Workshop on Essential Abstractions in GCC

Introduction to Machine Descriptions

GCC Resource Center
(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

2 July 2012

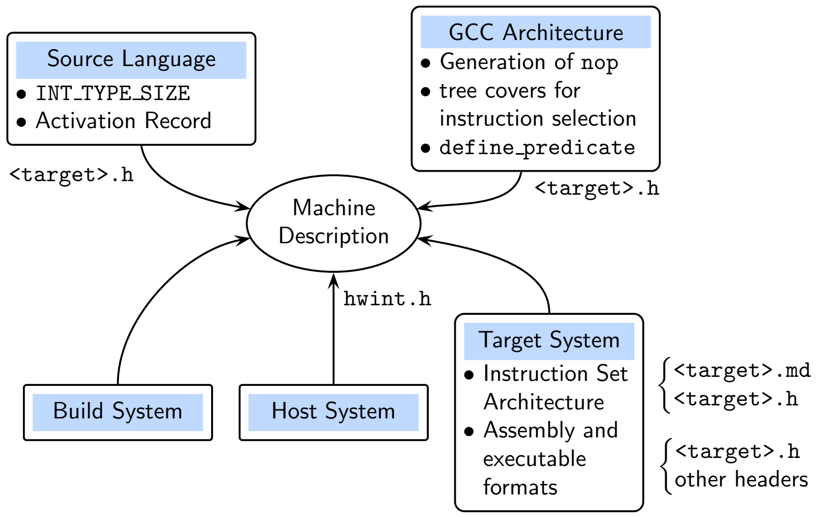
Outline

- Influences on GCC Machine Descriptions
- Organization of GCC Machine Descriptions
- Machine description constructs
- The essence of retargetability in GCC

Part 1

Influences on Machine Descriptions

Examples of Influences on the Machine Descriptions



Part 2

Organization of GCC MD

GCC Machine Descriptions

- Processor instructions useful to GCC
- Processor characteristics useful to GCC
- Target ASM syntax
- Target specific optimizations as IR-RTL $\rightarrow$ IR-RTL transformations
  (GCC code performs the transformation computations,
  MD supplies their *target patterns*)
  - Peephole optimizations
  - Transformations for enabling scheduling

Syntactic Entities in GCC MD

- Necessary Specifications
  - Processor instructions useful to GCC
    - One GIMPLE $\rightarrow$ One IR-RTL `define_insn`
    - One GIMPLE $\rightarrow$ More than one IR-RTL `define_expand`
  - Processor characteristics useful to GCC `define_cpu_unit`
  - Target ASM syntax part of `define_insn`
  - IR-RTL $\rightarrow$ IR-RTL transformations `define_split`
  - Target Specific Optimizations `define_peephole2`
- Programming Conveniences
  (eg. `define_insn_and_split`, `define_constants`, `define_cond_exec`, `define_automaton`)

File Organization of GCC MD

The GCC MD comprises of

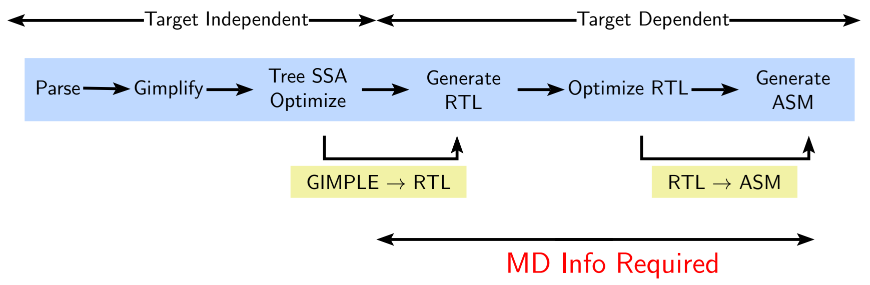
- `<target>.h`: A set of C macros that describe
  - HLL properties: e.g. `INT_TYPE_SIZE` to h/w bits
  - Activation record structure
  - Target Register (sub)sets, and characteristics
    (lists of read-only regs, dedicated regs, etc.)
  - System Software details: formats of assembler, executable etc.
- `<target>.md`: Target instructions described using MD constructs.

  `<target>.md`: Target instructions described using MD constructs.
  (Our main interest!)
- `<target>.c`: Optional, but usually required.
  C functions that implement target specific code
  (e.g. target specific activation layout).

