Loop-invariant code motion in `loop.c`

Reynald Affeldt*

November 21, 2000

Introduction

The loop optimization pass takes place after the global common subexpression pass. It is supposed to move constant expressions out of loops, do strength-reduction and loop unrolling. The `-O2` option turns on most loop optimizations. Loop unrolling is triggered by `-O3`. Other (self-explained) loop optimization-oriented options are: strength-reduce, rerun-loop-opt, unroll-loops, unroll-all-loops, move-all-movables, reduce-all-givs (usage: `-f`*optimization_name*). `-dL` dumps the RTL code after the loop optimization pass into a `.loop`-extended file.

We are here interested in loop-invariant detection and motion. That is:

(before invariant code motion)

```
inv = 0;
do {
  inv = constant_computation;
  acc[i] = inv;
  i++;
} while (i < lim);
ret = inv;
```

(after invariant code motion)

```
inv = 0;
tmp = constant_computation;
do {
  inv = tmp;
  acc[i] = inv;
  i++;
} while (i < lim);
ret = inv;
```

Hereafter, a computation is the execution of machine instructions that evaluate an expression with side-effects (e.g., definitions) or not (e.g., conditional tests).

Definition 1 (Loop-invariant) A computation is loop-invariant within a given loop if for each operand:

1. the operand is a constant, or
2. all the definitions of the operand that reach the computation are outside the loop, or
3. only one definition of the operand reaches the computation, this definition is inside the loop, and it is itself loop-invariant.

Because of this inductive definition, loop-invariant computations can be found by an iterative algorithm:

1. mark as loop-invariant constant operands,
2. mark as loop-invariant operands defined outside the loop,
3. mark repeatedly as loop-invariant computations whose operands:
   - are marked as loop-invariant, and/or
   - have only one definition inside the loop marked as loop-invariant.

Once loop-invariant computations are found, we may move them to the beginning of the loop, provided their order is respected. We can always transfer a computation without side-effects. For a computation with side-effects, it is slightly more difficult.

*affeldt@yl.is.s.u-tokyo.ac.jp, 96393

(more then one definition)

```
inv = 0;
do {
  inv = constant_computation;
  acc[i] = inv;
  inv = 0;
  acc[j] = inv;
  i++;
} while (i < lim);
ret = inv;
```

(one use before the definition)

```
inv = 0;
do {
  acc[j] = inv;
  inv = constant_computation;
  acc[i] = inv;
  i++;
} while (i < lim);
ret = inv;
```

Definition 2 (To dominate) A definition dominates an instruction in a loop if every path in the control-flow graph going from the beginning of the loop to that instruction must go through the definition.

A loop-invariant definition may be moved to the beginning of the loop in which it is detected:

- if it dominates all the exits of the loop (this handles in particular the case in which the loop may be executed zero times), and
- if it dominates all the uses of the defined variable in the loop, and
- if it is the only one definition of the defined variable in the loop.

As long as they fulfilled those conditions, loop-invariant instructions can be moved to the beginning of the loop (provided the order they were defined in is not altered).

References

[APP99] Andrew W. Appel, *Modern Compiler Implementation in C*, Cambridge University Press, 1999.

[MUC97] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997.

In the rest of this document, we illustrate loop-invariant code motion by its implementation in the optimizer of the GNU compiler. We first review some points about RTL, then present some basic data structures and helper functions, and eventually move on to following the flow of control in the file `loop.c`.

Contents

1 RTL expressions

An RTL expression is a C structure `typedef`ed `*rtx` in `rtl.h`. RTL distinguishes expressions by their codes `enum`ed `rtx_code` in `rtl.h` and `rtl.def`. See Table 1.

RTX code	Description	RTX class
`CONST_INT`	Integer value	o
`CONST_DOUBLE`	Floating-point or large integer value	o
`SYMBOL_REF`	Memory location	o
`CONST`	Result of an assembly-time arithmetic computation	o
`LABEL_REF`	Value of an assembler label for code	o
`PC`	Machine's program counter	o
`CC0`	Machine's condition code register	o
`REG`	Register	o
`MEM`	Memory location	o
`INSN`	Instruction (neither a jump nor a function call)	i
`JUMP_INSN`	Possible jump	i
`CALL_INSN`	Possible function call	i
`SET`	Side effect	x

Table 1: RTL codes

For each RTL code, there is a sequence of letters called its format. See Table 2.

Format letter	Description
`e`	Expression
`i`	Integer
`w`	Wide integer
`s`	String
`E`	Vector of expressions

Table 2: RTL format

RTL expressions are manipulated through macros. See Table 3.

Macro	Description
`GET_CODE(...)`	Get the code of an RTL expression
`SET_CODE(...)`	Set the code of an RTL expression
`GET_RTX_FORMAT(...)`	Get the format of an RTL expression, as a C string.
`RTX_UNCHANGING_P(...)`	Nonzero if the value of the register or of the memory is not changed
`GET_MODE (...)`	Machine mode of the RTL expression
`XEXP (...)`	Access an operand in a RTL expression
`INSN_UID(...)`	Get the uid of an insn
`PREV_INSN(...)`	Get the previous insn
`NEXT_INSN(...)`	Get the next insn

Table 3: RTL macros

The code of a function is represented as a doubly-linked chain of RTL expressions called insns. The three first fields of an insn are:

- a unique id number (referred to as an `uid`),

- a pointer to the previous insn, and
- a pointer to the next insn.

Insns are further distinguished by six RTL expression codes:

- There are three kinds of instructions:
  - `insn`: no jump, no function call.
  - `jump_insn`: possible jump.
  - `call_insn`: possible function call.
- Labels to which a jump instruction may jump are represented by `code_label`s.
- After unconditional jumps or calls to functions that do not return (e.g., `exit (...)`), `barrier`s are placed: the control-flow cannot go through them.
- Notes (`note`s) represent extra information. See Table 4.

Note name	Description
`NOTE_INSN_BLOCK_BEG/END`	Block delimiters
`NOTE_INSN_LOOP_BEG/END`	While and for loops
`NOTE_INSN_LOOP_CONT`	`continue`s may jump here
`NOTE_INSN_LOOP_VTOP`	Virtual loop top
`NOTE_INSN_FUNCTION_BEG/END`	Function delimiters
`NOTE_INSN_BASIC_BLOCK`	Basic block entry
`REG_EQUIV/EQUAL`	
`REG_LIBCALL/REG_RETVAL`	Library sequence call delimiters

Table 4: RTL notes

`NOTE_INSN_LOOP_VTOP` indicates the place in a loop where the exit test begins for those loops in which the exit test has been duplicated. This position becomes another virtual start of the loop when considering loop invariants.

