First Level Gray Box Probing

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

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

1 July 2011

Outline

- Introduction to Graybox Probing of GCC
- Examining GIMPLE Dumps
  - Translation of data accesses
  - Translation of intraprocedural control flow
  - Translation of interprocedural control flow
- Examining RTL Dumps
- Examining Assembly Dumps
- Conclusions

What is Gray Box Probing of GCC?

- Black Box probing:
  Examining only the input and output relationship of a system
- White Box probing:
  Examining internals of a system for a given set of inputs
- Gray Box probing:
  Examining input and output of various components/modules
  - Overview of translation sequence in GCC
  - Overview of intermediate representations
  - Intermediate representations of programs across important phases
First Level Gray Box Probing of GCC

- Restricted to the most important translations in GCC

Basic Transformations in GCC

Transformation from a language to a different language

Basic Transformations in GCC

- Target Independent
- Target Dependent

- Parse → Simplify → Tree SSA
- Optimize → Generate RTL
- Optimize RTL → Generate ASM

Basic Transformations in GCC

- GIMPLE → RTL
- RTL → ASM

Transformation Passes in GCC 4.6.0

- A total of 207 unique pass names initialized in
  \$\{SOURCE\}/gcc/passes.c

  Total number of passes is 241.

    • Some passes are called multiple times in different contexts
      Conditional constant propagation and dead code elimination are called thrice
    • Some passes are enabled for specific architectures
    • Some passes have many variations (e.g., special cases for loops)
      Common subexpression elimination, dead code elimination

  The pass sequence can be divided broadly in two parts

    • Passes on GIMPLE
    • Passes on RTL

  Some passes are organizational passes to group related passes

Passes On GIMPLE in GCC 4.6.0

<table>
<thead>
<tr>
<th>Pass Group</th>
<th>Examples</th>
<th>Number of passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lowering</td>
<td>GIMPLE IR, CFG Construction</td>
<td>10</td>
</tr>
<tr>
<td>Simple Interprocedural Passes</td>
<td>Conditional Constant Propagation, Inlining, SSA Construction</td>
<td></td>
</tr>
<tr>
<td>(Non-LTO)</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>Regular Interprocedural Passes</td>
<td>Constant Propagation, Inlining, Pointer Analysis</td>
<td></td>
</tr>
<tr>
<td>(LTO)</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>LTO generation passes</td>
<td></td>
<td>02</td>
</tr>
<tr>
<td>Other Intraprocedural optimizations</td>
<td>Constant Propagation, Dead Code Elimination, PRE Value Range Propagation, Rename SSA</td>
<td></td>
</tr>
<tr>
<td>Loop Optimizations</td>
<td>Vectorization, Parallelization, Copy Propagation, Dead Code Elimination</td>
<td></td>
</tr>
<tr>
<td>Generating RTL</td>
<td></td>
<td>01</td>
</tr>
<tr>
<td>Total number of passes on GIMPLE</td>
<td></td>
<td>154</td>
</tr>
</tbody>
</table>
### Passes On RTL in GCC 4.6.0

<table>
<thead>
<tr>
<th>Pass Group</th>
<th>Examples</th>
<th>Number of passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intraprocedural Optimizations</td>
<td>CSE, Jump Optimization, Dead Code Elimination, Jump Optimization</td>
<td>27</td>
</tr>
<tr>
<td>Loop Optimizations</td>
<td>Loop Invariant Movement, Peeling, Unswitching</td>
<td>07</td>
</tr>
<tr>
<td>Machine Dependent Optimizations</td>
<td>Register Allocation, Instruction Scheduling, Peephole Optimizations</td>
<td>50</td>
</tr>
<tr>
<td>Assembly Emission and Finishing</td>
<td></td>
<td>03</td>
</tr>
<tr>
<td><strong>Total number of passes on RTL</strong></td>
<td></td>
<td><strong>87</strong></td>
</tr>
</tbody>
</table>

### Finding Out List of Optimizations

Along with the associated flags:

- A complete list of optimizations with a brief description
  
  ```
gcc -c --help=optimizers
  ```

- Optimizations enabled at level 2 (other levels are 0, 1, 3, and s)
  
