Upgrading your GCC port to use modern features

Michael Meissner
IBM
`meissner@linux.vnet.ibm.com`

Abstract

Some of the GCC backends were written years ago when the features for the md files were rather sparse. Even if a backend is more modern, it might have been cloned from an older backend. In this talk, I will cover various features that make writing the md file easier today than it was in the ancient past.

1 Using direct integer constants

Integer constants are sometimes used by ports for register and for **UNSPEC** numbers directly, such as:

```
(clobber (reg:SI 65))
(unspec ... 100)
```

This can reduce legibility and it can be a maintenance problem because you have to make sure you catch all instances of the number if you change the number. There are several different ways that you can define alphanumeric labels to use instead of integer constants.

1.1 define_constants

The **define_constants** operation is equivalent to `#define` in C to define a constant in the program.

```
(define_constants [(LR 65)])
(clobber (reg:SI LR))
```

1.2 define_c_enum

The **define_c_enum** operation is equivalent to `enum` in C to define an enumeration.

```
(define_c_enum "unspec" [UNSPEC_OP1])
(unspec ... UNSPEC_OP1)
```

Some enumeration names have special significance to GCC:

- If an enumeration called **unspecv** is defined, GCC will use it when printing out *unspec_volatile* expressions.
- If an enumeration called **unspec** is defined, GCC will use it when printing out *unspec* expressions.

1.3 define_enum

The **define_enum** operation is like **define_c_enum**. However, unlike **define_c_enum**, the enumerations defined by **define_enum** can be used in attribute specifications.

2 Using define_predicate

In the old days, you defined predicates that you would use with the *match_operand* in the C code, and defined the prototype in the headers to use. Now you can define the predicates in the md file instead of the C code:

```
(define_predicate "s5bit_operand"
  (and (match_code "const_int")
       (match_test
         "IN_RANGE (INTVAL (op),
                    -16, 15)")))
```

Most ports (except for *arc*, *crx*, and *pdp11*) have moved over to using **define_predicate**.

There is also **define_special_predicate** that is like **define_predicate**, except that genrecog will not warn about a **match_operand** with no mode if it has a predicate defined with **define_special_predicate**.

3 Splitting MD files into separate pieces

You can use the **include** directive to split files into various pieces. It can make it easier to edit if you have files organized logically, such as.

- You can use separate md files for target features, such as `sse.md` in the i386 port for *SSE* instructions.
- You can use separate md files for different machines scheduling characteristics, such as `athlon.md` and `atom.md` in the i386 port for *AMD athlon* and *Intel Atom* timing characteristics.

If you do use the **include** feature, be sure to modify your `t-*` file to set the **MD_INCLUDES** make variable to the list of md files that are included by your port so that when you edit the md files, the files that are created from the md files will get regenerated.

4 Using define_constraint

In the old days, ports would use the macros to define constraints:

- **REG_CLASS_FROM_LETTER**: Given a constraint character, return the register class for the constraint;
- **CONST_OK_FOR_LETTER_P**: Given a constraint letter ('*I*' ... '*P*'), and an integer value, return whether the value matches the constraint;
- **CONST_DOUBLE_OK_FOR_LETTER_P**: Given a constraint letter ('*G*' or '*H*'), and a *CONST_DOUBLE* operand, return whether the operand matches the constraint.
- **EXTRA_CONSTRAINT**: Given a constraint letter, and a RTL operand, return whether the operand matches the constraint.
- **EXTRA_MEMORY_CONSTRAINT**: A subset of **EXTRA_CONSTRAINT** constraints that reload should treat as memory locations.
- **EXTRA_ADDRESS_CONSTRAINT**: A subset of **EXTRA_CONSTRAINT** constraints that reload should treat as memory addresses.

Using these macros, a port is limited to the number of constraints it could define, and many of the letters were hard coded to specific types of constraints, such as the letters ('*I*' ... '*P*') were reserved for integer constants, and ('*G*' or '*H*') were floating point or long integer constants.

There are a parallel set of macros that were added a few years ago that take a string instead of a single letter, and the following ports use these macros, but don't yet use **define_constraint**: *frv*, and *m32c*. For most ports, it is probably better to switch to uses **define_constraint** rather than using the ...**CONSTRAINT** macros.

