != 0:
911                 raise Exception("Invalid number of args for ins: ",
                            ins.instr)
912         elif ins.instr == 'break':
913             if len(ins.args) != 0:
914                 raise Exception("Invalid number of args for ins: ",
                            ins.instr)
915         elif ins.instr == 'mflo':
916             if len(ins.args) == 1:
917                 self.gen = [ins.args[0]]
918                 self.need = [Register("$lo")]
919             else:
920                 raise Exception("Invalid number of args for ins: ",
                            ins.instr)
921
922         elif ins.instr == 'mtlo':
923             if len(ins.args) == 1:
924                 self.gen = [Register("$lo")]
925                 self.need = [ins.args[0]]
926             else:
927                 raise Exception("Invalid number of args for ins: ",
                            ins.instr)
928
929         elif ins.instr == 'mfhi':
930             if len(ins.args) == 1:
931                 self.gen = [ins.args[0]]
932                 self.need = [Register("$hi")]
933             else:
934                 raise Exception("Invalid number of args for ins: ",
                            ins.instr)
935
936         elif ins.instr == 'mthi':
937             if len(ins.args) == 1:
938                 self.gen = [Register("$hi")]
939                 self.need = [ins.args[0]]
940             else:
941                 raise Exception("Invalid number of args for ins: ",
                            ins.instr)
942
943         elif ins.instr == 'mtc1':
```

```
944             if len(ins.args) == 2:
945                 self.gen = [ins.args[1]]
946                 self.need = [ins.args[0]]
947             else:
948                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
949
950         elif ins.instr == 'mfc1':
951             if len(ins.args) == 2:
952                 self.gen = [ins.args[0]]
953                 self.need = [ins.args[1]]
954             else:
955                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
956
957         elif ins.instr == 'la':
958             if len(ins.args) == 2:
959                 self.gen = [ins.args[0]]
960                 self.ival = ins.args[1]
961             else:
962                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
963
964         elif ins.instr == 'lui':
965             if len(ins.args) == 2:
966                 self.gen = [ins.args[0]]
967                 self.ival = ins.args[1]
968             else:
969                 raise Exception("Invalid number of args for ins: ",
                        ins.instr)
970
971     def parse_unknown(self,ins):
972         """
973         As we scanned all files were we will run benchmarks on, we
                will not
974         encounter any unknown instructions for this assignment. This
                function
975         is for later.
976         """
977         raise Exception("Unknown instruction type: ", ins.instr)
978
979     def double_reg(self, reg):
980         """
981         Return two registers in a list: the one that is given as an
                argument
982         and the one that comes after it (numrical).
983         This is for float operations with double precision.
984         """
985         global numreg
986         if type(reg) != Register:
987             raise Exception("Not a register object")
988         g = re.match(numreg, reg.expr)
989         return [reg, Register(g.group(1) + str(int(g.group(2)) + 1))]
990
```

```
991     def is_reg(self, val):
992         return type(val) == Register
993 
994 def main():
995     # test code
996     from asmyacc import parser
997 
998     flat = []
999     for line in open('../benchmarks/whet.s', 'r').readlines():
1000         if not line.strip(): continue
1001         flat.append(parser.parse(line))
1002 
1003     for inst in flat:
1004         if type(inst)==Instr:
1005             try:
1006                 Instruction(inst)
1007             except Exception as e:
1008                 print inst.instr, inst.args
1009                 raise e
1010 
1011 if __name__ == '__main__':
1012     main()
1013     pass
```

A.13. peephole.py

```
1  """
2  File:          peephole.py
3  Course:        Compilerbouw 2011
4  Author:        Joris Stork, Lucas Swartsenburg, Jeroen Zuiddam
5 
6  The peephole module
7 
8  Description:
9      Contains the Peephole and Peeper classes used for optimising code.
10 
11 """
12 
13 
14 class Peephole(object):
15     """
16     wrapper for an iterable list of instructions; implemented to
           emulate a
17     standard Python iterable object; optimisations are made on one
           Peephole
18     instance at a time
19 
20     """
21 
22     def __init__(self, block, start, size):
23         self.block = block
24         self.start_index = start
25         self.size = size
```

