

Workshop on Essential Abstractions in GCC

Advanced Issues in Machine Descriptions

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

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



July 2010

Outline

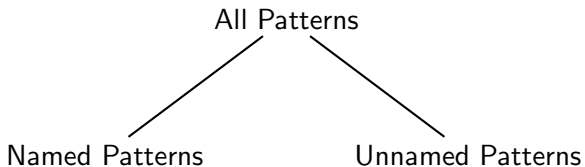
- Some details of MD constructs
 - ▶ On names of patterns in .md files
 - ▶ On the role of `define_expand`
 - ▶ On the role of constraints
 - ▶ Mode and code iterators
 - ▶ Defining attributes
 - ▶ Other constructs
- Improving machine descriptions and instruction selection
 - ▶ New constructs to factor out redundancy
 - ▶ Tree tiling based instruction selection



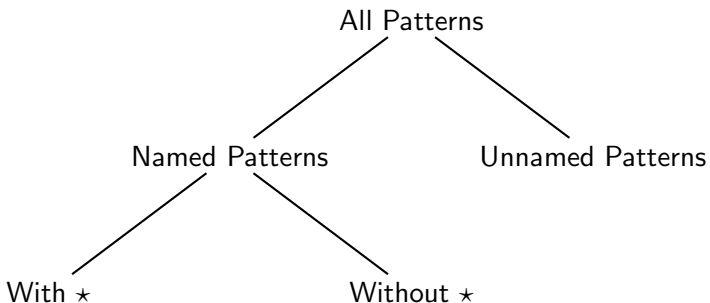
Part 1

Some Details of MD Constructs

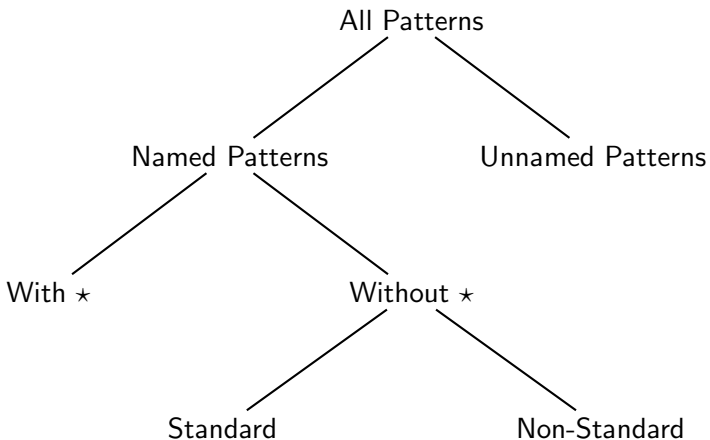
Pattern Names in .md File



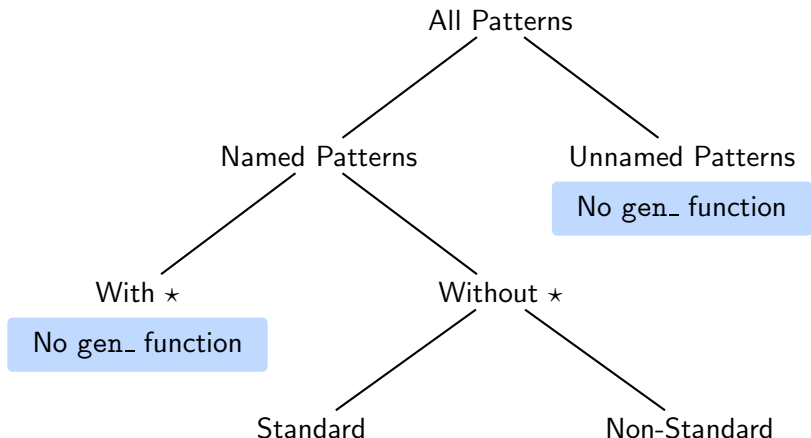
Pattern Names in .md File



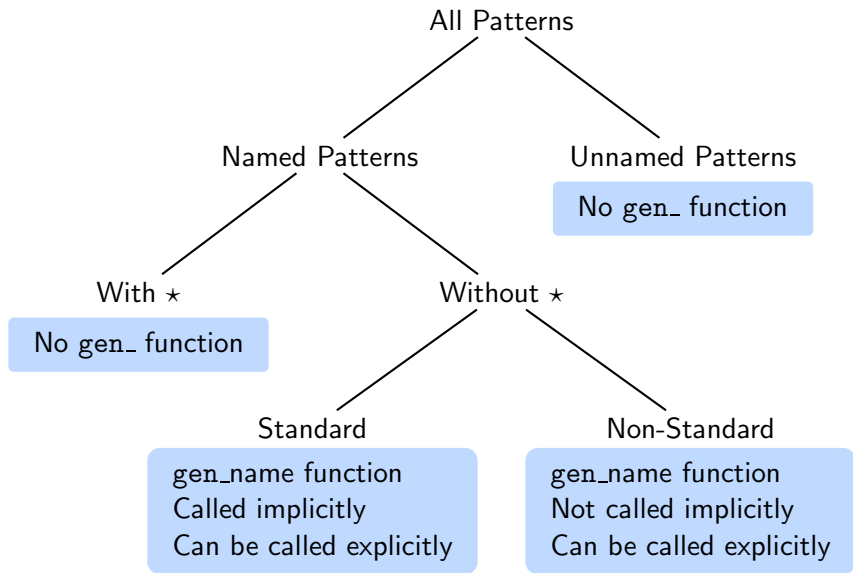
Pattern Names in .md File



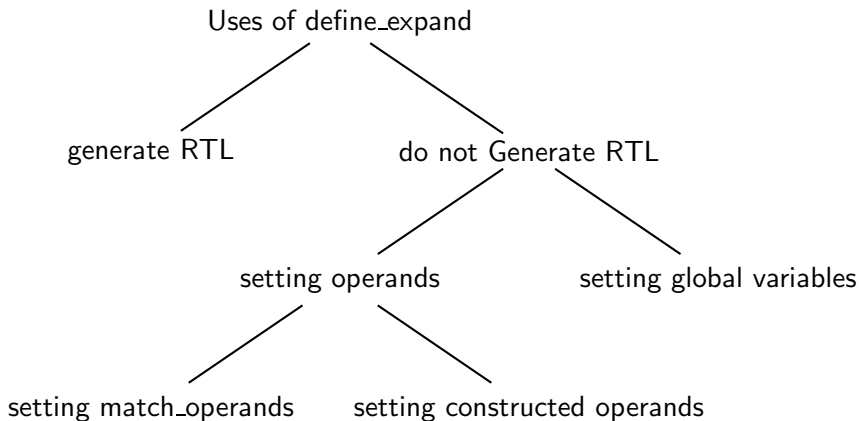
Pattern Names in .md File



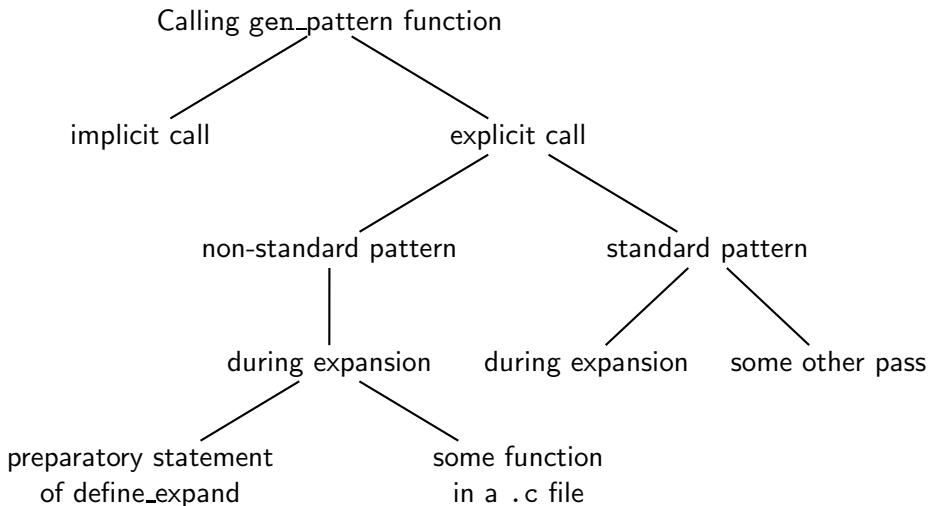
Pattern Names in .md File



Role of `define_expand`



Using `define_expand` for Generating RTL statements



Understanding Constraints

```
(define_insn ""  
  [(set (match_operand:SI 0 "general_operand" "=r")  