- **REG_CLASS_FROM_CONSTRAINT**
- **CONST_OK_FOR_CONSTRAINT_P**
- **CONST_DOUBLE_OK_FOR_CONSTRAINT_P**
- **EXTRA_CONSTRAINT_STR**

The constraint patterns give you more flexibity. If your port has more constraints than can be enumerated with the 52 lower and upper case letters, you can define similar constraints that begin with a common prefix letter. The *bfin*, *i386*, *m68k*, *mep*, *mips*, *rs6000*, and *s390* ports now define some constraints that take two letters.

If your port need more integer constant constraints other than the 8 that the original GCC provided for, you can define other constraints to be integer constants. Similarly if you need more than 2 constraints that target floating point constants, you can define other FP constant constraints. For example, the *bfin* port defines 21 different integer constant constraints.

The new constraint patterns are:

- **define_register_constraint**: Define a constraint for a register operand.
- **define_memory_constraint**: Define a constraint for a memory operand.
- **define_address_constraint**: Define a constraint that acts like an address.
- **define_constraint**: Define any other constraint.

If for instance, your port had two register constraints ('*f*' and '*a*'), one memory constraint ('*P*'), and an address constraint ('*Q*'), and one random constraint ('*R*'), you might have the following definitions.

```
#define REG_CLASS_FROM_LETTER(C) \
  (((C) == 'f') ? FPR_REGS : \
   ((C) == 'a') ? ACC_REGS : \
   NO_REGS)

#define EXTRA_CONSTRAINT(OP, C) \
  (((C) == 'P') \
     ? p_operand (OP, GET_MODE (OP))\
    : ((C) == 'Q') \
      ? q_operand (OP, GET_MODE (OP))\
    : ((C) == 'R') \
      ? r_operand (OP, GET_MODE (OP))\
    : 0)

#define EXTRA_MEMORY_CONSTRAINT(C,S)\
  ((C) == 'P')

#define EXTRA_ADDRESS_CONSTRAINT(C,S)\
  ((C) == 'Q')
```

You would recode this as:

```
(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"))

(define_memory_constraint "Q"
  "Q address operand"
  (match_operand 0 "q_operand"))

(define_constraint "R"
  "R random operand"
  (match_operand 0 "r_operand"))
```

If you are moving to using **define_constraint** because you have run out of constraint letters you would pick one letter prefix that isn't used, and define multiple constraints using two letter combinations:

```
(define_register_constraint "xf"
 "FPR_REGS"
 "Floating point register constraint")

(define_register_constraint "xa"
 "ACC_REGS"
 "Accumulator register constraint")
```

The following ports have not yet moved over to using **define_constraint**: *arc*, *cris*, *crx*, *fr30*, *h8300*, *iq2000*, *m68hc11*, *mcore*, *mmix*, *pdp11*, *score*, *stormy16*, and *v850*.

5 Using mode iterators

Mode iterators allow you to reduce the number of patterns that are written in the source file, by substituting different modes during expansion of the source, and can eliminate cut+paste errors. If you later decide to make a change, you can make it in one place, rather than having to change each of the identical patterns and potentially missing one.

While these examples use **define_insn**, you can also use mode iterators on **define_expand**, **define_split**, and **define_insn_and_split** operations.

5.1 Mode iterator example

For instance, if you have a machine with both single and double precision floating point, you might have two add patterns:

```
(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 "adddf3"
 [(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")
```

You could use a mode iterator to make this into one pattern.

5.2 Using PRINT_OPERAND with mode iterators

One method to generate `addf` for single precsion floating point and `addd` for double precsion floating point is to add a letter to the **PRINT_OPERAND** processing to elide the difference between the two instructions:

```
/* In your tm.h file */
#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")
```

This would get expanded as:

```
(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 "adddf3"
 [(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")
```

5.3 Using mode attributes

An alternative method of writing the two add instrucitons would be to use a *mode attribute* that contains text that is substituted for each mode. For example:

```
(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**.