```
26          self.instructions = self.block[start:size]
27          self.counter = 0
28
29
30      def __iter__(self):
31          return self
32
33
34      def next(self):
35          if self.counter >= self.size:
36              raise StopIteration
37          else:
38              index1 = self.start_index + self.counter
39              self.current_instruction = self.block[index1]
40              self.counter += 1
41          return self.current_instruction
42
43
44      def __getitem__(self, index):
45          """   """
46
47          try:
48              return self.block[self.start_index + index]
49          except TypeError:
50              index1 = self.start_index + index.start
51              index2 = self.start_index + index.stop
52              return self.block[index1:index2]
53
54
55      def __setitem__(self, index, value):
56          """   """
57
58          self.block[self.start_index + index] = value
59
60
61      def __delitem__(self, index):
62          """   """
63
64          try:
65              del self.block.instructions[self.start_index + index]
66          except TypeError:
67              index1 = self.start_index + index.start
68              index2 = self.start_index + index.stop
69              del self.block.instructions[index1:index2]
70          self.size = self.size - 1
71
72
73      def __len__(self):
74          """   """
75
76          return len(self.block[self.start_index:self.size])
77
78
79
```

```
80 class Peeper(object):
81     """
82     generates successive peepholes so that these "slide" over a basic
           block
83     of instructions
84 
85     """
86 
87     def __init__(self, block, peephole_size):
88         self.block = block
89         self.p_size = peephole_size
90         self.counter = 0
91 
92 
93     def __iter__(self):
94         return self
95 
96 
97     def next(self):
98         if self.p_size + self.counter > len(self.block):
99             raise StopIteration
100        else:
101            self.peephole = Peephole(self.block, self.counter, self.
                   p_size)
102            self.counter += 1
103        return self.peephole
```

A.14. ranker.py

```
1  #!/usr/bin/env python
2  """
3  File:          ranker.py
4  Course:        Compilerbouw 2011
5  Author:        Joris Stork, Lucas Swartsenburg, Jeroen Zuiddam
6 
7  The peephole module
8 
9  Description:
10     A little utility to compile a dict that ranks the instructions in
           the
11     benchmark suite by incidence. Opcodes are keys, with the numbers
           of
12     occurrences of the instruction in the benchmark suite stored in
           the
13     corresponding dict value.
14 
15 """
16 
17 from optparse import OptionParser
18 from asmyacc import parser
19 import ir
20 from operator import itemgetter
21 
```

```
22
23 class Ranker(object):
24
25     def __init__(self):
26         self.flat = []
27         self.rankingdict = {}
28         self.ranking = None
29
30     def addlines(self, lines):
31         for line in lines:
32             if not line.strip():
33                 continue
34             self.flat.append(parser.parse(line))
35
36
37     def rank(self):
38         for line in self.flat:
39             if isinstance(line, ir.Instr):
40                 if line.instr in self.rankingdict:
41                     self.rankingdict[line.instr] = self.rankingdict[
                           line.instr] + 1
42                 else:
43                     self.rankingdict[line.instr] = 1
44         self.ranking = sorted(self.rankingdict.iteritems(), key=
               itemgetter(1), reverse=True)
45         print self.ranking
46
47
48     def result(self):
49         return [str(instr)+'\n' for instr in self.ranking]
50
51
52 def main():
53     sourcefiles = []
54     ranker = Ranker()
55     filelist = ['acron.s', 'clinpack.s', 'dhrystone.s', 'pi.s', '
           slalom.s', 'whet.s']
56     for filename in filelist:
57         sourcefile = open(('../benchmarks/'+filename), 'r')
58         ranker.addlines(sourcefile.readlines())
59         sourcefile.close()
60     ranker.rank()
61     targetfile = open('irank', 'w')
62     targetfile.writelines(ranker.result())
63     targetfile.close()
64
65 if __name__ == '__main__':
66     main()
```

A.15. test_block_optimise.py