        (plus:SI (match_dup 0)  
                  (match_operand:SI 1 "general_operand" "r")))]  
  ""  
  "...")
```



Understanding Constraints

Constraints

```
(define_insn ""  
  [(set (match_operand:SI 0 "general_operand" "=r")  
        (plus:SI (match_dup 0)  
                  (match_operand:SI 1 "general_operand" "r")))]  
  ""  
  "...")
```



Understanding Constraints

Constraints

```
(define_insn ""  
  [(set (match_operand:SI 0 "general_operand" "=r")  
        (plus:SI (match_dup 0)  
                  (match_operand:SI 1 "general_operand" "r")))]  
  ""  
  "...")
```

- Reloading operands in the most suitable register class



Understanding Constraints

Constraints

```
(define_insn ""  
  [(set (match_operand:SI 0 "general_operand" "=r")  
        (plus:SI (match_dup 0)  
                  (match_operand:SI 1 "general_operand" "r")))]  
  ""  
  "...")
```

- Reloading operands in the most suitable register class
- Fine tuning within the set of operands allowed by the predicate



Understanding Constraints

Constraints

```
(define_insn ""  
  [(set (match_operand:SI 0 "general_operand" "=r")  
        (plus:SI (match_dup 0)  
                  (match_operand:SI 1 "general_operand" "r")))]  
  ""  
  "...")
```