```
(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 "adddf3"
 [(set (match_operand:DF 0 "r_op" "=f")
   (plus:DF
    (match_operand:DF 1 "r_op" "f")
    (match_operand:DF 2 "r_op" "f")))]
 ""
 "addd %0,%1,%2")
```

5.4 Using conditions on mode iterators

You can also put conditions on the mode iteration:

```
(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")
```

This would be equivalent to:

```
(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")))]
 "TARGET_SFMODE"
 "addf %0,%1,%2")

(define_insn "adddf3"
 [(set (match_operand:DF 0 "r_op" "=f")
   (plus:DF
    (match_operand:DF 1 "r_op" "f")
    (match_operand:DF 2 "r_op" "f")))]
 "TARGET_DFMODE"
 "addd %0,%1,%2")
```

5.5 Using nested mode iterators

You can also use two or more mode iterators together. For example, if you have 3 modes in the first iterator and 2 modes in the second, then the expansion would generate 6 different insns.

If we have 4 different convert instructions that handle the conversions from 32/64-bit integer to 32/64-bit floating point types, we could write this as:

```
(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<I:mode><F:mode>2"
 [(set (match_operand:<F> 0 "r_op" "=f")
   (float:<F>
    (match_operand:<I> 1 "i_op" "d")))]
 ""
 "cvt<I:I_s><F:F_s> %0,%1")
```

This would generate 4 insns:

- **floatsisf2** insn: Generates `cvtif` instruction.
- **floatsidf2** insn: Generates `cvtid` instruction.
- **floatdisf2** insn: Generates `cvtlf` instruction.
- **floatdidf2** insn: Generates `cvtld` instruction.

6 Using scratch registers

Many times, ports need extra registers to complete an operation. The traditional way of doing this is to use a **define_expand** operation, and allocate the registers inside of the **define_expand**.

6.1 Scratch register initial example

Here is an example of a 64-bit integer add operation, that needs a scratch register. Note, in this pattern, we use the `adddi3_int` generator to generate the actual pattern, and we use `DONE` to not generate the normal pattern in the **define_expand**. The clobber of the scratch register uses the ampersand ('*&*') in the constraints to make sure that the scratch register is different from one of the input or output registers.

```
(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"))]
 ""
 "#")
```

6.2 Scratch register with match_dup

A common method of rewriting this is to use the **match_dup** operation. The **match_dup** is not used as an argument to the call to the **define_expand** function, but it is expected to be defined in the C code that makes up the body the the **define_expand** operation before the

RTL is generated. In this case, we do not use the DONE operation to allow the **define_expand** to generate the default pattern after the C code finishes. We don't need a generator function for the real insn in this case.

```
(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"))]
 ""
 "#")
```

6.3 Scratch register using match_scratch

The RTL machinery includes a method to allocate scratch registers just for this insn. Unlike **match_operand**, the **match_scratch** operation does not take a predicate (register_operand is assumed for scratch registers).

Before register allocation, SCRATCH is used in the RTL expansion and after register allocation it is replaced by a REG. This can complicate splitting the insns. See the section on **define_split** for more details.

```
(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"))]
 ""
 "#")
```

6.4 Using the combiner with scratch registers

One advantage of using **match_scratch** is that the combiner can delete or add new scratch registers as needed to make a viable pattern. For example, 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 **sign_expand** and the **plus**, and it will delete the clobber of the scratch register.

```
(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"))))]
  ""
  "#")
```

6.5 Omitting the scratch register for an alternative

You can use the '*X*' constraint for an alternative that does not need the scratch register. For example, consider a machine that needs a scratch register when it is adding 2 64-bit registers, but does not need a scratch register when 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"))]
 ""
 "#")
```

7 Using define_split

The **define_split** operation takes an insn and splits it into separate parts. In the original GCC compiler, there was no split operation, and the backend would just emit multiple instructions in a single string. Splitting the instructions to their independent parts allows various passes that move instructions around like the scheduler or when filling branch delay slots to rearrange low level instructions.

It is important that the RTL contains all of the dependence information, and the split instructions don't rely on setting status flags that aren't described in the machine description. If you have such a flag (like a *carry flag*), you might not be able to use **define_split**.

When **define_split** was first added, splits occured at three times:

- Before the first scheduling pass (before reload) if `-fschedule-insns` was used;
- Before the second schelduing pass (after reload) if `-fschedule-insns2` was used;
- In `final` if the insn string was hash ('`#`').

In the current compiler, **define_split** is now always called several times:

- After the control flow has been finalized and before the first scheduling pass (this split is now always done);
- After the `reload` and `gcse2` passes (this split is always done);
- Just before the second scheduling pass if `-fschedule-insns2`;
- Just after creation of floating point stack registers on the i386 port;
- Just before shortening branches on machines that don't have the register stack;
- In final if the insn string was hash ('`#`').

