

# GCC RTL & MACHINE DESCRIPTION

By Priyatham Bollimpalli



Little review of basics

What are the stages of a compiler?

# Compilation Stage-1:Preprocessing



- ❑ Performed by a program called the preprocessor
- ❑ Modifies the source code (in RAM) according to preprocessor directives (preprocessor commands) embedded in the source code
- ❑ Strips comments and white space from the code
- ❑ The source code as stored on disk is not modified

# Compilation Stage-2:Compilation



- ❑ Performed by a program called the compiler
- ❑ Translates the preprocessor-modified source code into object code
- ❑ Checks for syntax errors and warnings
- ❑ Saves the object code to a disk file, if instructed to do
- ❑ If any compiler errors are received, no object code file will be generated
- ❑ An object code file will be generated if only warnings, not errors, are received

# Compilation Stage-3:Linking



- Combines the program object code with other object code to produce the executable file
- The other object code can come from the Run-Time Library, other libraries, or object files that you have created
- Saves the executable code to a disk file. On the Linux system, that file is called a.out
- If any linker errors are received, no executable file will be generated

# Compiler Phases

Phase	Output
<i>Programmer (source code producer)</i>	Source string
<i>Scanner (performs lexical analysis)</i>	Token string
<i>Parser (performs syntax analysis based on the grammar of the programming language)</i>	Parse tree or abstract syntax tree
<i>Semantic analyzer (type checking etc)</i>	Annotated parse tree or abstract syntax tree
<i>Intermediate code generator</i>	Three-address code, quads, or RTL
<i>Optimizer</i>	Three-address code, quads, or RTL
<i>Code generator</i>	Assembly code

- You know about front end and some back end phases

# Why Front End?



- Machine Independent
- Can be written in a high level language
- Re-use Oriented Programming
- Lessens Time Required to Generate New Compilers
- Makes developing new programming languages simpler



# After Front End?



- All language compilers
  - Read source code
  - Output assembly code
- Language compilers have different front ends
  - Each front end parses input and produces an abstract syntax tree
- AST is converted to a common middle-end format
  - GENERIC or GIMPLE
  - Next these are converted to RTL (Register Transfer Language)



# “Middle-End”



- *GENERIC* is an intermediate representation language used as a "middle-end" while compiling source code into executable binaries.
- A subset, called *GIMPLE*, is targeted by all the front-ends of *GCC*
- The middle stage of *GCC* does all the code analysis and optimization, working independently of both the compiled language and the target architecture, starting from the *GENERIC* representation and expanding it to Register Transfer Language.

# Why “Middle-End”?



- In transforming the source code to GIMPLE, complex expressions are split into a three address code using temporary variables
- This is for simplifying the analysis and optimization of imperative programs
- In GCC, RTL is generated from the GIMPLE representation
- Important to know about RTL to get an idea of how assembly code is generated from RTL with the help of templates

# RTL



- Register transfer language (RTL) : A Scheme-like language based on virtual registers
- Sometimes, initial RTL is generated with hints about the target machine
- RTL is refined through many (58) passes. Details at <http://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html>
- Final passes use target machine registers and instructions
- From there, conversion to machine-specific assembly language is very easy