- Reloading operands in the most suitable register class
- Fine tuning within the set of operands allowed by the predicate
- If omitted, operands will depend only on the predicates



Observations - Role of Constraints

Consider the following two instruction patterns:

- ```
(define_insn ""
 [(set (match_operand:SI 0 "general_operand" "=r")
 (plus:SI (match_dup 0)
 (match_operand:SI 1 "general_operand" "r")))]
 ""
 "...")
```

The destination and left operands must be identical

- ```
(define_insn ""  
  [(set (match_operand:SI 0 "general_operand" "=r")  
        (plus:SI (match_operand:SI 1 "general_operand" "z")  
                  (match_operand:SI 2 "general_operand" "r")))]  
  ""  
  "...")
```



Role of Constraints

- Consider an insn of the form

```
(insn n prev next
  (set (reg:SI 3)
    (plus:SI (reg:SI 6) (reg:SI 109)))
  ...))}
```
- Predicates of the first pattern does not match
- Constraints do not match for first operand of the second pattern



Role of Constraints

- Consider an insn of the form

```
(insn n prev next
  (set (reg:SI 3)
    (plus:SI (reg:SI 6) (reg:SI 109)))
  ...))}
```

- Predicates of the first pattern does not match
- Constraints do not match for first operand of the second pattern
- Reload pass generates additional insn to that the first pattern can be used

```
(insn n2 prev n
  (set (reg:SI 3) (reg:SI 6))
  ...)
(insn n n2 next
  (set (reg:SI 3)
    (plus:SI (reg:SI 3) (reg:SI 109)))
  ...)
```



Observations: Constraints

- `define_insns` patterns have operand predicates and constraints



Observations: Constraints

- `define_insns` patterns have operand predicates and constraints
- While generating the RTL from GIMPLE only the operand predicates are checked and the constraints are completely ignored



Observations: Constraints

- `define_insns` patterns have operand predicates and constraints
- While generating the RTL from GIMPLE only the operand predicates are checked and the constraints are completely ignored
- The RTL which is generated in the expander is modified in the reload pass to fulfil the constraint



Observations: Constraints

- `define_insns` patterns have operand predicates and constraints
- While generating the RTL from GIMPLE only the operand predicates are checked and the constraints are completely ignored
- The RTL which is generated in the expander is modified in the reload pass to fulfil the constraint
- This final form of RTL can be generated in expander by using constraints early



Handling Mode Differences

```
(define_insn "subsi3"  
  [(set (match_operand:SI 0 "register_operand" "=d")  
        (minus:SI (match_operand:SI 1 "register_operand" "d")  
                   (match_operand:SI 2 "register_operand" "d")))]  
  ""  
  "subu\t%0,%1,%2"  
  [(set_attr "type" "arith")  
   (set_attr "mode" "SI")])  
  
(define_insn "subdi3"  
  [(set (match_operand:DI 0 "register_operand" "=d")  
        (minus:DI (match_operand:DI 1 "register_operand" "d")  
                   (match_operand:DI 2 "register_operand" "d")))]  
  ""  
  "dsubu\t%0,%1,%2"  
  [(set_attr "type" "arith")  
   (set_attr "mode" "DI")])
```



Handling Mode Differences

```
(define_insn "subsi3"  
  [(set (match_operand:SI 0 "register_operand" "=d")  
        (minus:SI (match_operand:SI 1 "register_operand" "d")  
                  (match_operand:SI 2 "register_operand" "d")))]  
  ""  
  "subu\t%0,%1,%2"  
  [(set_attr "type" "arith")  
   (set_attr "mode" "SI")])  
  
(define_insn "subdi3"  
  [(set (match_operand:DI 0 "register_operand" "=d")  
        (minus:DI (match_operand:DI 1 "register_operand" "d")  
                  (match_operand:DI 2 "register_operand" "d")))]  
  ""  
  "dsubu\t%0,%1,%2"  
  [(set_attr "type" "arith")  
   (set_attr "mode" "DI")])
```



Mode Iterators: Abstracting Out Mode Differences

```
(define_mode_iterator GPR [SI (DI "TARGET_64BIT")])
(define_mode_attr d [(SI "") (DI "d")])
(define_insn "sub<mode>3"
  [(set (match_operand:GPR 0 "register_operand" "=d")
        (minus:GPR (match_operand:GPR 1 "register_operand" "d")
                    (match_operand:GPR 2 "register_operand" "d")))]
  ""
  "<d>subu\t %0,%1,%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "<MODE>")])
```



Handling Code Differences

```
(define_expand "bunordered"  
  [(set (pc) (if_then_else (unordered:CC (cc0) (const_int 0))  
                            (label_ref (match_operand 0 ""))  
                            (pc)))]  
  ""  
  { mips_expand_conditional_branch (operands, UNORDERED);  
    DONE;  
  })
```

```
(define_expand "bordered"  
  [(set (pc) (if_then_else (ordered:CC (cc0) (const_int 0))  
                            (label_ref (match_operand 0 ""))  
                            (pc)))]  
  ""  
  { mips_expand_conditional_branch (operands, ORDERED);  
    DONE;  
  })
```