7.1 Define_split example

As an example, consider a machine that generates multiple instructions to add a 64-bit integer. Because we are using **match_scratch** in this example, we need to delay the split until after reload. After the split, 4 instructions will be generated (add the upper registers, add the lower registers, generate the carry into the scratch register, and add in the carry).

```
(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_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)))]
 "
{
 rtx op0 = operands[0];
 rtx op1 = operands[1];
 rtx op2 = operands[2];

 operands[4]
  = gen_highpart (SImode, op0);
 operands[5]
  = gen_highpart (SImode, op1);
 operands[6]
  = gen_highpart (SImode, op2);

 operands[7]
  = gen_lowpart (SImode, op0);
 operands[8]
  = gen_lowpart (SImode, op1);
 operands[9]
  = gen_lowpart (SImode, op2);
}")
```

7.2 Define_split before register allocation

Since the first split pass is now always done, we can split the example before register allocation if we convert the `SCRATCH` registers into real pseudo registers. In older versions of GCC this was not possible, because the first split pass was not always run, and when it was run, you could not allocate pseudo registers. The current compiler is now more dynamic in that you can create pseudo registers later.

```
(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_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 ""))]
 ""
 [(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)))]
 "
{
 rtx op0 = operands[0];
 rtx op1 = operands[1];
 rtx op2 = operands[2];
 rtx op3 = operands[3];

 if (GET_CODE (op3) == SCRATCH)
  operands[3] = gen_reg_rtx (SImode);

 operands[4]
  = gen_highpart (SImode, op0);
 operands[5]
  = gen_highpart (SImode, op1);
 operands[6]
  = gen_highpart (SImode, op2);

 operands[7]
  = gen_lowpart (SImode, op0);
 operands[8]
  = gen_lowpart (SImode, op1);
 operands[9]
  = gen_lowpart (SImode, op2);
}")
```

7.3 Allocating memory with define_split

When I was reworking the powerpc floating point conversion routines, I discovered that not only can you allocate new pseudo registers during the first split pass, you can allocate new memory locations. On most powerpc machines, there is a separation between the general purpose registers, and the floating point registers, and you generally have to store the value from the general purpose register and load it up in the floating point register to do the convert. It is complicated by older machines not having a way to store 32-bit integer values after conversion, so the port allocated a stack temporary to move the values between the register sets. This allocation occured during the RTL expansion phase, and prevented some optimizations later. Here is a cut down example of the code that allocates stack temporaries if they are needed in the first split pass.

```
(define_insn "floatsidf2_lfiwax"
 [(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"
 "#")

(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_pseduo_p ()"
 [(pc)]
 "
{
 rtx dest = operands[0];
 rtx src = operands[1];
 rtx tmp = operands[2];
 if (GET_CODE (tmp) == SCRATCH)
  tmp = gen_reg_rtx (DImode);
 if (MEM_P (src))
  emit_insn (gen_lfiwax (tmp, src));
 else
  {
   rtx stack
    = assign_stack_temp (SImode,
         GET_MODE_SIZE (SImode), 0);
    emit_move_insn (stack, src);
    emit_insn (gen_lfiwax (tmp, stack));
   }
  }
  emit_insn (gen_floatdidf2 (dest, tmp));
  DONE;
}")
```

8 Using define_insn_and_split

The **define_insn_and_split** operation is a convenience operation that allows you to combine a **define_insn** and a **define_split** in one pattern. For example, using my 64-bit add example earlier, you could rewrite this to be:

```
(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)))]
 "
{
 rtx op0 = operands[0];
 rtx op1 = operands[1];
 rtx op2 = operands[2];
 rtx op3 = operands[3];

 if (GET_CODE (op3) == SCRATCH)
  operands[3] = gen_reg_rtx (SImode);

 operands[4]
  = gen_highpart (SImode, op0);
 operands[5]
  = gen_highpart (SImode, op1);
 operands[6]
  = gen_highpart (SImode, op2);

 operands[7]
  = gen_lowpart (SImode, op0);
 operands[8]
  = gen_lowpart (SImode, op1);
 operands[9]
  = gen_lowpart (SImode, op2);
}")
```