`REG_LIBCALL` and `REG_RETVAL` are notes that indicate a multi-insn sequence that performs a library call. They always occur in pair, and each insn points to the other. The `REG_LIBCALL` note is attached to the first insn of the sequence. The corresponding insn sets up arguments for the call. The `REG_RETVAL` note is attached to the last insn of the sequence. The corresponding insn copies the return value of the call.

`REG_EQUAL` is a note that describes the output of an insn. It is used in passes prior to register allocation and tells how to think of a value. It gives for the register being set an expression to which the register will be equal at runtime at the end of this insn (but not necessarily elsewhere in the function). It is typically an arithmetic expression. See Section 2 for an example.

`REG_EQUIV` is similar to `REG_EQUAL`. It is rather used after register allocation. It indicated an available substitute expression that may be used in place of a register if insufficient registers are available. The register will be equal to that expression during the entire lifetime of the function. It is typically used when a constant is loaded into a register that is not assigned anywhere else.

About machine modes, see Table 5.

Modes	Description
`SImode`	Single Integer, a four-byte integer
`VOIDmode`	No mode or unspecified mode

Table 5: Machine modes

2 Example of RTL code

To illustrate RTL, we compile we program "Simple loop" in two different ways.

Simple loop

```
int main () {
  int i, inv, acc;

  for (i = 7; i > 0; i--) {
    inv = 11;
    acc += inv;
  }

  return acc;
}
```

First, "Simple loop" is compiled with option `-O2`. Therefore, loop optimization has been performed. The RTL code is shown on the left, the corresponding assembly language code in shown on the right.

```
(note NOTE_INSN_BASIC_BLOCK)
(call_insn 6 (call (mem:QI (symbol_ref:SI ("__main")))))
(insn 11 (set (reg/v:SI 22) (const_int 7))
        (expr_list:REG_EQUAL (const_int 7)))

(note NOTE_INSN_LOOP_BEG)
(code_label L6)
(note NOTE_INSN_BASIC_BLOCK)
(insn 25 (set (reg/v:SI 24) (plus:SI (reg/v:SI 24)
                                (const_int 11))))
(note NOTE_INSN_LOOP_CONT)
(insn 31 (set (reg/v:SI 22) (plus:SI (reg/v:SI 22)
                                (const_int -1))))
(note NOTE_INSN_LOOP_VTOP)
(insn 14 (set (cc0) (reg/v:SI 22)))
(jump_insn 15 (set (pc) (if_then_else (gt (cc0)
                                    (const_int 0))
                                (label_ref L6)
                                (pc))))
(note NOTE_INSN_LOOP_END)
(note NOTE_INSN_BASIC_BLOCK)
(insn 41 (set (reg/i:SI %eax) (reg/v:SI 24)))
(insn 42 (use (reg/i:SI %eax)) -1)
```

```
 _main:
        call ___main
        movl 7,%eax

L6:
        addl 11,%ebx
        decl %eax
        testl %eax,%eax
        jg L6
        movl %ebx,%eax
```


Secondly, we compile "Simple loop" without any optimization. The RTL code is shown on the left, the assembly language code is shown on the right.

```
(call_insn 6 (call (mem:QI (symbol_ref:SI ("__main")))))
(insn 11 (set (mem/f:SI (plus:SI (reg:SI 18) (const_int -4)))
            (const_int 7)))

(note NOTE_INSN_LOOP_BEG)
(code_label L3)
(insn 14 (set (cc0) (mem/f:SI (plus:SI (reg:SI 18)
                                    (const_int -4)))))
(jump_insn 15 (set (pc) (if_then_else (gt (cc0)
                                        (const_int 0))
                                    (label_ref L6)
                                    (pc))))
(jump_insn 16 (set (pc) (label_ref L4)))
(barrier)

(code_label L6)
(insn 22 (set (mem/f:SI (plus:SI (reg:SI 18) (const_int -8)))
            (const_int 11)))
(insn 25 (set (reg:SI 22)
            (mem/f:SI (plus:SI (reg:SI 18) (const_int -8)))))
(insn 26 (set (mem/f:SI (plus:SI (reg:SI 18) (const_int -12)))
            (plus:SI (mem/f:SI (plus:SI (reg:SI 18) (const_int -12)))
                    (reg:SI 22)))
        (expr_list:REG_EQUAL
            (plus:SI (mem/f:SI (plus:SI (reg:SI 18) (const_int -12)))
                    (mem/f:SI (plus:SI (reg:SI 18) (const_int -8))))))

(note NOTE_INSN_LOOP_CONT)
(code_label L5)
(insn 32 (set (mem/f:SI (plus:SI (reg:SI 18) (const_int -4)))
            (plus:SI (mem/f:SI (plus:SI (reg:SI 18) (const_int -4)))
                    (const_int -1))))
(jump_insn 33 (set (pc) (label_ref L3)))
(barrier)
(note NOTE_INSN_LOOP_END)

(code_label L4)
(insn 39 (set (reg:SI 24)
            (mem/f:SI (plus:SI (reg:SI 18) (const_int -12)))))
(insn 41 (set (reg/i:SI %eax) (reg:SI 24)))
(insn 42 (use (reg/i:SI %eax)))
(jump_insn 43 (set (pc) (label_ref L2)))
(barrier)

(code_label L2)
```

```
        call ___main
        movl 7,-4(%ebp)

L3:
        cmpl 0,-4(%ebp)
        jg L6
        jmp L4

L6:
        movl 11,-8(%ebp)
        movl -8(%ebp),%eax
        addl %eax,-12(%ebp)

L5:
        decl -4(%ebp)
        jmp L3

L4:
        movl -12(%ebp),%edx
        movl %edx,%eax
        jmp L2

L2:
```