Handling Code Differences

```
(define_expand "bunordered"  
  [(set (pc) (if_then_else (unordered:CC (cc0) (const_int 0))  
                           (label_ref (match_operand 0 ""))  
                           (pc)))]  
  ""  
  { mips_expand_conditional_branch (operands, UNORDERED);  
    DONE;  
  })
```

```
(define_expand "bordered"  
  [(set (pc) (if_then_else (ordered:CC (cc0) (const_int 0))  
                           (label_ref (match_operand 0 ""))  
                           (pc)))]  
  ""  
  { mips_expand_conditional_branch (operands, ORDERED);  
    DONE;  
  })
```



Code Iterators: Abstracting Out Code Differences

```
(define_code_iterator any_cond [unordered ordered])
(define_expand "b<code>"
  [(set (pc)
        (if_then_else (any_cond:CC (cc0)
                          (const_int 0))
                       (label_ref (match_operand 0 ""))
                       (pc)))]
  ""
  { mips_expand_conditional_branch (operands, <CODE>);
    DONE;
  })
```



Defining Attributes

- Classifications are need based
- Useful to GCC phases – e.g. pipelining

Property: Pipelining

Need: To classify target instructions

Construct: `define_attr`



Defining Attributes

- Classifications are need based
- Useful to GCC phases – e.g. pipelining

Property: Pipelining

Need: To classify target instructions

Construct: `define_attr`

`;; Instruction type.`

`(define_attr "type"`

`"other,multi, alu,alu1,negnot, ... str,cld, ..."`

`(const_string "other"))`



Defining Attributes

- Classifications are need based
- Useful to GCC phases – e.g. pipelining

Property: Pipelining

Need: To classify target instructions

Construct: `define_attr`

`;; Instruction type.`

`(define_attr "type"`

`"other,multi,alu,alu1,negnot, ... str,cld, ..."`

`(const_string "other"))`

Fields:

Attribute name,



Defining Attributes

- Classifications are need based
- Useful to GCC phases – e.g. pipelining

Property: Pipelining

Need: To classify target instructions

Construct: `define_attr`

`;; Instruction type.`

`(define_attr "type"`

`"other,multi, alu,alu1,negnot, ... str,cld, ..."`

`(const_string "other"))`

Fields:

Attribute name, all possible values,



Defining Attributes

- Classifications are need based
- Useful to GCC phases – e.g. pipelining

Property: Pipelining

Need: To classify target instructions

Construct: `define_attr`

`;; Instruction type.`

`(define_attr "type"`

`"other,multi, alu,alu1,negnot, ... str,cld, ..."`

`(const_string "other"))`

Fields:

Attribute name, all possible values, one of the possible values,



Defining Attributes

- Classifications are need based
- Useful to GCC phases – e.g. pipelining

Property: Pipelining

Need: To classify target instructions

Construct: `define_attr`

`:: Instruction type.`

`(define_attr "type"`

`"other,multi, alu,alu1,negnot, ... str,cld, ..."`

`(const_string "other"))`

Fields:

Attribute name, all possible values, one of the possible values, default.



Specifying Instruction Attributes

- **Optional field** of a `define_insn`
- For an i386, we choose to **mark** string instructions with the attribute value **str**

```
(define_insn "*strmovdi_rex_1"  
  [(set (mem:DI (match_operand:DI 2 ...))  
    "TARGET_64BIT && (TARGET_SINGLE_ ...)"  
    "movsq"  
    [(set_attr "type" "str")  
     ...  
     (set_attr "memory" "both")])])
```

NOTE

An instruction may have more than one attribute!



Using Attributes

```
(define_insn_reservation "pent_str" 12
  (and (eq_attr "cpu" "pentium")
        (eq_attr "type" "str") )
  "pentium-np*12")
```

Pipeline specification requires the CPU type to be “pentium” and the instruction type to be “str”



Some Other RTL Constructs

- `define_split`: Split complex insn into simpler ones
e.g. for better use of delay slots
- `define_insn_and_split`: A combination of `define_insn` and `define_split`
Used when the split pattern matches and insn exactly.
- `define_peephole2`: Peephole optimization over insns that substitutes insns. Run after register allocation, and before scheduling.
- `define_constants`: Use literal constants in rest of the MD.