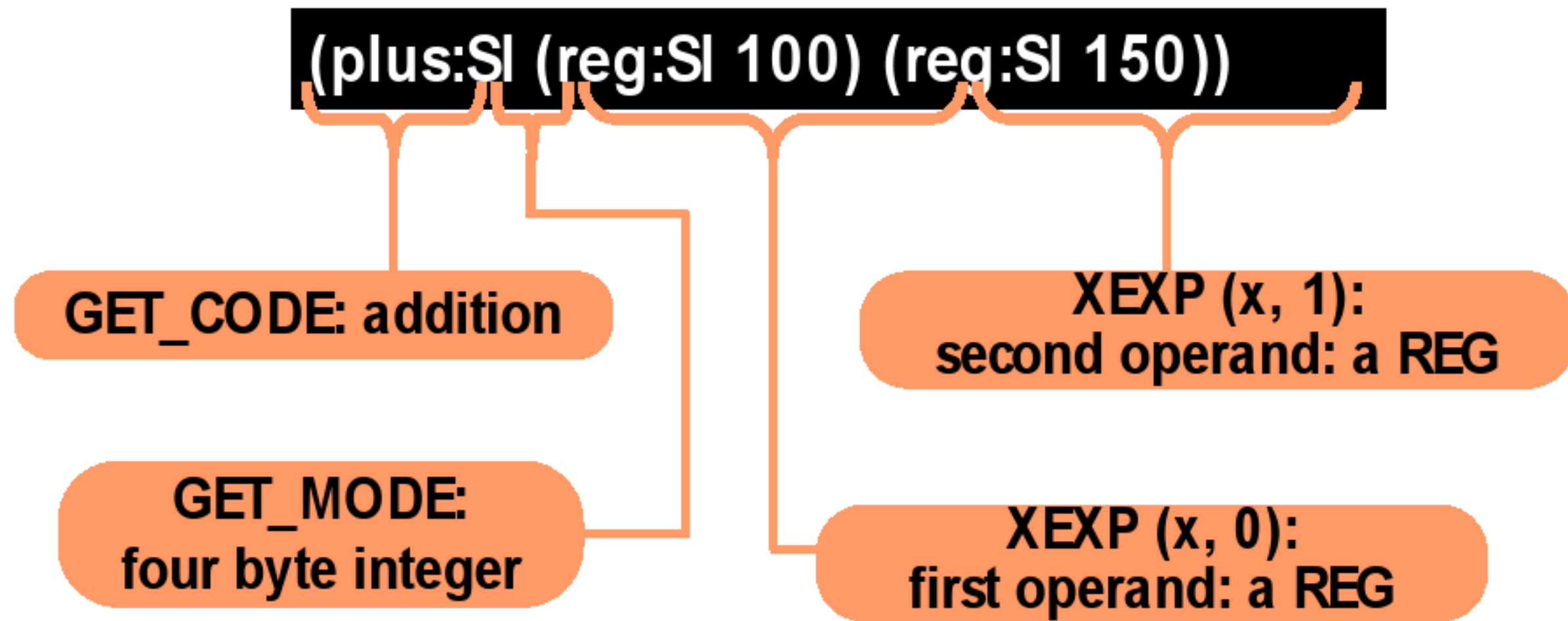
# An Example

- GCC's RTL is usually written in a form which looks like a Lisp S-expression. For example:

```
(set (reg:SI 140)
      (plus:SI (reg:SI 138)
                (reg:SI 139)))
```

- It says "add the contents of register 138 to the contents of register 139 and store the result in register 140".
- The SI specifies the access mode for each registers. Here it is "SI mode", i.e. "access the register as 32-bit integer"

# Example of an RTL Node



# Generating Code for If-Then-Else

```
if (cond1)
...
elseif (cond2)
...
else
...

```

Call **expand\_start\_tree** with tree for cond1.

Generate code.

Call **expand\_start\_elseif** with tree for cond2.

Generate code.

Call **expand\_start\_else**.

Generate code.

Call **expand\_end\_cond**.

# Generating Code for Loops

- Uses a struct nesting; need not know format, take structure given and pass back.
- `expand_start_loop`
  - ▣ Make a new loop.
- `expand_start_loop_continue_elsewhere`
  - ▣ Similar but provide continuation point
- `expand_end_loop`
  - ▣ Mark the end of the loop
- `expand_{continue,exit}_loop`
  - ▣ Exit or continue specified loop
- Flags are used in an RTL Expressions



# Examples of Optimizations

- Many types of optimizations during RTL generation
- Merging of comparisons  
`a >= 10 && a <= 100` becomes  
`(unsigned) a - 10 <= 90`
- `if (cond) x = A; else x = B;` into  
`x = B; if (cond) x = A;`
- `xor (not X) (not Y)` to  
`(xor X Y)`
- Extensive theory and research

# Machine Descriptions



- Generation of ASM from RTL
- Depends only on target machine
- A machine description has two parts
  - ▣ a file of instruction patterns (.md file)
  - ▣ a C header file of macro definitions (around 500)
- Linked to file md in build directory.
- Written in RTL, represented in LISP-like format.
- Read by various programs to generate .h and .c files that become part of GCC.

# Templates

- Bulk of MD file
- Specified with `define_insn`
  - ▣ For simple and direct instructions
  - ▣ template given is inserted into the insn list
- Some specified with `define_expand`. One of the following may happen based on the condition logic
  - ▣ The condition logic may manually create new insns for the insn list and invoke `DONE`.
  - ▣ For certain named patterns, it may invoke `FAIL` to tell the compiler to use an alternate way of performing that task.
  - ▣ If it invokes neither `DONE` nor `FAIL`, the template given in the pattern is inserted, as if it were a `define_insn`

# define\_insn

```
(define_insn "addsi3_internal"

  [(set (match_operand:SI 0 "register_operand"
    "=d,d") (plus:SI (match_operand:SI 1 "reg_or_0_operand"
    "dJ,dJ") (match_operand:SI 2 "arith_operand" "d,Q"))))]

  "!TARGET_MIPS16"
  "@
    addu\t%0,%z1,%2
    addiu\t%0,%z1,%2"

  [(set_attr "type" "arith")
   (set_attr "mode" "SI")])
```

# Syntax

The basic structure of a `define_insn` in MD is:

```
(define_insn
KEY (also called NAME - Optional)
RTL TEMPLATE
C CONDITION
ASM
OPTIONAL ATTRIBUTES SPECIFICATION )
```

- `condition-string` (C expression) that is the final test to decide whether an `insn` body matches this pattern

Correspondence between the generic `define_insn` and the concrete MIPS example.