```
1 #! /usr/bin/env python
2 import unittest
```

```
3  import block_optimise
4  import ir
5  import cfg
6
7
8  class TestBlockOptimisers(unittest.TestCase):
9      """ unittests for the block_optimise module  """
10
11
12     def setUp(self):
13         pass
14
15
16     def tearDown(self):
17         pass
18
19
20     def test_const_fold_addu(self):
21         dest1 = ir.Register('$dest1')
22         dest2 = ir.Register('$dest2')
23         dest3 = ir.Register('$dest3')
24         adest1 = ir.Register('$adest1')
25         adest2 = ir.Register('$adest2')
26         instrs = [ir.Instr('filler', ['dest',1634563464,64])]
27         instrs.append(ir.Instr('li', [dest1, 0x0001]))
28         instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
29         instrs.append(ir.Instr('li', [dest2, 0x0002]))
30         instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
31         instrs.append(ir.Instr('addu', [adest1,dest1,dest2]))
32         instrs.append(ir.Instr('li', [dest3, 0x0003]))
33         instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
34         instrs.append(ir.Instr('addu', [adest2,adest1,dest3]))
35         instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
36         block = cfg.BasicBlock(instrs)
37         bop = block_optimise.ConstantFold(block = block)
38         bop.optimise()
39         expected = str(ir.Instr('li', [adest2, '0x6']))
40         result = str(block[8])
41         self.assertTrue(result == expected)
42
43
44     def test_copyprop_move(self):
45         copy1 = ir.Register('$copy1')
46         orig1 = ir.Register('$orig1')
47         sp = ir.Register('$sp')
48         fp = ir.Register('$fp')
49         r31 = ir.Register('$31')
50         offsetadd1 = ir.Register('20($sp)')
51         offsetadd1_f = ir.Register('20($fp)')
52         offsetadd2 = ir.Register('16($sp)')
53         offsetadd2_f = ir.Register('16($fp)')
54         instrs = [ir.Instr('filler', ['$filler',1634563464,64])]
55         instrs.append(ir.Instr('move', [copy1, orig1]))
56         instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
```

```
57          instrs.append(ir.Instr('addu', ['$adest1',copy1,'somereg']))
58          instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
59          instrs.append(ir.Instr('addu', [copy1,'somereg',copy1]))
60          instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
61          instrs.append(ir.Instr('addu', ['$adest2',copy1,'somereg']))
62          instrs.append(ir.Instr('move', [sp,fp]))
63          instrs.append(ir.Instr('lw', [r31, offsetadd1]))
64          instrs.append(ir.Instr('lw', [fp, offsetadd2]))
65          instrs.append(ir.Instr('addu', [sp,sp,24]))
66          block = cfg.BasicBlock(instrs)
67          bop = block_optimise.CopyPropagation(block = block)
68          bop.optimise()
69          exp_instrs = [ir.Instr('filler', ['$filler',1634563464,64])]
70          exp_instrs.append(ir.Instr('move', [copy1, orig1]))
71          exp_instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
72          exp_instrs.append(ir.Instr('addu', ['$adest1', orig1, 'somereg
                ']))
73          exp_instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
74          exp_instrs.append(ir.Instr('addu', [copy1, 'somereg', orig1]))
75          exp_instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
76          exp_instrs.append(ir.Instr('addu', ['$adest2', copy1, 'somereg
                ']))
77          exp_instrs.append(ir.Instr('move', [sp,fp]))
78          exp_instrs.append(ir.Instr('lw', [r31, offsetadd1_f]))
79          exp_instrs.append(ir.Instr('lw', [fp, offsetadd2_f]))
80          exp_instrs.append(ir.Instr('addu', [sp,sp,24]))
81          for i in xrange(len(block)):
82              self.assertTrue(str(block[i]) == str(exp_instrs[i]))
83
84
85      def test_dead_code(self):
86          reg1 = ir.Register('$1')
87          reg2 = ir.Register('$2')
88          reg3 = ir.Register('$3')
89          reg4 = ir.Register('$4')
90          reg5 = ir.Register('$5')
91          sp = ir.Register('$sp')
92          fp = ir.Register('$fp')
93          r31 = ir.Register('$31')
94          offsetadd1 = ir.Register('20($sp)')
95          offsetadd2 = ir.Register('16($sp)')
96          instrs = [ir.Instr('filler', ['$filler',1634563464,64])]
97          instrs.append(ir.Instr('addu', [reg1,reg2,reg3]))
98          instrs.append(ir.Instr('filler2', ['dest',9999,64]))
99          instrs.append(ir.Instr('sll', [reg3, reg4, 10]))
100         instrs.append(ir.Instr('filler3', ['dest',9999,64]))
101         instrs.append(ir.Instr('lw', [reg3, '0($dp)']))
102         instrs.append(ir.Instr('filler4', ['dest',9999,64]))
103         instrs.append(ir.Instr('addu', [reg1,reg2,reg4]))
104         instrs.append(ir.Instr('filler5', ['dest',9999,64]))
105         instrs.append(ir.Instr('addu', [reg3,reg1,reg2]))
106         instrs.append(ir.Instr('filler6', ['dest',9999,64]))
107         instrs.append(ir.Instr('div', [reg1,reg2,reg4]))
108         instrs.append(ir.Instr('move', [sp,fp]))
```