Part 2

Improving Instruction Selection and Machine Descriptions

Improving Machine Descriptions and Instruction Selection

The Problems:

- Instruction selection algorithms are quite adhoc
- The specification mechanism for Machine descriptions is quite adhoc
- Adhoc design decisions



Improving Machine Descriptions and Instruction Selection

The Problems:

- Instruction selection algorithms are quite adhoc
 - ▶ Full tree matching instead of tree tiling
- The specification mechanism for Machine descriptions is quite adhoc
- Adhoc design decisions



Improving Machine Descriptions and Instruction Selection

The Problems:

- Instruction selection algorithms are quite adhoc
 - ▶ Full tree matching instead of tree tiling
- The specification mechanism for Machine descriptions is quite adhoc
 - ▶ Only syntax borrowed from LISP, neither semantics not spirit!
 - ▶ Non-composable rules
 - ▶ Mode and code iterator mechanisms are insufficient
- Adhoc design decisions



Improving Machine Descriptions and Instruction Selection

The Problems:

- Instruction selection algorithms are quite adhoc
 - ▶ Full tree matching instead of tree tiling
- The specification mechanism for Machine descriptions is quite adhoc
 - ▶ Only syntax borrowed from LISP, neither semantics not spirit!
 - ▶ Non-composable rules
 - ▶ Mode and code iterator mechanisms are insufficient
- Adhoc design decisions
 - ▶ Honouring operand constraints delayed to global register allocation
During GIMPLE to RTL translation, a lot of C code is required
 - ▶ Choice of insertion of NOPs



Design Flaws in Machine Descriptions

Multiple patterns with same structure

- Repetition of almost similar RTL expressions across multiple `define_insn` and `define_expand` patterns
 - ▶ Only Modes, Predicates, Constraints, Boolean Condition, or RTL Expression may differ
 - ▶ One RTL expression may appear as a sub-expression of some other RTL expression
- Repetition of C code along with RTL expressions in these patterns.



Consequence of Design Flaws in Machine Descriptions

- The machine descriptions are too verbose, detailed, repetitive and require a lot of C code
- A compiler developer needs to visualize and specify meaningful combinations of instructions for generating good quality code
- The machine descriptions are difficult to construct, understand, maintain, and enhance
- GCC has become a **hacker's paradise** instead of a clean, production quality compiler generation framework



Insufficient Iterator Mechanism

- Iterators can not be used across `define_insn`, `define_expand`, `define_peephole2` and other patterns
- Defining iterator attribute for each varying parameter becomes tedious.
- For same set of modes and rtx codes change in other fields of pattern makes use of iterators impossible
- Mode and code attributes can not be defined for operator or operand number, name of the pattern.
- Patterns with different RTL template share attribute value vector for which iterators can not be used.



Many Similar Patterns Cannot be Combined

```
(define_expand "iordi3"  
  [(set (match_operand:DI 0 "nonimmediate_operand" "")  
        (ior:DI (match_operand:DI 1 "nonimmediate_operand" "")  
                (match_operand:DI 2 "x86_64_general_operand" "")))]  
  (clobber (reg:CC FLAGS_REG))]  
  "TARGET_64BIT"  
  "ix86_expand_binary_operator (IOR, DI_mode, operands); DONE;")  
  
(define_insn "*iordi_1_rex64"  
  [(set (match_operand:DI 0 "nonimmediate_operand" "=rm,r")  
        (ior:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0")  
                (match_operand:DI 2 "x86_64_general_operand" "re,rme" )))]  
  (clobber (reg:CC FLAGS_REG))]  
  "TARGET_64BIT"  
  && ix86_binary_operator_ok (IOR, DI_mode, operands)"  
  "or{q}\t{%2, %0|%0, %2}"  
  [(set_attr "type" "alu")  
   (set_attr "mode" "DI")])
```



Step 1: Avoiding Verbosity in Machine Description

- New constructs to facilitate more concise machine descriptions
 - ▶ `define_rtltemplate`
Introduces non-terminals for common RTL expressions instead of rewriting them in each `define_insn` or `define_expand` pattern
 - ▶ `define_code`
Introduces non-terminals for C/Assembly code instead of rewriting them in each `define_insn` or `define_expand` pattern
 - ▶ `define_pattern`
Allows specification of multiple `define_insn` and `define_expand` sharing RTL template, assembly template, or C code



Step 1: Avoiding Verbosity in Machine Description

- New constructs to facilitate more concise machine descriptions
 - ▶ `define_rtltemplate` defining
Introduces non-terminals for common RTL expressions instead of rewriting them in each `define_insn` or `define_expand` pattern
 - ▶ `define_code` defining
Introduces non-terminals for C/Assembly code instead of rewriting them in each `define_insn` or `define_expand` pattern
 - ▶ `define_pattern` instantiating
Allows specification of multiple `define_insn` and `define_expand` sharing RTL template, assembly template, or C code
- Generate existing machine descriptions from new descriptions
 - ⇒ No change in GCC source
 - ⇒ Incremental changes with gradual transition to new descriptions
 - ⇒ Non-disruptive transition



Improvement Statistics for 'i386.md'

Machine description file i386.md is rewritten. **5720** lines are reduced from current MD specification for 'i386.md(21474 lines). Ignoring the comments this reduction is **28%** of the complete MD file.