Part 3

Essential Constructs in Machine Descriptions

The GCC Phase Sequence



The GCC Phase Sequence

Observe that

- RTL is a target specific IR
- GIMPLE → non strict RTL → strict RTL.
- Standard Pattern Name (SPN):
  "Semantic Glue" between GIMPLE and RTL
  - operator match + coarse operand match, and
  - refine the operand match
- Finally: Strict RTL ⇔ Unique target ASM string

Consider generating RTL expressions of GIMPLE nodes

- Two constructs available: `define_insn` and `define_expand`

Running Example

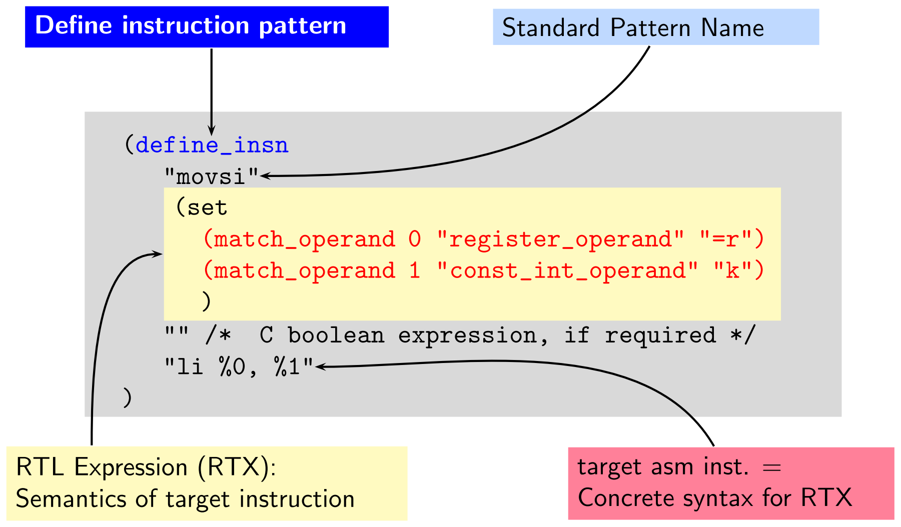
Consider a *data move* operation

- reads data from source location, and
- writes it to the destination location.
- GIMPLE node: `GIMPLE_ASSIGN`
- SPN: "`movsi`"

Some possible combinations are:

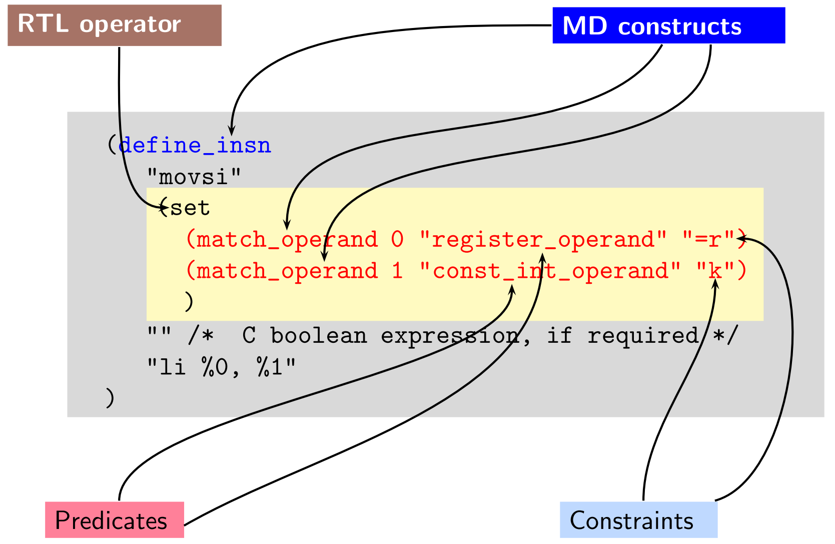
- Reg ← Reg : Register move
- Reg ← Mem : Load
- Reg ← Const : Load immediate
- Mem ← Reg : Store
- Mem ← Mem : Illegal instruction
- Mem ← Const : Illegal instruction