3 Data structures and helper functions

3.1 The movable data structure

The GCC optimizer calls loop-invariant movables. During the analysis of a loop, a chain of `struct movable`s is made to record all the movable insns found. Then the entire can be scanned to decide which to move.

```
struct movable
{
  rtx insn;                       /* a movable insn */
  rtx set_src;                    /* set from that expression */
  rtx set_dest;                   /* destination register */
  rtx dependencies;               /* if the insn is a libcall , the list of expression used in that libcall */
  int consec;                     /* number of consecutive insn that must be moved with this one */
  int regno;                      /* number of the register set */
  short lifetime;                 /* lifetime of that register , a difference between uids */
  unsigned int cond : 1;          /* 1 if only conditionally movable */
  unsigned int force : 1;         /* 1 means we must move this insn */
  unsigned int global : 1;        /* 1 means reg is live outside this loop */
  unsigned int done : 1;          /* 1 inhibits further processing of this */
  unsigned int partial : 1;       /* 1 means this reg is used for zero-extending */
  unsigned int move_insn : 1;     /* 1 means that we call emit_move_insn to load src , rather than copying insn */
  unsigned int move_insn_first:1; /* same as above, when the insn is the first a group */
  unsigned int is_equiv : 1;      /* 1 means a REG_EQUIV is present on insn */
  struct movable *match;          /* first entry for same value */
  struct movable *forces;         /* an insn that must be moved if this is */
  struct movable *next;
};
```

3.2 invariant_p: is an expression invariant?

This function returns a nonzero value if the `rtx` passed is at least conditionally invariant over the current loop. In particular, 2 means a conditionally invariant expression.

First, we consider the basic cases. Note that for registers, it essentially relies on `set_in_loop` (number of times a register is set during the loop, negative value if it is only conditional).

```
int
invariant_p (x)
     register rtx x;
{
  code = GET_CODE (x);
  switch (code)
    {
    case CONST_INT:
    case CONST_DOUBLE:
    case CONST:
    case LABEL_REF:
      return 1;
    case PC:
    case CC0:
      return 0;
    case REG:
      if ((x == frame_pointer_rtx || x == hard_frame_pointer_rtx || x == arg_pointer_rtx)
          && ! current_function_has_nonlocal_goto)
        return 1;

      if (loop_has_call && REGNO (x) < FIRST_PSEUDO_REGISTER && call_used_regs[REGNO (x)])
        return 0;

      if (VARRAY_INT (set_in_loop, REGNO (x)) < 0)
        return 2;

      return VARRAY_INT (set_in_loop, REGNO (x)) == 0;
    default:
      break;
```



```
    }
```

Then, if we face a composite expression, we get its format (See Table 2) and scan subexpressions.

```
  fmt = GET_RTX_FORMAT (code);
  for (i = GET_RTX_LENGTH (code) - 1; i >= 0; i--)
    {
      if (fmt[i] == 'e')
        {
          int tem = invariant_p (XEXP (x, i));
          if (tem == 0)
            return 0;
          if (tem == 2)
            conditional = 1;
        }
      else if (fmt[i] == 'E')
        {
          register int j;
          for (j = 0; j < XVECLEN (x, i); j++)
            {
              int tem = invariant_p (XVECEXP (x, i, j));
              if (tem == 0)
                return 0;
              if (tem == 2)
                conditional = 1;
            }

        }
    }

  return 1 + conditional;
}
```

This function obviously contains a significant part of the definition of a loop-invariant computation.

3.3 consec_sets_invariant_p: are consecutive instructions invariant?

It returns nonzero if all the insns in the loop that set `reg` are `insn` and the immediately following insns, and if each of those insns sets `reg` in an invariant way (not counting uses of `reg` in them). The value is 2 if some of these insns are only conditionally invariant. It assumes moreover that `insn` itself is the first set of `reg` and that its source is invariant.

```
static int
consec_sets_invariant_p (reg, n_sets, insn)
     int n_sets;
     rtx reg, insn;
{
  /* crunch */
}
```

3.4 count_one_set: is a set safe to move?

`count_one_set` count and record any set in `x` which is contained in `insn`. It updates `may_not_move` and `last_set` for any register set in `x`.

`may_not_move` is set to 1 to avoid moving the register. It does that for clobbered registers. It does also that for registers set in the current block that were already set before. If this is not the first setting in current basic block, we check if the register was used between the previous one and this one. If so, neither one can be moved.

`last_set` is nonzero if and ony if the register has been set in the current basic block.

```
static void
count_one_set (insn, x, may_not_move, last_set)
     rtx insn, x;
     varray_type may_not_move;
     rtx *last_set;
{
  if (GET_CODE (x) == CLOBBER && GET_CODE (XEXP (x, 0)) == REG)
    VARRAY_CHAR (may_not_move, REGNO (XEXP (x, 0))) = 1;
```

```
if (GET_CODE (x) == SET || GET_CODE (x) == CLOBBER)
  {
    rtx dest = SET_DEST (x);
    while (GET_CODE (dest) == SUBREG
           || GET_CODE (dest) == ZERO_EXTRACT
           || GET_CODE (dest) == SIGN_EXTRACT
           || GET_CODE (dest) == STRICT_LOW_PART)
      dest = XEXP (dest, 0);
    if (GET_CODE (dest) == REG)
      {
        register int regno = REGNO (dest);
        if (VARRAY_INT (set_in_loop, regno) > 0 && last_set[regno] == 0)
          VARRAY_CHAR (may_not_move, regno) = 1;
        if (last_set[regno] != 0 && reg_used_between_p (dest, last_set[regno], insn))
          VARRAY_CHAR (may_not_move, regno) = 1;
        if (VARRAY_INT (set_in_loop, regno) < 127)
          ++VARRAY_INT (set_in_loop, regno);
        last_set[regno] = insn;
      }
  }
}
```

This function ensures the safety of code motion. It prevents a register that has been set more than once or whose value has been used before its definition from being elected as a loop-invariant. See the Introduction for relevant examples of implied errors.

3.5 count_loop_regs_set: information about register sets

count_loop_regs_set increments set_in_loop at the index of each register that is modified by an insn between the from and to arguments.