Instruction Group	Instruction Pattern count	define_rtltemplate count
Arithmetic instructions	154	50
Control flow and data move instructions	626	169
Logical and relational instructions	212	56
Shift and rotate instructions	311	67
Total	1303	342

Improvement statistics in terms of reduction in RTL templates



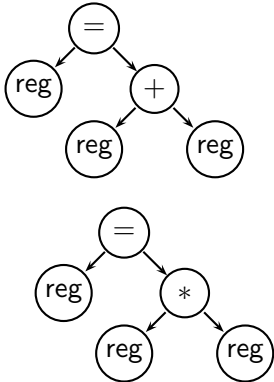
Step 2: Improving Instruction Selection

- Since rules become composable, tree tiling based instruction selection algorithms can be used
Currently rules are non-composable and GCC uses full tree matching algorithm



Full Tree Matching

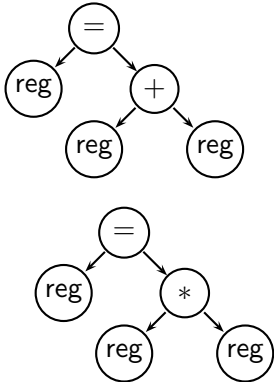
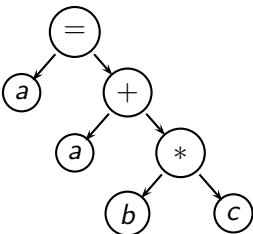
Instructions are viewed as independent non-composable rules

Instructions	Subject Tree	Modified Trees
		



Full Tree Matching

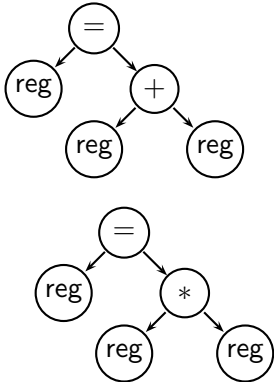
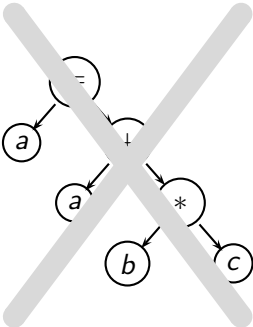
Instructions are viewed as independent non-composable rules

Instructions	Subject Tree	Modified Trees
		



Full Tree Matching

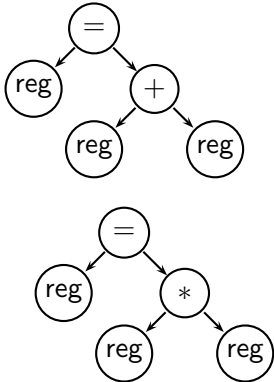
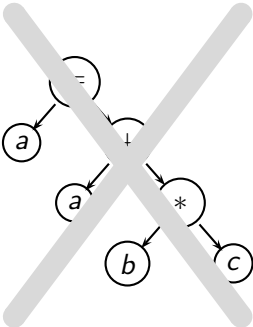
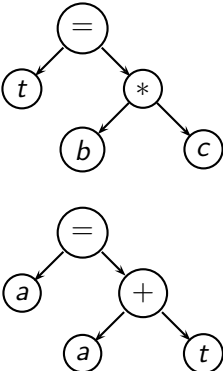
Instructions are viewed as independent non-composable rules

Instructions	Subject Tree	Modified Trees
		



Full Tree Matching

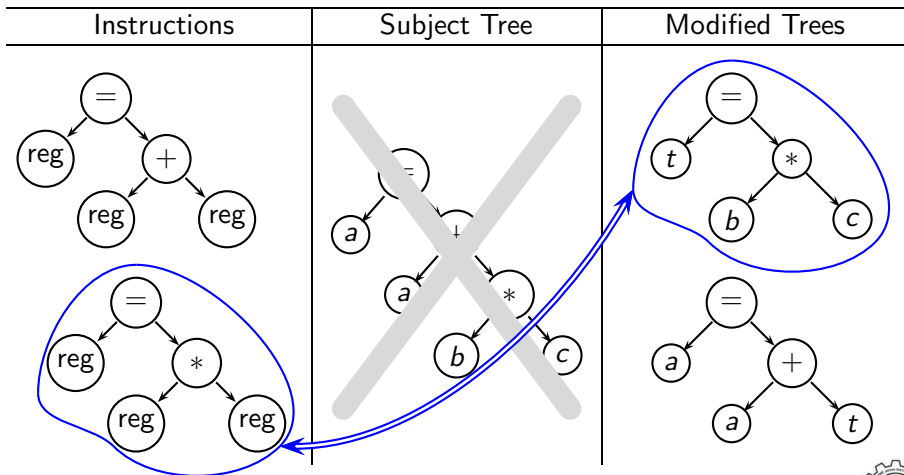
Instructions are viewed as independent non-composable rules