```
109         instrs.append(ir.Instr('lw', [r31, offsetadd1]))
110         instrs.append(ir.Instr('lw', [fp, offsetadd2]))
111         instrs.append(ir.Instr('addu', [sp,sp,24]))
112         block = cfg.BasicBlock(instrs)
113         bop = block_optimise.DeadCode(block = block)
114         bop.optimise()
115         exp_instrs = [ir.Instr('filler', ['$filler',1634563464,64])]
116         exp_instrs.append(ir.Instr('filler2', ['dest',9999,64]))
117         exp_instrs.append(ir.Instr('filler3', ['dest',9999,64]))
118         exp_instrs.append(ir.Instr('filler4', ['dest',9999,64]))
119         exp_instrs.append(ir.Instr('addu', [reg1,reg2,reg4]))
120         exp_instrs.append(ir.Instr('filler5', ['dest',9999,64]))
121         exp_instrs.append(ir.Instr('addu', [reg3,reg1,reg2]))
122         exp_instrs.append(ir.Instr('filler6', ['dest',9999,64]))
123         exp_instrs.append(ir.Instr('div', [reg1,reg2,reg4]))
124         exp_instrs.append(ir.Instr('move', [sp,fp]))
125         exp_instrs.append(ir.Instr('lw', [r31, offsetadd1]))
126         exp_instrs.append(ir.Instr('lw', [fp, offsetadd2]))
127         exp_instrs.append(ir.Instr('addu', [sp,sp,24]))
128         for i in xrange(len(block)):
129             self.assertTrue(str(block[i]) == str(exp_instrs[i]))
130 
131 
132 #     def test_algebra(self):
133 #         dest1 = ir.Register('$dest1')
134 #         dest2 = ir.Register('$dest2')
135 #         divdest1 = ir.Register('divdest1')
136 #         instrs = [ir.Instr('li', [dest1, 1634563464])]
137 #         instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
138 #         instrs.append(ir.Instr('li', [dest2, 64]))
139 #         instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
140 #         instrs.append(ir.Instr('div.d', [divdest1,dest1,dest2]))
141 #         instrs.append(ir.Instr('filler', ['dest',1634563464,64]))
142 #         block = cfg.BasicBlock(instrs)
143 #         bop = block_optimise.AlgebraicTransformations(block = block)
144 #         bop.optimise()
145 #         expected = str(ir.Instr('sra', [divdest1, 1634563464, 6.0]))
146 #         result = str(block[4])
147 #         #TODO: map assertion to list of (expected,result)
148 #         self.assertTrue(result == expected)
149 
150 
151 if __name__ == '__main__':
152     unittest.main()
```

A.16. uic.py

```
1  """
2  File:          uic.py
3  Course:        Compilerbouw 2011
4  Author:        Joris Stork, Lucas Swartsenburg, Jeroen Zuiddam
5 
6  The "useful instruction categories" module
```

```
7
8  Description:
9      Two lists and a dict to categorise instructions in ways that are
           useful for
10     the peephole optimisers.
11
12 """
13
14
15 """
16 instructions (keys) that respectively assign to the registers in args
       [(values)]
17
18 """
19 assign_to = {
20     'abs.d' :   0   ,
21     'abs.s' :   0   ,
22     'add'   :   0   ,
23     'add.d' :   0   ,
24     'addu'  :   0   ,
25     'add.s' :   0   ,
26     'and'   :   0   ,
27     'cvt.d.w':  0   ,
28     'cvt.d.s':  0   ,
29     'cvt.s.d':  0   ,
30     'cvt.s.w':  0   ,
31     'cvt'   :   0   ,
32     'c.s'   :   0   ,
33     'c.d'   :   0   ,
34     'divu'  :   0   ,
35     'sub.u' :   0   ,
36     'sub.d' :   0   ,
37     'add.d' :   0   ,
38     'div.d' :   0   ,
39     'div.s' :   0   ,
40     'div'   :   0   ,
41     'dlw'   :   0   ,
42     'lb'    :   0   ,
43     'l.s'   :   0   ,
44     'l.d'   :   0   ,
45     'l.a'   :   0   ,
46     'li'    :   0   ,
47     'lw'    :   0   ,
48     'lbu'   :   0   ,
49     'lh'    :   0   ,
50     'lhu'   :   0   ,
51     'move'  :   0   ,
52     'mov.d' :   0   ,
53     'mov.s' :   0   ,
54     'mult'  :   0   ,
55     'mult.s':   0   ,
56     'mult.d':   0   ,
57     'multu' :   0   ,
58     'neg.s' :   0   ,
```