It stores in single_usage the single insn in which the register is used, if it is only used once. This includes used in REG_EQUAL notes. Otherwise, it is set to 0 (for no use) or const0_rtx for more than one use.

last_set is nonzero if and only if the register has been set in the current basic block. In that case, it is the insn that last set the register.

```
static void
count_loop_regs_set (from, to, may_not_move, single_usage, count_ptr, nregs)
     register rtx from, to;
     varray_type may_not_move;
     varray_type single_usage;
     int *count_ptr;
     int nregs;
{
  register rtx *last_set = (rtx *) alloca (nregs * sizeof (rtx));
  register rtx insn;
  register int count = 0;

  bzero ((char *) last_set, nregs * sizeof (rtx));
  for (insn = from; insn != to; insn = NEXT_INSN (insn))
    {
      if (GET_RTX_CLASS (GET_CODE (insn)) == 'i')
        {
          ++count;

          find_single_use_in_loop (insn, PATTERN (insn), single_usage);

          if (REG_NOTES (insn))
            find_single_use_in_loop (insn, REG_NOTES (insn), single_usage);

          if (GET_CODE (PATTERN (insn)) == SET || GET_CODE (PATTERN (insn)) == CLOBBER)
            count_one_set (insn, PATTERN (insn), may_not_move, last_set);
          else if (GET_CODE (PATTERN (insn)) == PARALLEL)
            {
              register int i;
              for (i = XVECLEN (PATTERN (insn), 0) - 1; i >= 0; i--)
                count_one_set (insn, XVECEXP (PATTERN (insn), 0, i), may_not_move, last_set);
            }
        }

      if (GET_CODE (insn) == CODE_LABEL || GET_CODE (insn) == JUMP_INSN)
        bzero ((char *) last_set, nregs * sizeof (rtx));
    }
  *count_ptr = count;
}
```

4 Flow of control in loop.c

4.1 loop_optimize: entry point

loop_optimize is the entry point of loop.c. It performs loop optimization on the function whose first insn is f.

First, loop_optimize sets variables that describe the loops. For instance, it determines the number of loops by counting NOTE_INSN_LOOP_BEG notes and put the result in max_loop_num. It also allocates and fills arrays such as loop_number_loop_starts and loop_number_loop_ends that give respectively the first and the last insns of each loop, loop_outer_loop that gives for each loop the containing loop (-1 if none), loop_invalid that is nonzero for invalid loops (loops such that an insn outside them points to them), or loop_has_call that is nonzero if there is a function call in the loop. It is the work of the find_and_verify_loops function to set those variables.

Eventually, loop_optimize scans all the loops, starting with the last ones first. Inner loops are optimized first because applying loop-invariant code motion to nested loops may raise new opportunities for optimization of the outer code.

```
void
loop_optimize (f, dumpfile, unroll_p, bct_p)
     rtx f;
     FILE *dumpfile;
     int unroll_p, bct_p;
{
  /* crunch */

  find_and_verify_loops (f);

  /* crunch */

  for (i = max_loop_num-1; i >= 0; i--)
    if (! loop_invalid[i] && loop_number_loop_ends[i])
      scan_loop (loop_number_loop_starts[i], loop_number_loop_ends[i], loop_number_loop_cont[i], unroll_p, bct_p);

  /* crunch */
}
```

4.2 scan_loop: loop-invariant detection

scan_loop scans the loop delimited by the insns loop_start and end. These insns are supposed to be respectively NOTE_INSN_LOOP_BEG and NOTE_INSN_LOOP_END. The main work of scan_loop is to build up a chain of all the insns that are loop-invariant. Such insns are hereafter called movables (see Section 3.1).

```
static void
scan_loop (loop_start, end, loop_cont, unroll_p, bct_p)
     rtx loop_start, end, loop_cont;
     int unroll_p, bct_p;
{
```

The variable scan_start is set to the label that locates the beginning of the loop. However, the loop may start with a jump. In that case, the real beginning of the loop is the target of that jump. In that case, INSN_IN_RANGE_P tests whether or not the label is in the loop. If not, a subsequent test will simply return, giving up loop optimization for that loop. This may happen for code like **do** { ... } **while** (0);. loop_entry_jump and loop_top records respectively the jump insn and the insn following the label.

```
  for (p = NEXT_INSN (loop_start);
```

```
    p != end
      && GET_CODE (p) != CODE_LABEL
      && GET_RTX_CLASS (GET_CODE (p)) != 'i'
      && (GET_CODE (p) != NOTE
          || ( NOTE_LINE_NUMBER (p) != NOTE_INSN_LOOP_BEG && NOTE_LINE_NUMBER (p) != NOTE_INSN_LOOP_END));
    p = NEXT_INSN (p))
  ;
scan_start = p;
if (GET_CODE (p) == JUMP_INSN)
  {
    loop_entry_jump = p;
    if (simplejump_p (p)
        && JUMP_LABEL (p) != 0
        && INSN_IN_RANGE_P (JUMP_LABEL (p), loop_start, end))
      {
        loop_top = next_label (scan_start);
        scan_start = JUMP_LABEL (p);
      }
  }
```

`threshold` is set in `scan_loop` but really used in `move_movables` (the function which decides when it is worth moving an insn to the beginning of the loop, see Section 4.3). Intuitively, the smaller `threshold` is, the harder it is to make an insn eligible for loop-invariant code motion.

Moving a definition to the beginning of the loop increases the life time of the register being set. For big loops (i.e., loops that feature function calls), moving is not that important (the bigger the loop is, the smaller the gain will be) and moreover less registers are available (so that moving may make register usage more difficult). Therefore, we shall adopt the following rule: an insn is desirable to move if the new lifetime is less than `threshold` times the old lifetime. Consequently, `threshold` is smaller for big loops, making it harder to be elected as a movable instruction.

```
threshold = (loop_has_call ? 1 : 2) * (1 + n_non_fixed_regs);
```

Then, `scan_loop` initializes a couple of arrays, all indexed by registers. Namely: `n_times_set`, that contains the number of times a register is set during the loop being scanned; `set_in_loop`, that contains the number of times a register is set during the loop, or a negative number when the register is conditionally invariant (`-1` for a constant, `-2` otherwise), and zero if it is an invariant register for sure (set by `move_movables` (see Section 4.3)); `may_not_optimize`, is `1` if the register cannot be moved; and `reg_single_usage`, is the insn in which the register was used if it was used only once (`const0_rtx` otherwise).

`scan_loop` calls the helper function `count_loop_regs_set` that increments `set_in_loop` for each register that is modified by an insn between `loop_top` (or `loop_start`) and `end`. When a register is used only once, it store the insn that touched it into `reg_single_usage`. `count_loop_regs_set` stops when it has scanned the `insn_count` insns of the loop.

```
count_loop_regs_set (loop_top ? loop_top : loop_start, end, may_not_optimize, reg_single_usage, &insn_count, nregs);
```

4.2.1 Loop-invariant detection

Now that we have all the informations describing the loop and some basic information about register usage, we effectively go through the code to find insns that are safe to move.