Instructions	Subject Tree	Modified Trees
		



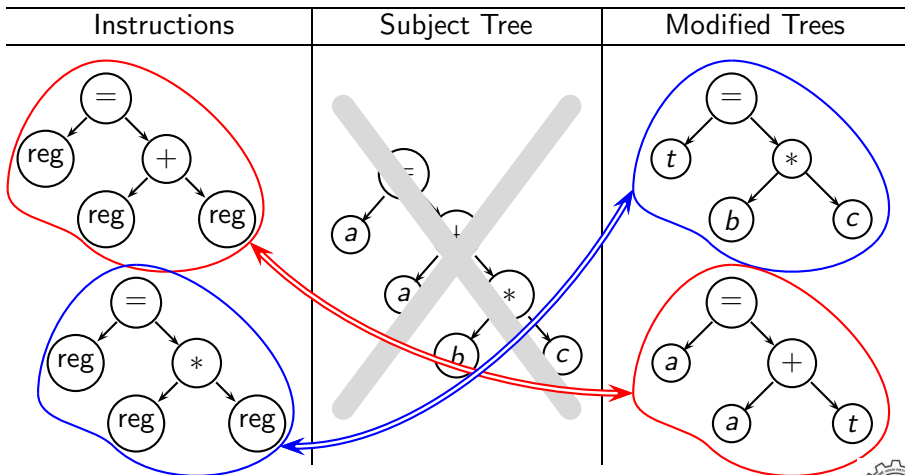
Full Tree Matching

Instructions are viewed as independent non-composable rules



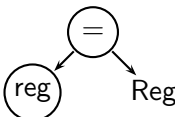

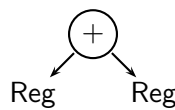
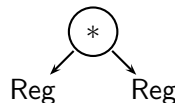
Full Tree Matching

Instructions are viewed as independent non-composable rules



Tree Tiling

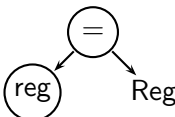
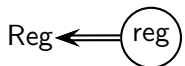
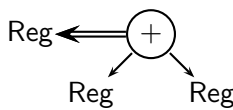
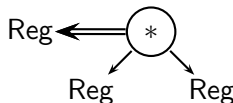
Instructions are viewed as composable rules

Instructions	Subject Tree
	
	
	
	



Tree Tiling

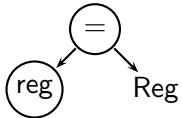
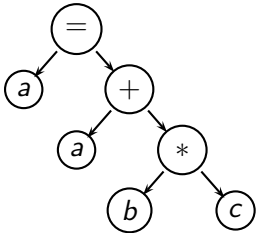
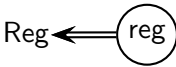
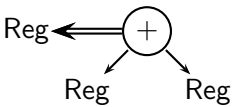
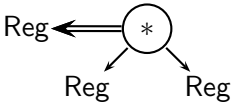
Instructions are viewed as composable rules

Instructions	Subject Tree
	
	
	
	



Tree Tiling

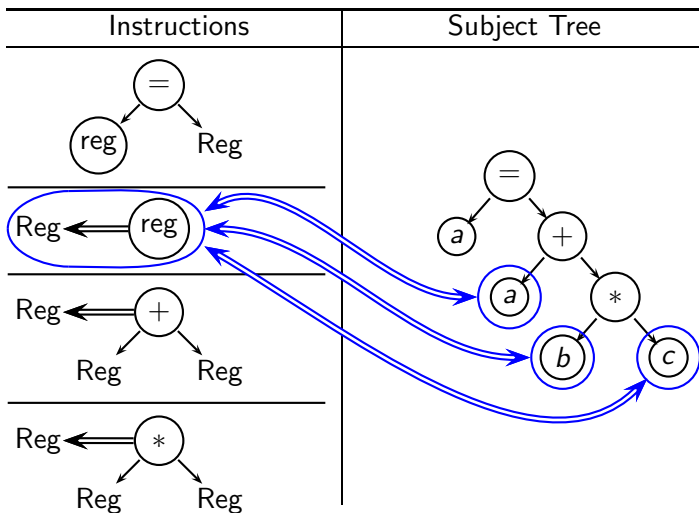
Instructions are viewed as composable rules

Instructions	Subject Tree
	
	
	
	



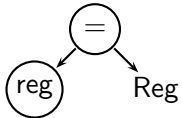
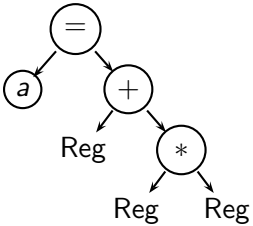
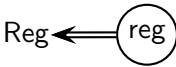
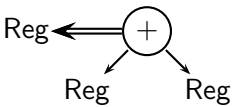
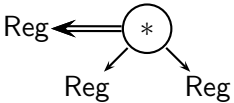
Tree Tiling