  ```
gcc -c -O2 --help=optimizers -Q
  ```

### Producing the Output of GCC Passes

- Use the option `-fdump-<ir>-<passname>`
  
  - `<ir>` could be:
    - `tree`: Intraprocedural passes on GIMPLE
    - `ipa`: Interprocedural passes on GIMPLE
    - `rtl`: Intraprocedural passes on RTL
  
- Use `all` in place of `<pass>` to see all dumps
  
  Example: `gcc -fdump-tree-all -fdump-rtl-all test.c`

- Dumping more details:
  
  - Suffix `raw` for tree passes and `details` or `slim` for RTL passes
  
  Individual passes may have more verbosity options (e.g. `-fsched-verbose=5`)

- Use `-S` to stop the compilation with assembly generation

- Use `--verbose-asm` to see more detailed assembly dump

### Total Number of Dumps

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Number of Dumps</th>
<th>Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>47</td>
<td>Fast compilation</td>
</tr>
<tr>
<td>O1</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>O2</td>
<td>158</td>
<td></td>
</tr>
<tr>
<td>O3</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>Os</td>
<td>156</td>
<td>Optimize for space</td>
</tr>
</tbody>
</table>
### Selected Dumps for Our Example Program

<table>
<thead>
<tr>
<th>GIMPLE dumps (t)</th>
<th>C Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>001t.tu</td>
<td>Parser</td>
</tr>
<tr>
<td>003t.original</td>
<td>AST</td>
</tr>
<tr>
<td>004t.gimple</td>
<td>Gimplifier</td>
</tr>
<tr>
<td>006t.vcg</td>
<td>GIMPLE</td>
</tr>
<tr>
<td>009t.omplower</td>
<td>CFG Generator</td>
</tr>
<tr>
<td>010t.lower</td>
<td>RTL Generator</td>
</tr>
<tr>
<td>012t.eh</td>
<td></td>
</tr>
<tr>
<td>013t.cfg</td>
<td></td>
</tr>
<tr>
<td>017t.ssa</td>
<td></td>
</tr>
<tr>
<td>018t.veclower</td>
<td></td>
</tr>
<tr>
<td>019t.inline_param1</td>
<td></td>
</tr>
<tr>
<td>020t.einline</td>
<td></td>
</tr>
<tr>
<td>037t.release_ssa</td>
<td></td>
</tr>
<tr>
<td>038t.inline_param2</td>
<td></td>
</tr>
<tr>
<td>044i.whole-program</td>
<td></td>
</tr>
<tr>
<td>048i.inline</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ipa dumps (i)</th>
<th>pro_epilogue generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000i.cgraph</td>
<td></td>
</tr>
<tr>
<td>014i.visibility</td>
<td></td>
</tr>
<tr>
<td>015i.early_local_cleanups</td>
<td></td>
</tr>
<tr>
<td>044i.whole-program</td>
<td></td>
</tr>
<tr>
<td>048i.inline</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rtl dumps (r)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>138t.cplxlower0</td>
<td></td>
</tr>
<tr>
<td>143t.optimized</td>
<td></td>
</tr>
<tr>
<td>224t.statistics</td>
<td></td>
</tr>
<tr>
<td>ipa dumps (i)</td>
<td></td>
</tr>
<tr>
<td>000i.cgraph</td>
<td></td>
</tr>
<tr>
<td>014i.visibility</td>
<td></td>
</tr>
<tr>
<td>015i.early_local_cleanups</td>
<td></td>
</tr>
<tr>
<td>044i.whole-program</td>
<td></td>
</tr>
<tr>
<td>048i.inline</td>
<td></td>
</tr>
<tr>
<td>rtl dumps (r)</td>
<td></td>
</tr>
<tr>
<td>144r.expand</td>
<td></td>
</tr>
<tr>
<td>145r.sibling</td>
<td></td>
</tr>
<tr>
<td>147r.initvals</td>
<td></td>
</tr>
<tr>
<td>148r.unshare</td>
<td></td>
</tr>
<tr>
<td>149r.vregs</td>
<td></td>
</tr>
<tr>
<td>150r.into_cfglayout</td>
<td></td>
</tr>
<tr>
<td>151r.jump</td>
<td></td>
</tr>
<tr>
<td>163r.reginfo</td>
<td></td>
</tr>
<tr>
<td>183r.outof_cfglayout</td>
<td></td>
</tr>
<tr>
<td>184r.split1</td>
<td></td>
</tr>
<tr>
<td>186r.dfiniut</td>
<td></td>
</tr>
<tr>
<td>187r.mode_sw</td>
<td></td>
</tr>
<tr>
<td>188r.asmcons</td>
<td></td>
</tr>
<tr>
<td>191r.ira</td>
<td></td>
</tr>
<tr>
<td>194r.split2</td>
<td></td>
</tr>
<tr>
<td>198r.pro_and_epiologue</td>
<td></td>
</tr>
<tr>
<td>211r.stack</td>
<td></td>
</tr>
<tr>
<td>212r.alignments</td>
<td></td>
</tr>
<tr>
<td>215r.mach</td>
<td></td>
</tr>
<tr>
<td>216r.barriers</td>
<td></td>
</tr>
<tr>
<td>220r.shorten</td>
<td></td>
</tr>
<tr>
<td>221r.nothrow</td>
<td></td>
</tr>
<tr>
<td>222r.final</td>
<td></td>
</tr>
<tr>
<td>223r.dfinish</td>
<td></td>
</tr>
<tr>
<td>assembly</td>
<td></td>
</tr>
</tbody>
</table>