KEY	"addsi3_internal"
RTL TEMPLATE	<pre>[(set (match_operand:SI 0 "register_operand" "=d,d")       (plus:SI (match_operand:SI 1 "reg_or_0_operand" "dJ,dJ")                 (match_operand:SI 2 "arith_operand" "d,Q")))]</pre>
C CONDITION	"!TARGET_MIPS16"
ASM	<pre>"@ addu\t%0,%z1,%2 addiu\t%0,%z1,%2"</pre>
ATTRIBUTES	<pre>[(set_attr "type"      "arith")  (set_attr "mode"      "SI")]</pre>



# Example

Consider a `define_insn` expression from `mips.md` file

```
(define_insn "add<mode>3"  
  [(set (match_operand:ANYF 0 "register_operand" "=f")  
    (plus:ANYF (match_operand:ANYF 1 "register_operand" "f")  
      (match_operand:ANYF 2 "register_operand" "f")))]  
  ""  
  "add.<fmt>\t%0,%1,%2"  
  [(set_attr "type" "fadd")  
    (set_attr "mode" "<UNITMODE>")])
```

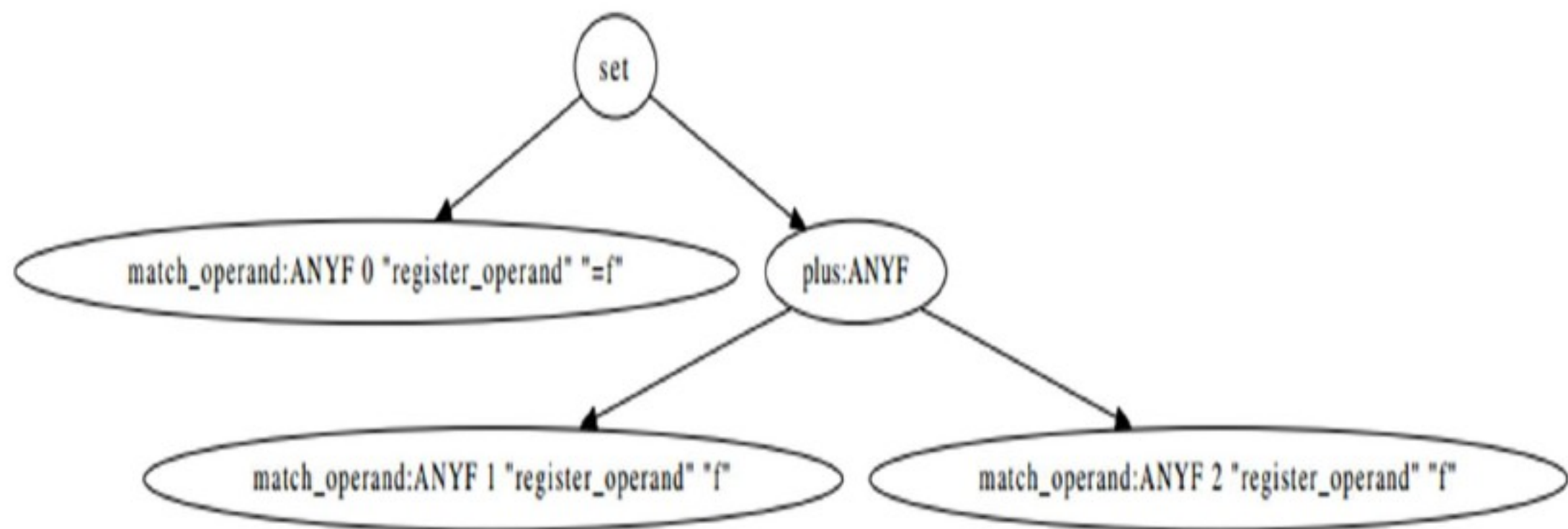




Here the RTL template is

```
[(set (match_operand:ANYF 0 "register_operand" "=f")  
(plus:ANYF (match_operand:ANYF 1 "register_operand" "f")  
  (match_operand:ANYF 2 "register_operand" "f")))]
```


This can be represented in a tree form as



- (match\_operand: *m n predicate constraint*)
- operand number *n* of the insn
- *predicate* is a string that is the name of a function that accepts two arguments, an expression and a machine mode.
- *constraint* controls reloading and the choice of the best register class to use for a value
  - m memory operand is allowed
  - r register operand is allowed provided that it is in a general register
  - = write only
  - f floating operand

