



IBM

Upgrading your GCC port to use modern features

Michael Meissner
meissner@linux.vnet.ibm.com

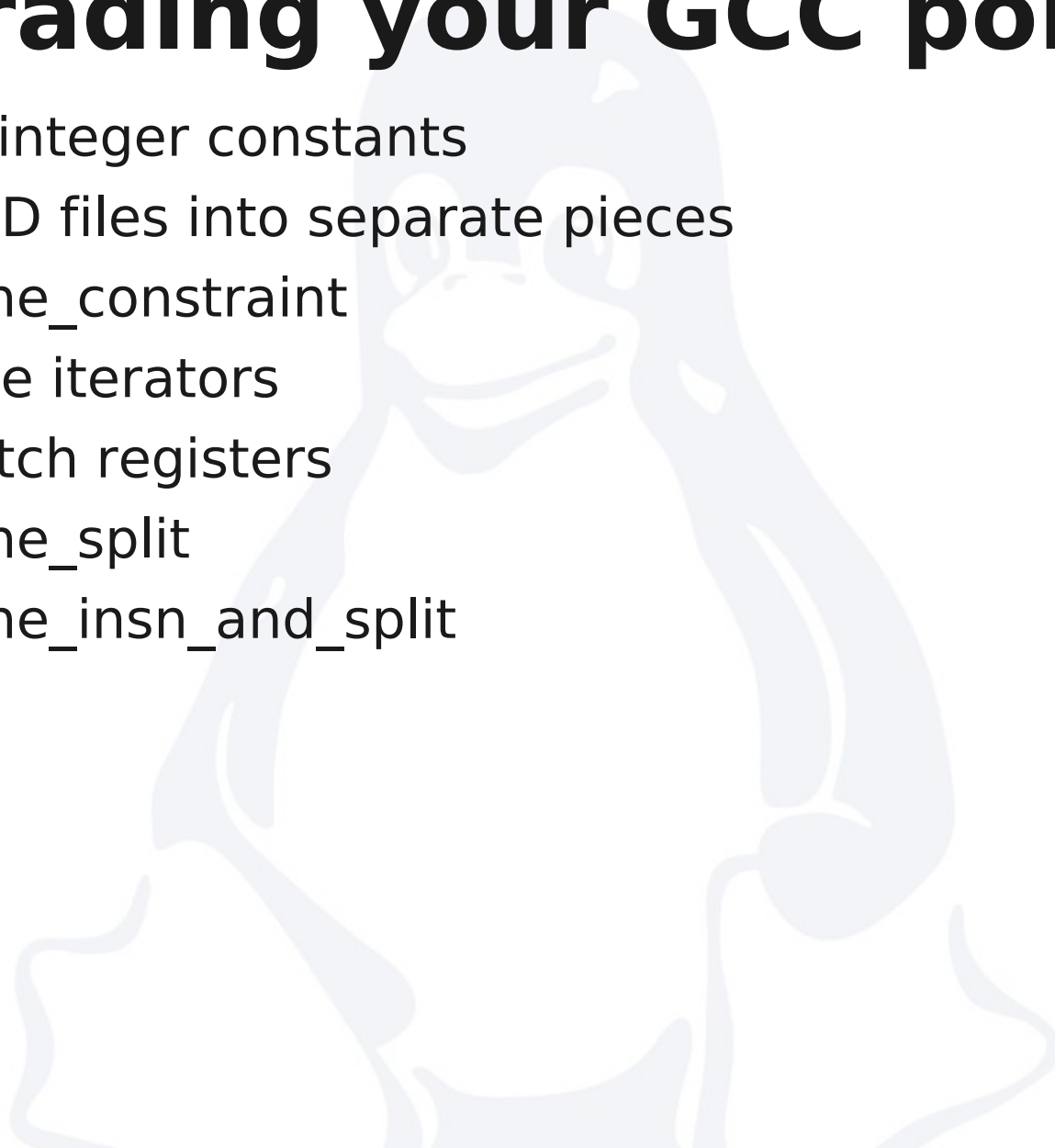
GCC summit 2010

IBM



Upgrading your GCC port

- Replacing integer constants
- Splitting MD files into separate pieces
- Using `define_constraint`
- Using mode iterators
- Using scratch registers
- Using `define_split`
- Using `define_insn_and_split`