First, `in_libcall` is set to `1` if we are inside a multi-insn sequence that performs a library call, `0` otherwise. Loop optimization uses that information to treat the sequence as a single operation when performing code motion.

```
for (p = next_insn_in_loop (scan_start, scan_start, end, loop_top);
     p != NULL_RTX;
     p = next_insn_in_loop (p, scan_start, end, loop_top))
  {
    if (GET_RTX_CLASS (GET_CODE (p)) == 'i' && find_reg_note (p, REG_LIBCALL, NULL_RTX))
      in_libcall = 1;
    else if (GET_RTX_CLASS (GET_CODE (p)) == 'i' && find_reg_note (p, REG_RETVAL, NULL_RTX))
      in_libcall = 0;
```

If the insn is a single set, we are now concerned with the problem of finding what to use as the source of this set.

```
    if (GET_CODE (p) == INSN
        && (set = single_set (p))
        && GET_CODE (SET_DEST (set)) == REG
        && ! VARRAY_CHAR (may_not_optimize, REGNO (SET_DEST (set))))
      {
```


If a `REG_EQUIV` note is given, we shall use it as the source and mark that we should move this insn by calling `emit_move_insn` rather than duplicating the insn.

If a `REG_EQUAL` note is given that is a constant, we do as above. Indeed, the expression given by a `REG_EQUAL` note is not assured to be valid throughout the function (if it not constant).

If both a `REG_EQUAL` note and a `REG_RETVAL` note are given, we use the contents of `REG_EQUAL` anyway. However, we check what registers are referred to in the multi-insn sequence (`dependencies` is `0` when register are referred to, or a chain of insns otherwise) so as to move the dependent registers together with the current insn if it declared to be movable.

```
        rtx src = SET_SRC (set);

        temp = find_reg_note (p, REG_EQUIV, NULL_RTX);
        if (temp)
          src = XEXP (temp, 0), move_insn = 1;
        else
          {
            temp = find_reg_note (p, REG_EQUAL, NULL_RTX);
            if (temp && CONSTANT_P (XEXP (temp, 0)))
              src = XEXP (temp, 0), move_insn = 1;
            if (temp && find_reg_note (p, REG_RETVAL, NULL_RTX))
              {
                src = XEXP (temp, 0);
                dependencies = libcall_other_reg (p, src);
              }
          }
```

From here, starts a sequence of seven consecutive tests to determine whether or not the current insn is movable.

If the optimization of an inner loop creates a new register, we don't try to optimize the insn. The reason is that we don't know its lifetime and it is therefore harder to determine if we can benefit from moving it up.

```
        if (REGNO (SET_DEST (set)) >= max_reg_before_loop)
          ;
```

4.2.2 Loop-invariant safety

From now on, insns may be collected as movable computations in `movables`, a chain of `struct movable*`. `last_movable` is always the last element in `movables`, so we can add elements at the end.

The first condition requires that the source of the insn be at least conditionally invariant.

The second condition requires that, if dependent registers exist that must be moved together with the insn, then they must be at least conditionally invariant.

The third condition requires that the destination of the insn is set only once in the loop, or that it is only set by subsequent insns whose sources are at least conditionally invariant.

`maybe_never` is `1` if we are scanning insns that could be executed zero times. `call_passed` is `1` if we are scanning insns that might never executed due to a function call which might exit before they are reached.

If the insn can cause a trap (such as divide by zero, arithmetic overflow, ...), and if one of the above conditions is fulfilled, then moving the insn may introduce a bug: it is only OK to move an insn that can trap if it is guaranteed to execute once loop is entered. A function call might prevent the trap insn from being executed since it might exit.

```
        else if ((tem = invariant_p (src))
                 && (dependencies == 0
                     || ( tem2 = invariant_p (dependencies)) != 0)
                 && (VARRAY_INT (set_in_loop, REGNO (SET_DEST (set))) == 1
                     || ( tem1 = consec_sets_invariant_p
                             (SET_DEST (set),
                              VARRAY_INT (set_in_loop, REGNO (SET_DEST (set))),
                              p)))
                 && ! ((maybe_never || call_passed)
                       && may_trap_p (src)))
          {
```

Here is a special disposition settled in case of large loops (`loop_has_call` is nonzero). It may happen that two insns can be combined as long as they are both in the loop, but code motion of one of them will prevent us from performing that optimization. The condition is quite complicated: it is possible when the destination register is set only once, when it does not have a `REG_RETVAL` note, when is is not a hard register, ...

We replace the usage of the register by a note containing the destination register and the source expression. This note is kept, the previous insn is deleted, and the set register is set free.

```
register struct movable *m;
register int regno = REGNO (SET_DEST (set));
if (loop_has_call
    && VARRAY_RTX (reg_single_usage, regno) != 0
    && VARRAY_RTX (reg_single_usage, regno) != const0_rtx
    && REGNO_FIRST_UID (regno) == INSN_UID (p)
    && (REGNO_LAST_UID (regno) == INSN_UID (VARRAY_RTX (reg_single_usage, regno)))
    && VARRAY_INT (set_in_loop, regno) == 1
    && ! side_effects_p (SET_SRC (set))
    && ! find_reg_note (p, REG_RETVAL, NULL_RTX)
    && (! SMALL_REGISTER_CLASSES
         || (! ( GET_CODE (SET_SRC (set)) == REG && REGNO (SET_SRC (set)) < FIRST_PSEUDO_REGISTER)))
    && ! modified_between_p (SET_SRC (set), p, VARRAY_RTX (reg_single_usage, regno))
    && no_labels_between_p (p, VARRAY_RTX (reg_single_usage, regno))
    && validate_replace_rtx (SET_DEST (set), SET_SRC (set), VARRAY_RTX (reg_single_usage, regno)))
  {
    REG_NOTES (VARRAY_RTX (reg_single_usage, regno))
      = replace_rtx (REG_NOTES (VARRAY_RTX (reg_single_usage, regno)),
                     SET_DEST (set),
                     copy_rtx (SET_SRC (set)));
    PUT_CODE (p, NOTE);
    NOTE_LINE_NUMBER (p) = NOTE_INSN_DELETED;
    NOTE_SOURCE_FILE (p) = 0;
    VARRAY_INT (set_in_loop, regno) = 0;
    continue;
  }
```

At this point we have found an insn at least conditionally invariant. We allocate space for it and fill the data structure with related information.

If the insn is in some way conditionally invariant, we record this information in the `cond` field: it is `1` if either `invariant_p` or `consec_sets_invariant_p` returned `2`.