Specifying Target Instruction Semantics



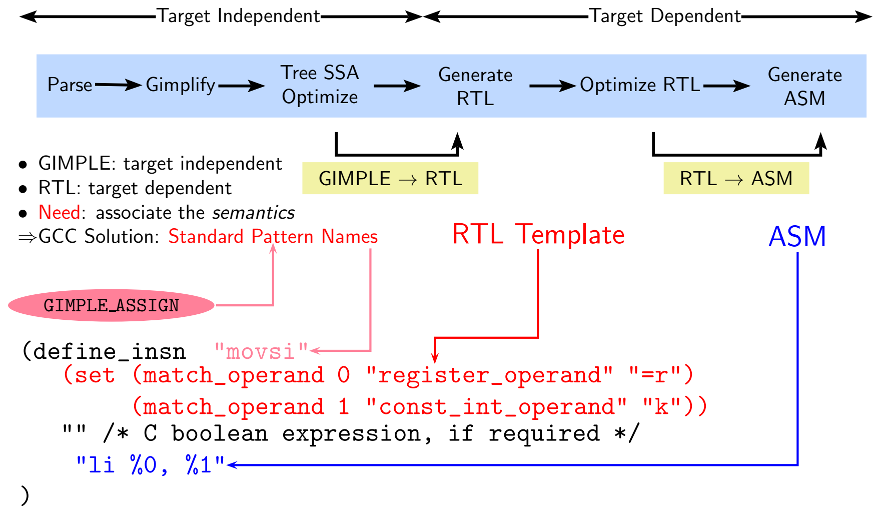
```
(define_insn
   "movsi"
   (set
     (match_operand 0 "register_operand" "=r")
     (match_operand 1 "const_int_operand" "k")
     )
   "" /*  C boolean expression, if required */
   "li %0, %1"
)
```

Specifying Target Instruction Semantics



```
(define_insn
   "movsi"
   (set
     (match_operand 0 "register_operand" "=r")
     (match_operand 1 "const_int_operand" "k")
     )
   "" /*  C boolean expression, if required */
   "li %0, %1"
)
```

Instruction Specification and Translation



- GIMPLE: target independent
- RTL: target dependent
- Need: associate the *semantics*

⇒GCC Solution: Standard Pattern Names

```
(define_insn  "movsi"
   (set (match_operand 0 "register_operand" "=r")
        (match_operand 1 "const_int_operand" "k"))
   "" /* C boolean expression, if required */
    "li %0, %1"
)
```

General Move Instruction

```
(define_insn "maybe_spn_like_movsi"
    (set  (match_operand 0 "general_operand" "")
          (match_operand 1 "general_operand" ""))
    ""
    "mov  %0, %1"
)
```

- This define_insn can generate data movement patterns of all combinations
- Even Mem → Mem is possible.
- We need a mechanism to generate more restricted data movement RTX instances!

The define_expand Construct

```
(define_expand "movsi"
 [(set (match_operand:SI 0 "nonimmediate_operand" "")
       (match_operand:SI 1 "general_operand" "")
   )]
 ""
 {
   if (GET_CODE (operands[0]) == MEM &&
       GET_CODE (operands[1]) != REG)
     if (can_create_pseudo_p())
         operands[1] = force_reg (SImode, operands[1]);
 }
)
```

Relationship Between <target>.md, <target>.c, and <target>.h Files

Example:

- Register class constraints are used in <target>.md file
- Register class is defined in <target>.h file
- Checks for register class are implemented in <target>.c file

Register Class Constraints in <target>.md File

```
;; Here z is the constraint character defined in
;; REG_CLASS_FROM_LETTER_P
;; The register $zero is used here.
(define_insn "IITB_move_zero"
   [(set
      (match_operand:SI 0 "nonimmediate_operand" "=r,m")
      (match_operand:SI 1 "zero_register_operand" "z,z")
    )]
   ""
   "@
   move \t%0,%1
   sw \t%1, %m0"
)
```

The Register Class letter code

Register Class specification in <target>.h File

```
/* From spim.h */
#define REG_CLASS_FROM_LETTER_P                         \
   reg_class_from_letter
enum reg_class                                          \
{                                                       \
        NO_REGS,              ZERO_REGS,                \
        CALLER_SAVED_REGS,    CALLEE_SAVED_REGS,        \
        BASE_REGS,            GENERAL_REGS,             \
        ALL_REGS,             LIM_REG_CLASSES           \
};

#define REG_CLASS_CONTENTS                              \
{0x00000000, 0x00000001, 0xff00ffff, 0x00ff0000, \
  0xf0000000, 0x0cfffff3, 0xffffffff}
```