Replacing Integer constants

- Originally for register numbers, unspec numbers, and other places that took register constants, you had to just enter the value numerically where you used it, which could lead to a lack of clarity or problems if you ever needed to change the number.
 - (clobber (reg:SI 23))
 - (unspec [(...)] 49)
- There are 3 ways to define constants now:
 - define_constants
 - define_c_enum
 - define_enum



define_constants

- define_constants is just like a #define
- Values are put in insn-constants.h which is included by tm.h
- For example:
 - (define_constants [(LR 65)])
 - (clobber (reg:SI LR))



define_c_enum

- `define_c_enum` is like a C enum.
- Two `c_enum`'s are special:
 - `Unspec`: associate name with `UNSPEC` for rtl dumps
 - `Unspecv`: associate name with `UNSPEC_VOLATILE` for rtl dumps
- For example:
 - `(define_c_enum "unspec" [UNSPEC_OP1])`
 - `(unspec [...] UNSPEC_OP1)`
- You can have multiple enums of the same class, in separate locations to allow for different md files to add their own enums.



define_enum

- define_enum is like define_c_enum
- define_enum values can be available in attribute lists (define_c_enum values can't be used in attributes)
- This is useful for two attributes that share the same values
- For example:
 - (define_enum “proc” [MOD1 MOD2 MOD3])
 - (define_attr “arch” “MOD1,MOD2,MOD3”)
 - (define_attr “tune” “MOD1,MOD2,MOD3”)



Splitting MD files

- It can be useful to split the MD definitions into multiple files
 - Group features into a file (like sse.md)
 - Scheduling for a specific processor (like athlon.md).
- For example:
 - (include “foo1.md”)
- If you use include, update the MD_INCLUDES make variable in your t-* file, so that dependencies are correct:
 - MD_INCLUDES += \$(src)/config/foo/foo1.md



Using define_constraint

- In the old days, you had to specify the constraints in the <machine>.h file:
 - REG_CLASS_FROM_LETTER: Return a regclass for a given letter.
 - CONST_OK_FOR_LETTER_P: Return true if the 'l'..'P' constraints were valid CONST_INT constants.
 - CONST_DOUBLE_OK_FOR_LETTER_P: return true if 'F' and 'G' constraints were valid CONST_DOUBLE constants.
 - EXTRA_CONSTRAINT, EXTRA_MEMORY_CONSTRAINT, EXTRA_ADDRESS_CONSTRAINT: Return true if a RTL value matches a constraint letter



Using define_constraint #2

- Various limits due to use of single letter for the constraint:
 - 2 floating point constraints
 - 8 integer constraints
 - Various standard constraints reduces the number of constraints that are possible
- Standard macros exanded to be able to take multiple letters:
 - REG_CLASS_FROM_CONSTRAINT
 - CONST_OK_FOR_CONSTRAINT_P
 - CONST_DOUBLE_OK_FOR_CONSTRAINT_P
 - EXTRA_CONSTRAINT_STR
- Better to define these in md file.



Using define_constraint #3

- For instance:
 - `#define REG_CLASS_FROM_LETTER(C) (((C) = 'f') ? FPR_REGS : ((C) = 'a') ? ACC_REGS : NO_REGS`
 - `#define EXTRA_MEMORY_CONSTRAINT(C,S) ((C) == 'P')`
- To:
 - `(define_register_constraint "f" "FPR_REGS" "Floating point register constraint")`
 - `(define_register_constraint "a" "ACC_REGS" "Accumulator register constraint")`
 - `(define_memory_constraint "P" "P memory operand" (match_operand 0 "p_operand"))`



Using define_constraint #4

- You can go beyond a single letter constraint by using a common prefix for constraints.
- For example:
 - (define_register_constraint “xf” “FPR_REGS”
“Floating point register constraint”)
 - (define_register_constraint “xa” “ACC_REGS”
“Accumulator register constraint”)