### Part 2

**Examining GIMPLE Dumps**

- **About GIMPLE**
  - Three-address representation derived from GENERIC
  - Computation represented as a sequence of basic operations
  - Temporaries introduced to hold intermediate values
  - Control construct are explicated into conditional jumps

- **Examining GIMPLE Dumps**
  - Examining translation of data accesses
  - Examining translation of control flow
  - Examining translation of function calls
GIMPLE: Composite Expressions Involving Local and Global Variables

```c
int main()
{
    int x = 10;
    int y = 5;
    x = a + x * y;
    y = y - a * x;
}
```

Global variables are treated as “memory locations” and local variables are treated as “registers”.

GIMPLE: 1-D Array Accesses

```c
int main()
{
    int a[3], x;
    x = a[1] + a[2];
    a[0] = a[1] + a[1]*x;
}
```

GIMPLE: 2-D Array Accesses

```c
int main()
{
    int a[3][3], x, y;
    a[0][0] = 7;
    a[1][1] = 8;
    a[2][2] = 9;
    x = a[0][0] / a[1][1];
    y = a[1][1] % a[2][2];
}
```

☆ No notion of “addressable memory” in GIMPLE.
☆ Array reference is a single operation in GIMPLE and is linearized in RTL during expansion.

GIMPLE: Use of Pointers

```c
int main()
{
    int * D.1953; int * * a; int b; int c;
    b = &c;
    a = &b;
    **a = 10; /* c = 10 */
}
```
**GIMPLE: Use of Structures**

```c
typedef struct address {
    char *name;
} ad;

typedef struct student {
    int roll;
    ad *ct;
} st;

int main()
{
    st *s;
    s = malloc(sizeof(st));
    s->roll = 1;
    s->ct = malloc(sizeof(ad));
    s->ct->name = "Mumbai";
}
```

**GIMPLE: Pointer to Array**

```c
int main()
{
    int *p_a, a[3];
    p_a = &a[0];
    *p_a = 10;
    *(p_a+1) = 20;
    *(p_a+2) = 30;
}
```

**GIMPLE: Translation of Conditional Statements**

```c
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
        if (a<=12)
            a = a+b+c;
    }
}
```

**GIMPLE: Translation of Loops**

```c
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a + 1;
        if (a<=12)
            a = a+b+c;
    }
```
Control Flow Graph: Textual View

test.c.004t.gimple

if (a <= 12) goto <D.1200>; else goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:

Control Flow Graph: Pictorial View

test.c.013t.cfg

if (a <= 7)
a = a + 1;
if (a <= 12)
a = a + b + c;
while(a <= 7)
a = a + 1;
1 July 2011
Graybox Probing-I: Examining GIMPLE Dumps
24/43

GIMPLE: Function Calls and Call Graph

`test.c`

```c
extern int divide(int, int);
int multiply(int a, int b) {
    return a*b;
}

int main()
{ int x,y;
  x = divide(20,5);
  y = multiply(x,2);
  printf("%d\n", y);
}
```

```c
GIMPLE: Call Graphs for Recursive Functions

`test.c`