```
59      'neg.d' :   0   ,
60      'nor'   :   0   ,
61      'or'    :   0   ,
62      'sra'   :   0   ,
63      'sub.s' :   0   ,
64      'subu'  :   0   ,
65      'sub'   :   0   ,
66      'srl'   :   0   ,
67      'sll'   :   0   ,
68      'sra'   :   0   ,
69      'slt'   :   0   ,
70      'sqrt.s':   0   ,
71      'sqrt.d':   0   ,
72      'sltu'  :   0   ,
73      'xor'   :   0   ,
74      'xori'  :   0
75      }
76
77
78  """
79  these instruction types may be subject to substituting copied-to
        register
80  references with original register references as part of a copy
        propagation
81  optimisation (see report)
82
83  """
84  copy_prop_targets = [
85      'add',       #- integer add
86      'addu',      #- integer add unsigned
87      'add.s',     #- single-precision (SP) add
88      'add.d',     #- double-precision (DP) add
89      'and',       #- logical AND
90      'beq',    #- branch == 0
91      'bne',    #- branch != 0
92      'blez',   #- branch <= 0
93      'bgtz',   #- branch > 0
94      'bltz',   #- branch < 0
95      'bgez',   #- branch >= 0
96      'bc1f',    #- Branch on floating point compare false.
97      'bc1t',    #- Branch on floating point compare true.
98      'cvt',        #- int., single, double conversion
99      'c.s',       #- SP compare
100     'c.d',       #- DP compare
101     'dsw',   #- store double word
102     'div',       #- integer divide
103     'divu',      #- integer divide unsigned
104     'dlw',   #- load double word
105     'or',        #- logical OR
106     'nor',       #- logical NOR
107     'srl',       #- shift right logical
108     'sra',       #- shift right arithmetic
109     'slt',       #- set less than
110     'sltu',      #- set less than unsigned
```

```
111     'sub.s',     #- SP subtract
112     'sub.d',     #- DP subtract
113     'div.s',     #- SP divide
114     'div.d',     #- DP divide
115     'abs.s',     #- SP absolute value
116     'abs.d',     #- DP absolute value
117     'neg.s',     #- SP negation
118     'neg.d',     #- DP negation
119     'cvt',       #- int., single, double conversion
120     'c.s',       #- SP compare
121     'c.d',       #- DP compare
122     'jr',     #- jump register
123     'jalr',  #- jump and link register
124     'lb',   #- load byte
125     'lbu',  #- load byte unsigned
126     'lh',   #- load half (short)
127     'lhu',  #- load half (short) unsigned
128     'lw',   #- load word
129     'l.s',  #- load single-precision FP
130     'l.d',  #- load double-precision FP
131     'li'
132     'move', # extra
133     'mult.s',    #- SP multiply
134     'mult.d',    #- DP multiply
135     'mult',      #- integer multiply
136     'multu',     #- integer multiply unsigned
137     'nor',  #- logical NOR
138     'or',   #- logical OR
139     'sll',       #- shift left logical
140     'srl',  #- shift right logical
141     'sra',  #- shift right arithmetic
142     'slt',  #- set less than
143     'sltu', #- set less than unsigned
144     'sub',       #- subtract
145     'subu',      #- integer subtract unsigned
146     'sub.s',     #- SP subtract
147     'sub.d',     #- DP subtract
148     'sqrt.s',    #- SP square root
149     'sqrt.d',    #- DP square root
150     'sb',   #- store byte
151     'sbu',  #- store byte unsigned
152     'sh',   #- store half (short)
153     'shu',  #- store half (short) unsigned
154     'sw',   #- store word
155     's.s',  #- store single-precision FP
156     's.d',  #- store double-precision FP
157     'xor'       #- logical XOR
158     ]
159
160
161 """
162 if a copy propagate optimiser encounters one of these instructions
        with a
```

```
163 copied-to register reference or a copied-from register reference in
        its
164 arguments list during a scan of instructions below a copy, no
        instructions
165 beyond that instruction may be considered safe; ultimately this list
        should be
166 empty, since it just holds instructions we haven't checked for format
        and
167 semantics
168
169 """
170 copy_prop_unsafe = [
171     'mflo',
172     'mtcl',
173     'la',
174     'mfc1',
175     'dmfc1',
176     ]
```