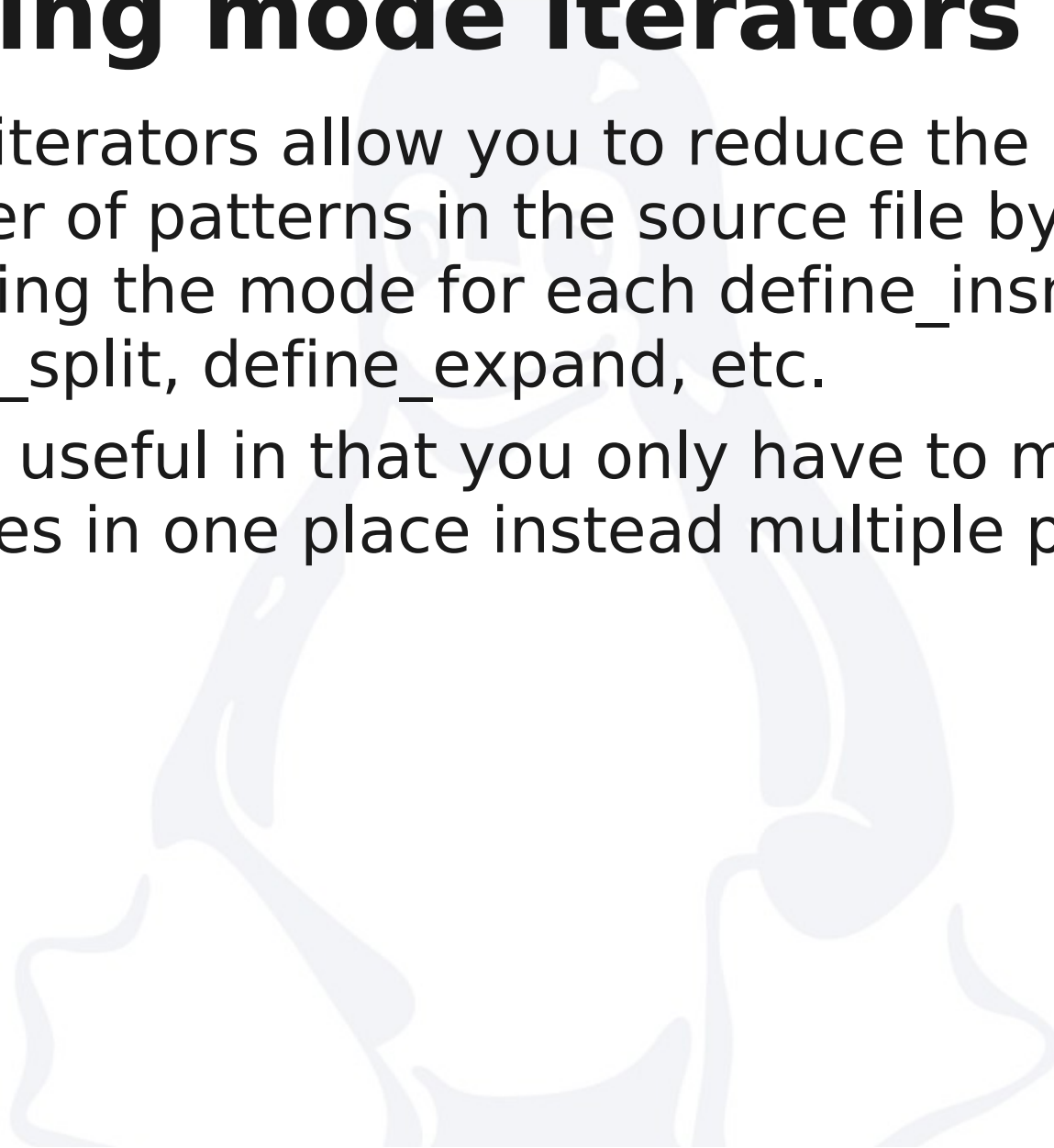
Using define_constraint #3

- Define_register_constraint: Create new register constraints.
- Define_memory_constraints: Create new memory constraints.
- Define_address_constraints: Create new address constraints.
- By convention, constraints should go in constraints.md.