```c
int even(int n)
{ if (n == 0) return 1;
  else return (!odd(n-1));
}

int odd(int n)
{ if (n == 1) return 1;
  else return (!even(n-1));
}

main()
{ int n;
  n = abs(readNumber());
  if (even(n))
    printf("n is even\n");
  else printf("n is odd\n");
}
```
int x=2, y=3;
```
x = y++ + ++x + ++y;
```

What are the values of x and y?

\[ x = 10, y = 5 \]

\[
\begin{array}{c|c}
  x & 3 \\
  y & 4 \\
  (y + x) & 6 \\
  (y + x) + y & 11 \\
\end{array}
\]
Inspect GIMPLE When in Doubt (2)

- How is \( a[i] = i++ \) handled?
  This is an undefined behaviour as per C standards.

- What is the order of parameter evaluation?
  For a call \( f(getX(), getY()) \), is the order left to right? arbitrary?
  Is the evaluation order in GCC consistent?

- Understanding complicated declarations in C can be difficult
  What does the following declaration mean:
  ```c
  int (* (*MYVAR)(int))(10);
  ```
  Hint: Use `‐fdump-tree-original-raw-verbose` option. The dump to see is `003t.original`

---

RTL for i386: Arithmetic Operations (1)

Translation of \( a = a + 1 \)

**Dump file**: `test.c.144r.expand`

```c
(insn 12 11 13 4 (parallel [
  (set (mem/c/i:SI
       (plus:SI
         (reg/f:SI 54 virtual-stack-vars)
         (const_int -4 [0xffffffff]) [0 a+0 S4 A32])
       (plus:SI
         (mem/c/i:SI
          (plus:SI
           (reg/f:SI 54 virtual-stack-vars)
           (const_int -4 [0xffffffff]) [0 a+0 S4 A32])
          (const_int 1 [0x1])))
    (clobber (reg:CC 17 flags))
  ]) t.c:24 -1 (nil))
```
RTL for i386: Arithmetic Operations (1)

Translation of $a = a + 1$

Dump file: test.c.144r.expand

(a is a local variable allocated on stack)

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay

1 July 2011

Graybox Probing-I: Examining RTL Dumps

29/43
Translation of $a = a + 1$

**Dump file:** test.c.144r.

Current Instruction

```
{[r54:SI-0x4]=[r54:SI-0x4]+0x1;
clobber flags:CC;
}
```

Previous Instruction

```
(insn 12 11 13 4 (parallel [
  (set (mem/c/i:SI
    (plus:SI
      (reg/f:SI 54 virtual-stack-vars)
      (const_int -4 [0xfffffffc])) [0 a+0 S4 A32])
    (plus:SI
      (mem/c/i:SI
        (plus:SI
          (reg/f:SI 54 virtual-stack-vars)
          (const_int -4 [0xfffffffc])) [0 a0 S4 A32])
        (plus:SI
          (mem/c/i:SI
            (plus:SI
              (reg/f:SI 54 virtual-stack-vars)
              (const_int -4 [0xfffffffc])) [0 a0 S4 A32])
            (clobber (reg:CC 17 flags))
          ])) t.c:24 -1 (nil))
```

Next Instruction