We set `set_in_loop` negative for the register being set, so that this register will be considered invariant for subsequent insns. We consider whether subsequent insns use the register in deciding whether it is worth actually moving.

Eventually, we add `m` to the end of the chain `movables`.

```
m = (struct movable *) alloca (sizeof (struct movable));
m->next = 0;
m->insn = p;
m->set_src = src;
m->dependencies = dependencies;
m->set_dest = SET_DEST (set);
m->force = 0;
m->consec = VARRAY_INT (set_in_loop, REGNO (SET_DEST (set))) - 1;
m->done = 0;
m->forces = 0;
m->partial = 0;
m->move_insn = move_insn;
m->move_insn_first = 0;
m->is_equiv = (find_reg_note (p, REG_EQUIV, NULL_RTX) != 0);
m->savemode = VOIDmode;
m->regno = regno;
m->cond = ((tem | tem1 | tem2) > 1);
m->global = (uid_luid[REGNO_LAST_UID (regno)] > INSN_LUID (end)
             || uid_luid[REGNO_FIRST_UID (regno)] < INSN_LUID (loop_start));
m->match = 0;
m->lifetime = (uid_luid[REGNO_LAST_UID (regno)] - uid_luid[REGNO_FIRST_UID (regno)]);
m->savings = VARRAY_INT (n_times_set, regno);
if (find_reg_note (p, REG_RETVAL, NULL_RTX))
  m->savings += libcall_benefit (p);
VARRAY_INT (set_in_loop, regno) = move_insn ? -2 : -1;
if (movables == 0)
  movables = m;
else
  last_movable->next = m;
last_movable = m;
```


4.2.3 Zero-extending a movable

This case handles a particular situation. It may happen that a register is set within the loop by zero-extending a narrower value. Then we would want to change these to zero the entire register once before the loop and merely copy the low part within the loop.

This condition detects such a situation. It checks whether the register is always set within its `STRICT_LOW_PART` or set to zero. This means that its higher bytes are always constant. So we want to perform the following optimization: clear the register outside the loop and inside the loop, just load the lower bytes. Of course, the machine must have an instruction to do so. Also, if the value loaded into the register depends on the same register, this cannot be done.

We set the field `partial` of the movable to `1`, meaning the register is used for zero-extending.

Eventually, we add `m` to the end of the chain `movables`.

```
else if (SET_SRC (set) == const0_rtx
         && GET_CODE (NEXT_INSN (p)) == INSN
         && (set1 = single_set (NEXT_INSN (p)))
         && GET_CODE (set1) == SET
         && (GET_CODE (SET_DEST (set1)) == STRICT_LOW_PART)
         && (GET_CODE (XEXP (SET_DEST (set1), 0)) == SUBREG)
         && (SUBREG_REG (XEXP (SET_DEST (set1), 0)) == SET_DEST (set))
         && !reg_mentioned_p (SET_DEST (set), SET_SRC (set1)))
  {
    register int regno = REGNO (SET_DEST (set));
    if (VARRAY_INT (set_in_loop, regno) == 2)
      {
        register struct movable *m;
        m = (struct movable *) alloca (sizeof (struct movable));
        m->next = 0;
        m->insn = p;
        m->set_dest = SET_DEST (set);
        m->dependencies = 0;
        m->force = 0;
        m->consec = 0;
        m->done = 0;
        m->forces = 0;
        m->move_insn = 0;
        m->move_insn_first = 0;
        m->partial = 1;
        m->global = (INSN_UID (p) >= max_uid_for_loop
                     || ( uid_luid[REGNO_LAST_UID (regno)] > INSN_LUID (end))
                     || ( uid_luid[REGNO_FIRST_UID (regno)] < INSN_LUID (p))
                     || ( labels_in_range_p (p, uid_luid[REGNO_FIRST_UID (regno)])));
        if (maybe_never && m->global)
          m->savemode = GET_MODE (SET_SRC (set1));
        else
          m->savemode = VOIDmode;
        m->regno = regno;
        m->cond = 0;
        m->match = 0;
        m->lifetime = (uid_luid[REGNO_LAST_UID (regno)] - uid_luid[REGNO_FIRST_UID (regno)]);
        m->savings = 1;
        VARRAY_INT (set_in_loop, regno) = -1;
        if (movables == 0)
          movables = m;
        else
          last_movable->next = m;
        last_movable = m;
      }
  }
}
```

4.2.4 Subroutine call passed

`call_passed` is `1` if we are scanning insns that might never be executed due to a subroutine call which might exit before they are reached. This matters for insns that trap. However, function call inside a `REG_LIBCALL/REG_RETVAL` block always return, so they don't matter.

```
    else if (GET_CODE (p) == CALL_INSN && ! in_libcall)
      call_passed = 1;
```

4.2.5 Loop not executed

maybe_never is 1 if we are scanning insns that could be executed zero time. This matter in case such as this one:

```
inv = 0;
while (i < lim) {
  inv = constant_computation;
  acc[i] = inv;
  i++;
}
ret = inv;
```

We may not enter the loop at all, so that moving the invariant set will corrupt the rest of the program.

The condition says that past a label or a jump, we get to insns for which we can't count on whether or how many times they will be executed during each iteration.

If we enter the loop in the middle, and scan around to the beginning, don't set maybe_never for that. This must be an unconditional jump, otherwise the code at the top of the loop might never be executed. Unconditional jumps are followed a by barrier then loop end.

```
    else if ((GET_CODE (p) == CODE_LABEL || GET_CODE (p) == JUMP_INSN)
             && ! (GET_CODE (p) == JUMP_INSN
                   && JUMP_LABEL (p) == loop_top
                   && NEXT_INSN (NEXT_INSN (p)) == end
                   && simplejump_p (p)))
      maybe_never = 1;
```

4.2.6 Virtual top implies execution

At the virtual top of a converted loop, insns are again known to be executed.

```
    else if (GET_CODE (p) == NOTE)
      {
        if (NOTE_LINE_NUMBER (p) == NOTE_INSN_LOOP_VTOP && loop_depth == 0)
          maybe_never = call_passed = 0;
        else if (NOTE_LINE_NUMBER (p) == NOTE_INSN_LOOP_BEG)
          loop_depth++;
        else if (NOTE_LINE_NUMBER (p) == NOTE_INSN_LOOP_END)
          loop_depth--;
      }
  }
```

This is the end of the tests determining if an insn is safe to move.