Using mode iterators

- Mode iterators allow you to reduce the number of patterns in the source file by replacing the mode for each `define_insn`, `define_split`, `define_expand`, etc.
- This is useful in that you only have to make changes in one place instead multiple places.



Mode iterators example

```
(define_insn "addsf3"  
  [(set (match_operand:SF 0 "r_op" "=f")  
        (plus:SF  
          (match_operand:SF 1 "r_op" "f")  
          (match_operand:SF 2 "r_op" "f")))]  
  ""  
  "addf %0,%1,%2")
```

```
(define_insn "addddf3"  
  [(set (match_operand:DF 0 "r_op" "f")  
        (plus:SF  
          (match_operand:DF 1 "r_op" "f")  
          (match_operand:DF 2 "r_op" "f")))]  
  ""  
  "addd %0,%1,%2")
```



Mode iterators PRINT_OPERAND

```
#define PRINT_OPERAND(FILE, OP, C)\
do {\
    enum machine_mode mode\
        = GET_MODE (OP); \
    switch ((C))\
    {\
        case 'x':\
            if (mode == SFmode)\
                putc ('f', FILE);\
            else if (mode == DFmode)\
                putc ('d', FILE);\
            else\
                gcc_unreachable ();\
            break;\
        default:\
            gcc_unreachable ();\
    }\
    while (0)
```

/* In a constraints.md file included
from the tm.md file. */

(define_mode_iterator F [SF DF])

```
(define_insn "add<mode>3"  
  [(set (match_operand:<F> 0 "r_op" "=f")  
    (plus:<F>  
      (match_operand:<F> 1 "r_op" "f")  
      (match_operand:<F> 2 "r_op" "f")))]  
  ""  
  "add%x0 %0,%1,%2")
```



Mode iterators Expansion

```
(define_insn "addsf3"  
  [(set (match_operand:SF 0 "r_op" "=f")  
        (plus:SF  
          (match_operand:SF 1 "r_op" "f")  
          (match_operand:SF 2 "r_op" "f")))]  
  ""  
  "add%x0 %0,%1,%2")
```

```
(define_insn "addddf3"  
  [(set (match_operand:DF 0 "r_op" "=f")  
        (plus:DF  
          (match_operand:DF 1 "r_op" "f")  
          (match_operand:DF 2 "r_op" "f")))]  
  ""  
  "add%x0 %0,%1,%2")
```



Using mode attributes

```
(define_mode_iterator F [SF DF])  
(define_mode_attr F_suffix [(SF "f") (DF "d")])
```

```
(define_insn "add<mode>3"  
  [(set (match_operand:<F> 0 "r_op" "=f")  
        (plus:<F>  
          (match_operand:<F> 1 "r_op" "f")  
          (match_operand:<F> 2 "r_op" "f")))]  
  ""  
  "add<F_suffix> %0,%1,%2")
```

:: The <F_suffix> would get replaced by 'f' when the mode is
:: SFmode and 'd' when the mode is DFmode.

Mode iterators conditions

```
(define_mode_iterator F  
  [(SF "TARGET_SFMODE")  
   (DF "TARGET_DFMODE")])
```

```
(define_mode_attr F_suffix [(SF "f") (DF "d")])
```

```
(define_insn "add<mode>3"  
  [(set (match_operand:<F> 0 "r_op" "=f")  
        (plus:<F>  
          (match_operand:<F> 1 "r_op" "f")  
          (match_operand:<F> 2 "r_op" "f")))]  
  ""  
  "add<F_suffix> %0,%1,%2")
```

Nested mode iterators

```
(define_mode_iterator F [SF DF])  
(define_mode_iterator I [SI DI])  
  
(define_mode_attr F_s [(SF "f") (DF "d")])  
  
(define_mode_attr I_s [(SI "i") (DI "l")])  
  
(define_insn "float<l:mode><F:mode>2"  
  [(set (match_operand:<F> 0 "r_op" "=f")  
        (float:<F>  
          (match_operand:<l> 1 "i_op" "d")))]  
  ""  
  "cvt<l:I_s><F:F_s> %0,%1")
```