```
(insn 12 11 13 4 (parallel [
  (set (mem/c/i:SI
    (reg/f:SI 54 virtual-stack-vars)
    (const_int -4 [0xfffffffc])) [0 a0 S4 A32])
    (plus:SI
      (mem/c/i:SI
        (plus:SI
          (reg/f:SI 54 virtual-stack-vars)
          (const_int -4 [0xfffffffc])) [0 a0 S4 A32])
        (plus:SI
          (mem/c/i:SI
            (plus:SI
              (reg/f:SI 54 virtual-stack-vars)
              (const_int -4 [0xfffffffc])) [0 a0 S4 A32])
            (clobber (reg:CC 17 flags))
          ])) t.c:24 -1 (nil))
```
Additional Information in RTL

(insn 12 11 13 4 (parallel [
  (set (mem/c/i:SI
   (plus:SI
    (reg/f:SI 54 virtual-stack-vars)
    (const_int -4 [0xffffffffc])) [0 a+0 S4 A32])
   (plus:SI
    (mem/c/i:SI
     (plus:SI
      (reg/f:SI 54 virtual-stack-vars)
      (const_int -4 [0xffffffffc])) [0 a+0 S4 A32])
     (const_int 1 [0x1])))
  (clobber (reg:CC 17 flags))
]) t.c:24 -1 (nil))

File name: Line number

memory reference that does not trap

scalar that is not apart of an aggregate
Additional Information in RTL

(insn 12 11 13 4 (parallel [
  (set (mem/c/i:SI
       (plus:SI
         (reg/f:SI 54 virtual-stack-vars)
         (const_int -4 [0xffffffff])) [0 a+0 S4 A32])
       (const_int 1 [0x1]))
       (clobber (reg:CC 17 flags))])) t.c:24 -1 (nil))

Load a into reg64

Dump file: test.c.144r.expand

RTL for i386: Arithmetic Operations (2)

Translation of \( a = a + 1 \) when \( a \) is a global variable

Dump file: test.c.144r.expand

(insn 11 10 12 4 (set
   (reg:SI 64 [ a.0 ])
   (mem/c/i:SI (symbol_ref:SI ("a")
     <var_decl 0xb7d8d000 a>) [0 a+0 S4 A32])) t.c:26 -1 (nil))
RTL for i386: Arithmetic Operations (2)

Translation of $a = a + 1$ when $a$ is a global variable

**Dump file:** test.c.144r.expand

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insn 11 10 12 4</td>
<td>(set (reg:SI 64 [a.0]) (mem/c:i:SI (symbol_ref:SI (&quot;a&quot;) [var_decl 0xb7d8d000 a]) [0 a+0 S4 A32]))</td>
</tr>
<tr>
<td>insn 12 11 13 4</td>
<td>(parallel [ (set (reg:SI 63 [a.1]) (plus:SI (reg:SI 64 [a.0]) (const_int 1 [0x1]))) (clobber (reg:CC 17 flags)) ]) t.c:26 -1 (nil))</td>
</tr>
</tbody>
</table>

---

RTL for i386: Arithmetic Operations (2)

Translation of $a = a + 1$ when $a$ is a global variable

**Dump file:** test.c.144r.expand

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insn 11 10 12 4</td>
<td>(set (reg:SI 64 [a.0]) (mem/c:i:SI (symbol_ref:SI (&quot;a&quot;) [var_decl 0xb7d8d000 a]) [0 a+0 S4 A32]))</td>
</tr>
<tr>
<td>insn 12 11 13 4</td>
<td>(parallel [ (set (reg:SI 63 [a.1]) (plus:SI (reg:SI 64 [a.0]) (const_int 1 [0x1]))) (clobber (reg:CC 17 flags)) ]) t.c:26 -1 (nil))</td>
</tr>
</tbody>
</table>

---

RTL for i386: Arithmetic Operations (3)

Translation of $a = a + 1$ when $a$ is a formal parameter

**Dump file:** test.c.144r.expand

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insn 10 9 11 4</td>
<td>(parallel [ (set (mem/c:i:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32])) (plus:SI (mem/c:i:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]) (const_int 1 [0x1]))) (clobber (reg:CC 17 flags)) ]) t1.c:25 -1 (nil))</td>
</tr>
</tbody>
</table>
RTL for i386: Arithmetic Operations (3)

Translation of $a = a + 1$ when $a$ is a formal parameter

Dump file: test.c.144r.expand

(\texttt{insn 10 9 11 4 (parallel [}}
(\texttt{set [}}
(\texttt{mem/c/i:SI [}}
(\texttt{(reg/f:SI 53 virtual-incoming-args)}}
(\texttt{(const_int 4 [0x4])) [0 a+0 S4 A32])
(\texttt{PLUS:SI [}}
(\texttt{(mem/c/i:SI)}}
(\texttt{(reg/f:SI 53 virtual-incomming-args)}}
(\texttt{(const_int 1 [0x1])))
(\texttt{clobber (reg:CC 17 flags)) [}}