# Another Example

- ```
(define_insn "tstsi"
  [(set (cc0)
        (match_operand:SI 0 "general_operand" "rm"))]
  ""
  {
    if (TARGET_68020 || !ADDRESS_REG_P (operands[0]))
      return "tstl %0";
    return "cmpl #0,%0";
  })
```
- Sets the condition codes based on the value of a general operand.
- The name `tstsi' means "test a SImode value" - when it is necessary to test such a value, an insn to do so can be constructed using this pattern.
- Based on the kind of operand and the specific type of CPU for which code is being generated, Output is generated.



```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "general_operand" "=r,m")
    (plus:SI (match_operand:SI 1 "general_operand" "0,0")
      (match_operand:SI 2 "general_operand" "g,r")))]
  ""
  "@
  addr %2,%0
  addm %2,%0")
```

▣ What does it do?

# define\_expand

- A `define_expand` can produce more than one RTL instruction
- Syntax
  - ▣ Optional name
  - ▣ RTL Template
  - ▣ A condition
  - ▣ Preparatory statements
- The condition logic may manually create new insns for the insn list, say via `emit_insn()`, and invoke DONE



```

        (define_expand "mul<mode>3"
          [(set (match_operand:GPR 0 "register_operand")
                (mult:GPR (match_operand:GPR 1 "register_operand")
                          (match_operand:GPR 2 "register_operand")))]
          "")
    {
      if (TARGET_LOONGSON_2EF || TARGET_LOONGSON_3A)
        emit_insn (gen_mul<mode>3_mul3_loongson (operands[0],
  operands[1], operands[2]));
      else if (ISA_HAS_<D>MUL3)
        emit_insn (gen_mul<mode>3_mul3 (operands[0],
   operands[1], operands[2]));
      else if (TARGET_FIX_R4000)
        emit_insn (gen_mul<mode>3_r4000 (operands[0],
   operands[1], operands[2]));
      else
        emit_insn
          (gen_mul<mode>3_internal (operands[0],
                                   operands[1], operands[2]));
    }
    DONE;
  })

```



# Jump Instructions



- The machine description should define a single pattern, usually a `define_expand`, which expands to all the required insns.
- Usually, this would be a comparison insn to set the condition code and a separate branch insn testing the condition code and branching or not according to its value. For many machines, however, separating compares and branches is limiting
- So `define_expand` is used in GCC
- Widely used in conditional and looping constructs

# Loop Instructions

- Some machines have special jump instructions that can be utilized to make loops more efficient
- GCC has three special named patterns to support low overhead looping
  - ▣ decrement\_and\_branch\_until\_zero
  - ▣ doloop\_begin
  - ▣ doloop\_end

```
(define_insn "decrement_and_branch_until_zero"
  [(set (pc)
    (if_then_else
      (ge (plus:SI (match_operand:SI 0 "general_operand" "+d*am")
        (const_int -1))
        (const_int 0))
      (label_ref (match_operand 1 "" ""))
      (pc)))
    (set (match_dup 0)
      (plus:SI (match_dup 0)
        (const_int -1)))]
  "find_reg_note (insn, REG_NONNEG, 0)"
  "...")
```

# Last step

- Once the insn list is generated, various optimization passes convert, replace, and rearrange the insns in the insn list.
- For this, the `define_split` and `define_peephole` patterns get used.

**Branch on Zero**

```
(set (cc0) (reg:SI 100))  
(set (pc) (if_then_else (eq (cc0) (const_int 0)) (label_ref 18) (pc)))
```

to

```
(set (pc) (if_then_else (eq (reg:SI 100) (const_int 0))  
                        (label_ref 18) (pc)))
```

- Finally we obtain the assembly code!!

# Facts



- This is just a tiny whisk of hair on tip of the iceberg. The GCC documentation is huge!!

## **Size of GCC**

- Machine Descriptions : 421,741 lines
  - \*.md: 155,244
  - \*.h: 135,778
  - \*.c: 126,872
  - Makefile insertions: 3,847
- Distributed front ends: 357,744 lines
- Base compiler: 328,297 lines

# Facts

- Distributed front ends: 357,744 lines
  - C: 23,315
  - C++ : 97,642
  - Chill: 42,225
  - F77: 133,121
  - Java: 47,284
  - Objective-C: 7,973 + 6,880 (library)
- Base compiler: 328,297
  - Optimizer: 70,853
  - Generation of compiler code: 14,216
- **Total size: 1,107,782 lines**

# References



- <http://www.wikipedia.org/>
- <http://gcc.gnu.org/onlinedocs/gccint/>
- <http://www.gnat.com/~kenner/gccut.ppt>
- <http://www.cse.iitb.ac.in/~uday/>



*Thank you!*