The Register Classes

The Register Class Enumeration

The `<target>.c` File

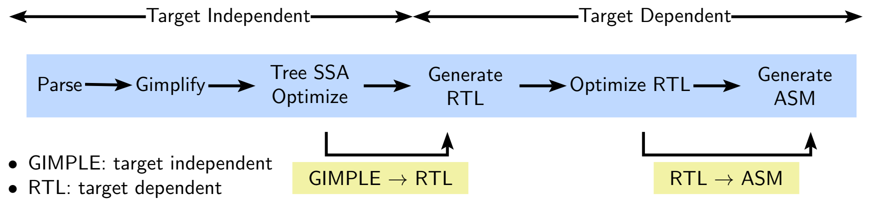
```
enum reg_class
reg_class_from_letter (char ch)
{
   switch(ch)
   {
   case 'b':return BASE_REGS;
   case 'x':return CALLEE_SAVED_REGS;
   case 'y':return CALLER_SAVED_REGS;
   case 'z':return ZERO_REGS;
   }
   return NO_REGS;
}
```

Get the enumeration from the Register class letter

Part 4

The Essence of Retargetability

Instruction Specification and Translation: A Recap



- GIMPLE: target independent
- RTL: target dependent
- Need: associate the *semantics*

⇒GCC Solution: Standard Pattern Names

RTL Template

ASM

GIMPLE_ASSIGN

```
(define_insn  "movsi"
   (set (match_operand 0 "register_operand" "=r")
        (match_operand 1 "const_int_operand" "k"))
   "" /* C boolean expression, if required */
    "li %0, %1"
)
```

Translation Sequence in GCC

```
(define_insn
   "movsi"
    (set
      (match_operand 0 "register_operand" "=r")
      (match_operand 1 "const_int_operand" "k")
      )
   "" /*  C boolean expression, if required */
   "li %0, %1"
)
```

Development

```
D.1283 = 10;  ⇒  (set
                     (reg:SI 58 [D.1283])
                     (const_int 10:  [0xa])
                  )                         ⇒  li $t0, 10
```

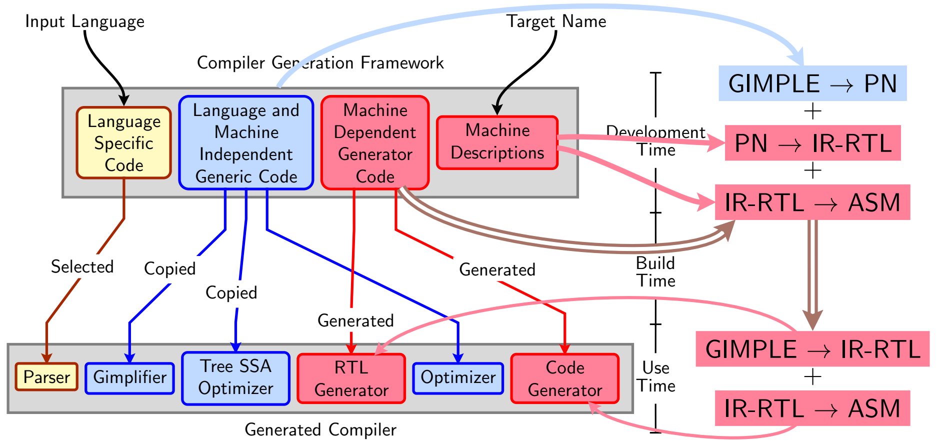
Use

The Essence of Retargetability

When are the machine descriptions read?

- During the build process
- When a program is compiled by gcc the information gleaned from machine descriptions is consulted

Retargetability Mechanism of GCC



Part 5

Summary

Summary

- GCC achieves retargetability by reading the machine descriptions and generating a back end customised to the machine descriptions
- Machine descriptions are influenced by:
  The HLLs, GCC architecture, and properties of target, host and build systems
- Writing machine descriptions requires:
  specifying the C macros, target instructions and any required support functions
- `define_insn` and `define_expand` are used to convert a GIMPLE representation to RTL