Using scratch registers

- Many times a port needs to allocate some extra registers for an operation.

```
(define_expand "adddi3"  
  [(set (match_operand:DI 0 "i_op" "")  
        (plus:DI  
          (match_operand:DI 1 "i_op" "")  
          (match_operand:DI 2 "i_op" "")))]  
  "")
```

```
{  
  rtx t = gen_reg_rtx (SImode);  
  emit_insn (gen_adddi3_int (operands[0],  
                             operands[1], operands[2], t));  
  DONE;  
})
```

```
(define_insn "adddi3_int"  
  [(set (match_operand:DI 0 "i_op" "=d")  
        (plus:DI  
          (match_operand:DI 1 "i_op" "d")  
          (match_operand:DI 2 "i_op" "d")))]  
  (clobber  
    (match_operand:SI 3 "i_op" "&d"))]  
  "")
```



Using match_dup

```
(define_expand "adddi3"  
  [(parallel  
    [(set (match_operand:DI 0 "i_op" "")  
      (plus:DI  
        (match_operand:DI 1 "i_op" "")  
        (match_operand:DI 2 "i_op" "")))]  
    (clobber (match_dup 3)))]  
  ""  
{  
  operands[3] = gen_reg_rtx (SImode);  
})
```

```
(define_insn "*adddi3_int"  
  [(set (match_operand:DI 0 "i_op" "=d")  
    (plus:DI  
      (match_operand:DI 1 "i_op" "d")  
      (match_operand:DI 2 "i_op" "d")))]  
  (clobber  
    (match_operand:SI 3 "i_op" "&d"))]  
  "#")
```



Using match_scratch

```
(define_insn "adddi3"  
  [(set (match_operand:DI 0 "i_op" "=d")  
        (plus:DI  
          (match_operand:DI 1 "i_op" "d")  
          (match_operand:DI 2 "i_op" "d")))  
    (clobber (match_scratch:SI 3 "&d"))]  
  ""  
  "#")
```



Match_scratch with combiner

- One advantage of using match_scratch is that the combiner can delete or add new scratch registers as needed to make a viable pattern.
- Consider a machine that normally needs a scratch register to do 64-bit integer adds, but doesn't need the scratch register if you are adding a 32-bit integer to the 64-bit integer. The combiner will combine the define_expand and the plus, and it will delete the clobber of the scratch register.



Match_scratch with combiner #2

```
(define_insn "adddi3"  
  [(set (match_operand:DI 0 "i_op" "=d")  
        (plus:DI  
          (match_operand:DI 1 "i_op" "d")  
          (match_operand:DI 2 "i_op" "d")))  
    (clobber (match_scratch:SI 3 "=&d"))]  
  ""  
  "#")
```

```
(define_insn "adddi3_s32"  
  [(set (match_operand:DI 0 "i_op" "=d")  
        (plus:DI  
          (match_operand:DI 1 "i_op" "0")  
          (sign_expand:DI  
            (match_operand:SI 1 "i_op" "d"))))]  
  ""  
  "#")
```



Omitting the scratch reg.

- You can use **X** for a scratch registers that isn't need for an alternative. For example, adding an integer constant.

```
(define_insn "adddi3"  
  [(set (match_operand:DI 0 "i_op" "=d,d")  
    (plus:DI  
      (match_operand:DI 1 "i_op" "d,d")  
      (match_operand:DI 2 "i2_op" "d,i")))  
    (clobber (match_scratch:SI 3 "&d,X")))]  
  ""  
  "#")
```



Using `define_split`

- `Define_split` takes an `insn` and splits it into 2 or more simpler `insns`.
- In the original GCC, there was no `define_split` and ports needed to emit the separate assembly instructions.
- When `define_split` was added, it was only run before the two scheduling passes, and in final when the `asm` string is “#”.
- Now `define_split` is run multiple times, before register allocation and after.