4.2.7 Code motion preamble

To prepare the RTL code for code motion, a few precautions must be taken.

First, we ignore any movable whose insn falls within a libcall which is part of another movable. We make use of the fact that the movable for the whole libcall value was made later and so appears later on the chain.

```
  ignore_some_movables (movables);
```

For each movable insn, we see if the register that it loads leads when it dies right into another conditionally movable insn. If so, record that the second insn forces the first one, since the second can be moved only if the first is.

```
  force_movables (movables);
```

See if there are multiple movable insns that load the same value. If there are, make all but the first point at the first one through the match field, and add the priorities of them all together as the priority of the first.

```
  combine_movables (movables, nregs);
```

Now consider each movable insn to decide whether it is worth moving. Store 0 in set_in_loop for each reg that is moved. Generally this increases code size, so do not move movables when optimizing for code size.

insn_count is the number of insns in the loop.

At the end, candidates that still are negative are those not moved. Change set_in_loop to indicate that those are not actually invariant.


```
  if (! optimize_size)
    move_movables (movables, threshold,
                   insn_count, loop_start, end, nregs);

  for (i = 0; i < nregs; i++)
    if (VARRAY_INT (set_in_loop, i) < 0)
      VARRAY_INT (set_in_loop, i) = VARRAY_INT (n_times_set, i);

  /* crunch */
}
```

4.3 move_movables: code motion

move_movables scans movables and move the insns that deserve to be moved. If two matching movables are found, i.e., movables that set different registers with the same value, move_movables can replace one register with the other.

```
static void
move_movables (movables, threshold, insn_count, loop_start, end, nregs)
     struct movable *movables;
     int threshold;
     int insn_count;
     rtx loop_start;
     rtx end;
     int nregs;
{
  register struct movable *m;

  for (m = movables; m; m = m->next)
    {
```

4.3.1 Code motion safety

It is almost the same test as the one for loop-invariant safety, the main difference being that we don't tolerate here conditionally invariants.

The first condition ignores the insn if it's already been moved.

The second condition says to move the insn if it is known to be invariant for sure. If not, the source, dependencies, and the consecutive insns are tested for invariance. Any conditionally invariant insn is ignored.

The last condition says an insn can be moved if it doesn't force any other insn to be moved, or if that other insn has already been moved.

```
      if (!m->done
          && (! m->cond
              || (1 == invariant_p (m->set_src)
                  && (m->dependencies == 0
                      || 1 == invariant_p (m->dependencies))
                  && (m->consec == 0
                      || 1 == consec_sets_invariant_p (m->set_dest, m->consec + 1, m->insn))))
          && (! m->forces || m->forces->done))
        {
```

At this point, the insn is safe to move. It is this test that decides the safety of moving an instruction.

4.3.2 Desirability

It is not always desirable to move an insn that proved to be loop-invariant. There are situations where it must be done, and other situations where we have a choice. To help the optimizer make this choice, we use the threshold variable already encountered in Section 4.2 in the computation of its desirability.

If already_moved is 1 for the current register, it means that the current insn must be moved. This may happen when another insn has been moved that it is safe only if the present insn is also moved.

The user can constraint an unconditional code motion by setting the flag_move_all_movables through command-line options (see the Introduction).

The last disjunction indicates that it is desirable to move the instruction when other moves that it forces have already been done. This is because the cost of moving the insn may therefore be smaller. The date have been prepared by force_movables (...) in scan_loop (...) (see Section 4.2).

To decide when it is worth moving an insn, we compute its desirability. The computation make use of the `threshold` variable set in `scan_loop` (see Section 4.2). An insn is desirable to move if the new lifetime of the register is no more than `threshold` times the old lifetime. If it's not desirable, it means the loop is so big that moving won't speed things up much, and it is liable to make register usage worse.

```
if (already_moved[regno]
    || flag_move_all_movables
    || (threshold * savings * m->lifetime) >= (moved_once[regno] ? insn_count * 2 : insn_count)
    || (m->forces && m->forces->done && VARRAY_INT (n_times_set, m->forces->regno) == 1))
  {
```

We are now ready to effectively move the insn. This operation may involve some special cases.

4.3.3 Code motion and zero-extension

First, let us consider the case where more than one register are set for zero-extending by the same value.

We find the last of the movables that set a register to this value (with loop **for** (m1 ...)). Then, we generate a `mov` instruction to reuse the value in that register (`gen_move_insn (...)`) and effectively emit a new insn at the beginning of the loop (`emit_insn_before (...)`). The moved register is marked as being able to share a hard register with the other matching insns. Then, it is erased from its previous location (`delete_insn (...)`).

`emit_insn_before` is the function used for code motion. It merely copies the insn at the beginning of the loop.

```
int count;
register struct movable *m1;
rtx first;
if (m->partial && m->match)
  {
    for (m1 = m; m1->match; m1 = m1->match);

    newpat = gen_move_insn (SET_DEST (PATTERN (m->insn)), SET_DEST (PATTERN (m1->insn)));

    i1 = emit_insn_before (newpat, loop_start);

    regs_may_share = gen_rtx_EXPR_LIST (VOIDmode, r1, gen_rtx_EXPR_LIST (VOIDmode, r2, regs_may_share));

    delete_insn (m->insn);
  }
```

4.3.4 Code motion and reemission

We are required to call `emit_move_insn (...)` to load the source register rather than copying the insn, that is to regenerate the insn being moved.

First, we delete what we have and then emit the move insn before the loop. If this is the first insn of a library call sequence, we skip to the end, delete every insn in the sequence except the last one, which will be handled in the normal manner. Then, the new sequence of insn is created by `emit_move_insn (...)`, and this is sequence is appended to the beginnin of the loop by `emit_insns_before (...)`.

Eventually, we reduce `threshold` by three. This is because the more registers we move, the less we like moving them.

```
else if (m->move_insn)
  {
    for (count = m->consec; count >= 0; count--)
      {
        if (GET_CODE (p) != NOTE && (temp = find_reg_note (p, REG_LIBCALL, NULL_RTX)))
          p = XEXP (temp, 0);

        if (GET_CODE (p) != NOTE && (temp = find_reg_note (p, REG_RETVAL, NULL_RTX)))
          {
            temp = XEXP (temp, 0);
            while (temp != p)
              temp = delete_insn (temp);
          }

        temp = p;
```



```
        p = delete_insn (p);
        while (p && GET_CODE (p) == NOTE)
          p = NEXT_INSN (temp) = NEXT_INSN (p);
      }

    start_sequence ();
    emit_move_insn (m->set_dest, m->set_src);
    temp = get_insns ();
    end_sequence ();

    i1 = emit_insns_before (temp, loop_start);
    if (! find_reg_note (i1, REG_EQUAL, NULL_RTX))
      REG_NOTES (i1)
        = gen_rtx_EXPR_LIST (m->is_equiv ? REG_EQUIV : REG_EQUAL, m->set_src, REG_NOTES (i1));

    threshold -= 3;
  }
```

4.3.5 Code motion

This is the third way of moving an insn.