Instructions are viewed as composable rules



Tree Tiling

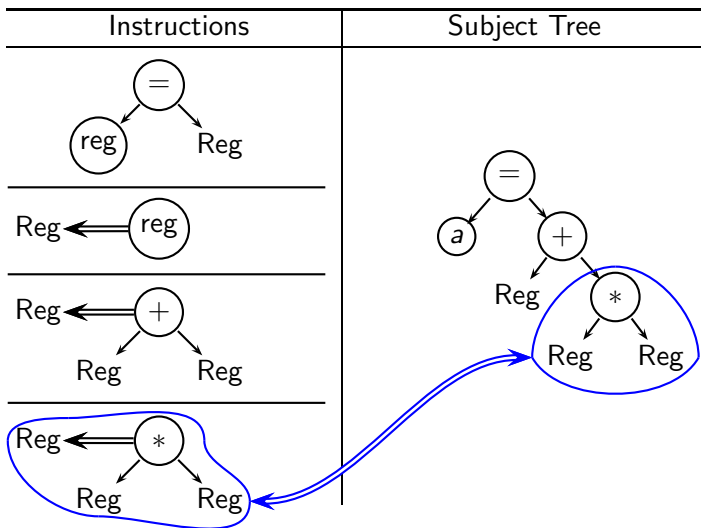
Instructions are viewed as composable rules

Instructions	Subject Tree
	
	
	
	



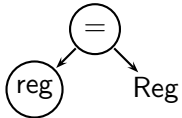
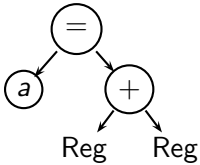
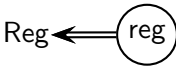
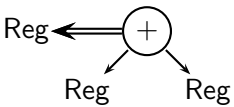
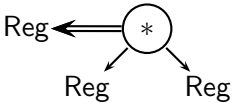
Tree Tiling

Instructions are viewed as composable rules



Tree Tiling

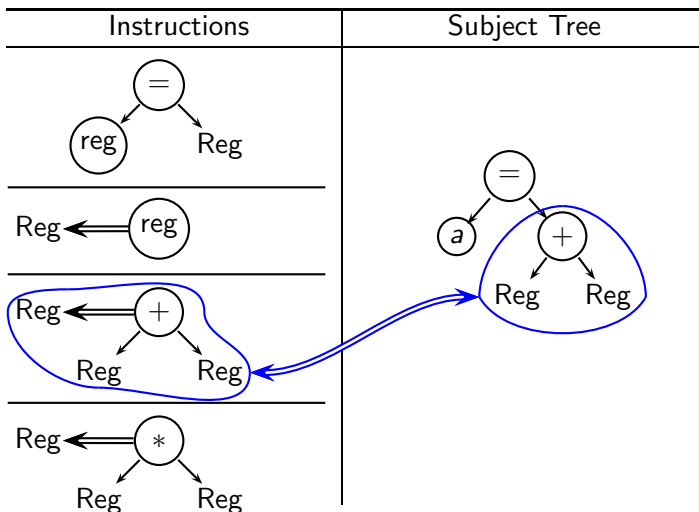
Instructions are viewed as composable rules

Instructions	Subject Tree
	
	
	
	



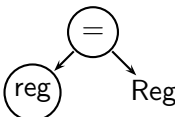
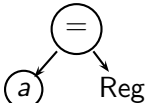
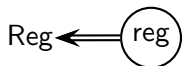
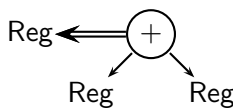
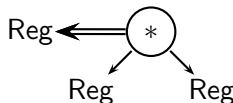
Tree Tiling

Instructions are viewed as composable rules



Tree Tiling

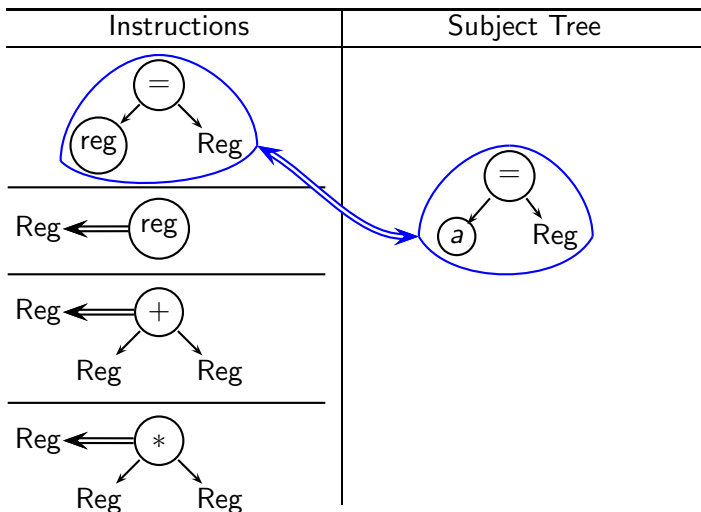
Instructions are viewed as composable rules

Instructions	Subject Tree
	
	
	
	



Tree Tiling

Instructions are viewed as composable rules



Improving Machine Descriptions and Instruction Selection

Current Status:

- Preliminary investigations seem very promising
 - ▶ Fewer rules
 - ▶ Simple rules



Improving Machine Descriptions and Instruction Selection

Current Status:

- Preliminary investigations seem very promising
 - ▶ Fewer rules
 - ▶ Simple rules
- Prototype of new code generator generator (cgg) is being tested in a toy compiler set up