Define_split example

```
(define_split
  [(set (match_operand:DI 0 "i_op" "")
        (plus:DI
          (match_operand:DI 1 "i_op" "")
          (match_operand:DI 2 "i_op" "")))
    (clobber (match_scratch:SI 3 ""))]
  "reload_completed"
  [(set (match_dup 4) ; add high part
        (plus:SI (match_dup 5)
                  (match_dup 6)))
    (set (match_dup 7) ; add low part
        (plus:SI (match_dup 8)
                  (match_dup 9)))
    (set (match_dup 3) ; set carry
        (ult:SI (match_dup 7)
                 (match_dup 9)))
    (set (match_dup 4) ; add carry
        (plus:SI (match_dup 4)
                  (match_dup 3)))]
  ""
  {
    operands[4] = gen_highpart (SImode, operands[0]);
    operands[5] = gen_highpart (SImode, operands[1]);
    operands[6] = gen_highpart (SImode, operands[2]);
    operands[7] = gen_lowpart (SImode, operands[0]);
    operands[8] = gen_lowpart (SImode, operands[1]);
    operands[9] = gen_lowpart (SImode, operands[2]);
  })
```



Define_split before reload

- If define_split is run before register allocation, you can allocate pseudos so the insn can be split.

```
"      ;; was reload_completed
{
  if (GET_CODE (operands[3]) == SCRATCH)
    operands[3] = gen_reg_rtx (SImode);

  operands[4] = gen_highpart (SImode, operands[0]);
  operands[5] = gen_highpart (SImode, operands[1]);
  operands[6] = gen_highpart (SImode, operands[2]);
  operands[7] = gen_lowpart (SImode, operands[0]);
  operands[8] = gen_lowpart (SImode, operands[1]);
  operands[9] = gen_lowpart (SImode, operands[2]);
})"
```



Allocating stack

- I discovered that you can allocate stack if the `define_split` is run before register allocation.

```
(define_split
  [(set (match_operand:DF 0 "f_op" "=d")
        (float:DF (match_operand:SI 1 "i_op" "r"))))
   (clobber (match_scratch:DI 2 "=d"))]
  "TARGET_LFIWAX && can_create_pseudo_p ()"
  [(pc)]
  ""
  {
    if (GET_CODE (operands[2]) == SCRATCH)
      operands[2] = gen_reg_rtx (DImode);
    if (MEM_P (operands[1]))
      emit_insn (gen_lfiwax (operands[2], operands[1]));
    else
      {
        rtx stack = assign_stack_temp (SImode,
                                       GET_MODE_SIZE (SImode), 0);
        emit_move_insn (stack, operands[1]);
        emit_insn (gen_lfiwax (operands[2], stack));
      }
  }
  emit_insn (gen_floatdidf2 (operands[0], operands[1]));
  DONE;
})
```



define_insn_and_split

```
(define_insn_and_split "adddi3"
  [(set (match_operand:DI 0 "i_op" "&d")
        (plus:DI
          (match_operand:DI 1 "i_op" "d")
          (match_operand:DI 2 "i_op" "d"))))
  (clobber (match_scratch:SI 3 "&d"))]
  "" ; condition for insn
  "#" ; asm code, usually "#"
  "" ; conditional for the split
  [(set (match_dup 4) ; add high part
        (plus:SI (match_dup 5)
                  (match_dup 6)))
   (set (match_dup 7) ; add low part
        (plus:SI (match_dup 8)
                  (match_dup 9)))
   (set (match_dup 3) ; set carry
        (ult:SI (match_dup 7)
                 (match_dup 9)))
   (set (match_dup 4) ; add carry
        (plus:SI (match_dup 4)
                  (match_dup 3)))]
  ""
  {
    if (GET_CODE (operands[3]) == SCRATCH)
      operands[3] = gen_reg_rtx (SImode);

    operands[4] = gen_highpart (SImode, operands[0]);
    operands[5] = gen_highpart (SImode, operands[1]);
    operands[6] = gen_highpart (SImode, operands[2]);
    operands[7] = gen_lowpart (SImode, operands[0]);
    operands[8] = gen_lowpart (SImode, operands[1]);
    operands[9] = gen_lowpart (SImode, operands[2]);
  })
```