`m->consec` is the number of consecutive following insns that must be moved with this one. Once again, the decision process features special cases for the treatment of library call sequences.

```
else
  {
    for (count = m->consec; count >= 0; count--)
      {
        rtx i1, temp;

        if (GET_CODE (p) != NOTE && (temp = find_reg_note (p, REG_LIBCALL, NULL_RTX)))
          p = XEXP (temp, 0);

        if (GET_CODE (p) != NOTE && (temp = find_reg_note (p, REG_RETVAL, NULL_RTX)))
          {
            /* crunch */
          }
```

Then, it depends on consecutive insn's characteristics. In the first case, the insn is in mode void (see Table 5).

```
        if (m->savemode != VOIDmode)
          {
            start_sequence ();

            /* crunch */

            sequence = gen_sequence ();
            end_sequence ();
            i1 = emit_insn_before (sequence, loop_start);
          }
```

In the second case, the consecutive insn is a function call.

```
        else if (GET_CODE (p) == CALL_INSN)
          {
            i1 = emit_call_insn_before (PATTERN (p), loop_start);
            if (CALL_INSN_FUNCTION_USAGE (p))
              CALL_INSN_FUNCTION_USAGE (i1) = copy_rtx (CALL_INSN_FUNCTION_USAGE (p));
          }
```

In the third case, the consecutive insn is the last insn of the consecutive insns that must be moved with the current movable. `move_insn_first` is 1: it means that we call `emit_move_insn` to move the insn, instead of just copying it.

```
        else if (count == m->consec && m->move_insn_first)
          {
            start_sequence ();
            emit_move_insn (m->set_dest, m->set_src);
```

```
        temp = get_insns ();
        end_sequence ();

        i1 = emit_insns_before (temp, loop_start);
        if (! find_reg_note (i1, REG_EQUAL, NULL_RTX))
          REG_NOTES (i1)
            = gen_rtx_EXPR_LIST ((m->is_equiv ? REG_EQUIV : REG_EQUAL), m->set_src, REG_NOTES (i1));
      }
```

The last case is also the general one. The consecutive insn is neither in void mode, nor function call, nor the last of all the consecutive insns. We here simply call `emit_insn_before (...)`.

```
    else
      i1 = emit_insn_before (PATTERN (p), loop_start);
```

At this point, the current insn (an insn consecutive to the current movable) has been moved or re-emitted. We can delete it and move on to the next one.

```
    temp = p;
    delete_insn (p);
    p = NEXT_INSN (p);

    while (p && GET_CODE (p) == NOTE)
      p = NEXT_INSN (temp) = NEXT_INSN (p);
  }
```

Eventually, we decrease the threshold to refrain from performing to much loop-invariant code motions (see Section 4.2).

```
    threshold -= 3;
  }
```

This is the end of the sequence of **if** () {...} **else** tests that decides how to move a movable. At this point, the movable has been moved.

4.3.6 After code motion

Some data structures need to be updated, so that we can take the last code motion into account. `already_moved` is self-explained, `moved_once` means the register as already been moved out of one loop. The register is now certainly invariant, this is reflected by `set_in_loop` being assigned to `0`. Because of code motion, the lifetime of the register must be recomputed: now, it is live at least the full length of the current loop. This is done by means of the `REGNO_FIRST_UID` and `REGNO_LAST_UID` macros.

```
already_moved[regno] = 1;

moved_once[regno] = 1;

if (! m->partial)
  VARRAY_INT (set_in_loop, regno) = 0;

m->done = 1;

if (uid_luid[REGNO_FIRST_UID (regno)] > INSN_LUID (loop_start))
  REGNO_FIRST_UID (regno) = INSN_UID (loop_start);
if (uid_luid[REGNO_LAST_UID (regno)] < INSN_LUID (end))
  REGNO_LAST_UID (regno) = INSN_UID (end);
```

4.3.7 Merging resgisters

We eventually looked at all the other matching movables, i.e., movables that set another register with the same value. The compilation can benefit from combining then with the current movable. The data have been set by `combine_movables (...)` at the end of `scan_loop (...)` (see Section 4.2).

The information is concealed into the `reg_map` array (indexed by registers, delivering back RTL expressions). You set an entry to a particular expression in order to schedule the later replacement of that entry by that expression. The scheduled insn is further deleted. If it is a library call sequence, the whole sequence is deleted.

The merged register is made invariant by assignment of its `set_in_loop` entry.


```
        if (! m->partial)
          for (m1 = movables; m1; m1 = m1->next)
            if (m1->match == m)
              {
                if (GET_MODE (m->set_dest) == GET_MODE (m1->set_dest))
                  reg_map[m1->regno] = m->set_dest;
                else
                  reg_map[m1->regno] = gen_lowpart_common (GET_MODE (m1->set_dest), m->set_dest);

                m1->done = 1;

                if ((temp = find_reg_note (m1->insn, REG_RETVAL, NULL_RTX)))
                  {
                    for (temp = XEXP (temp, 0); temp != m1->insn; temp = NEXT_INSN (temp))
                      delete_insn (temp);
                  }
                delete_insn (m1->insn);

                already_moved[m1->regno] = 1;

                if (! m->partial)
                  VARRAY_INT (set_in_loop, m1->regno) = 0;
              }
      }
    }
  }
```

At the end, we go through all the instructions in the loop. making all the register substitutions scheduled in `reg_map`.

```
  for (p = new_start; p != end; p = NEXT_INSN (p))
    if (GET_CODE (p) == INSN || GET_CODE (p) == JUMP_INSN || GET_CODE (p) == CALL_INSN)
      {
        replace_regs (PATTERN (p), reg_map, nregs, 0);
        replace_regs (REG_NOTES (p), reg_map, nregs, 0);
        INSN_CODE (p) = -1;
      }
}
}
```

End it's over. Strength-reduction and loop-unrolling will be covered next week.