(\texttt{t1.c:25 -1 (nil))})

Access through argument pointer register instead of frame pointer register
No offset required?

RTL for i386: Arithmetic Operations (4)

Translation of $a = a + 1$ when $a$ is the second formal parameter

Dump file: test.c.144r.expand

(\texttt{insn 10 9 11 4 (parallel [}}
(\texttt{set [}}
(\texttt{mem/c/i:SI [}}
(\texttt{(reg/f:SI 53 virtual-incoming-args)}}
(\texttt{(const_int 4 [0x4])) [0 a+0 S4 A32])
(\texttt{PLUS:SI [}}
(\texttt{(mem/c/i:SI)}}
(\texttt{(reg/f:SI 53 virtual-incomming-args)}}
(\texttt{(const_int 4 [0x4])) [0 a+0 S4 A32])
(\texttt{PLUS:SI [}}
(\texttt{(mem/c/i:SI)}}
(\texttt{(reg/f:SI 53 virtual-incomming-args)}}
(\texttt{(const_int 1 [0x1])))
(\texttt{clobber (reg:CC 17 flags)) [}}
(\texttt{t1.c:25 -1 (nil))})

Offset 4 added to the argument pointer register
When $a$ is the first parameter, its offset is 0!
Output with slim suffix
\{[r53:SI]=[r53:SI]+0x1; clobber flags:CC; \}
RTL for spim: Arithmetic Operations

Translation of \( a = a + 1 \) when \( a \) is a local variable

**Dump file:** test.c.144r.expand

\[
\begin{align*}
\text{insn 7} & \quad 6 \quad 8 \quad 4 \quad \text{(set (reg:SI 39)} \\
& \quad \text{(mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)} \\
& \quad \text{(const_int -4 [...]}) [...]}) -1 \quad \text{(nil)}
\end{align*}
\]

\[
\begin{align*}
\text{insn 8} & \quad 7 \quad 9 \quad 4 \quad \text{test.c:6 (set (reg:SI 40)} \\
& \quad \text{(plus:SI (reg:SI 39)} \\
& \quad \text{(const_int 1 [...]}) -1 \quad \text{(nil)}
\end{align*}
\]

\[
\begin{align*}
\text{insn 9} & \quad 8 \quad 10 \quad 4 \quad \text{test.c:6 (set} \\
& \quad \text{(mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)} \\
& \quad \text{(const_int -4 [...]}) [...]}) \\
& \quad \text{(reg:SI 40)) test.c:6 -1 (nil)}
\end{align*}
\]

In spim, a variable is loaded into register to perform any instruction, hence three instructions are generated.

RTL for i386: Control Flow

**What does this represent?**

\[
\begin{align*}
\text{(jump_insn 15 14 16 4 (set (pc)} \\
& \quad \text{(if_then_else (lt (reg:CCGC 17 flags)} \\
& \quad \text{(const_int 0 [0x0]))} \\
& \quad \text{(label_ref 12)} \\
& \quad \text{(pc))) pl.c:6 -1 (nil)} \\
& \quad \text{-> 12)}
\end{align*}
\]

\[
\begin{align*}
p = \text{r17} < 0 ? \text{label(12)} : \text{pc}
\end{align*}
\]

Observing Register Allocation for i386

**test.c**

**test.c.188r.asmcons**

(observable dump before register allocation)

\[
\begin{align*}
\text{(insn 10} & \quad 9 \quad 11 \quad 3 \quad \text{(set (reg:SI 59))} \\
& \quad \text{(mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)} \\
& \quad \text{(const_int -4 [0xfffffffff])}) [0 a+0 S4 A32])) -1 (nil)}
\end{align*}
\]

int main()
{
    int a=2, b=3;
    if(a==12)
        a = a * b;
}

\[
\begin{align*}
\text{(insn 11} & \quad 10 \quad 12 \quad 3 \quad \text{ parallel [} \\
& \quad \text{(set (reg:SI 60))} \\
& \quad \text{(mult:SI (reg:SI 59))} \\
& \quad \text{(const_int -8 [0xfffffffff])}) [0 b+0 S4 A32])
\end{align*}
\]

\[
\begin{align*}
\text{(insn 12} & \quad 11 \quad 22 \quad 3 \quad \text{ (set} \\
& \quad \text{(reg:SI 60)) 44 *movsi_internal test.c:5 (nil)}
\end{align*}
\]
**Observing Register Allocation for i386**

```asm


(set (mem/c/i:SI (plus:SI (reg/f:SI 20 frame) (const_int -4))) (reg:SI 60))
```

**Activation Record Structure in Spim**

- **Caller's Activation Record**
  - Parameter 0
  - Parameter 1
  - ... Parameter n - 1
  - Parameter n
  - Return Address
  - Caller's FPR (Control Link)
  - Caller's SPR
  - Callee Saved Registers
  - Local Variable 1
  - Local Variable 2
  - ...
  - Local Variable n

**RTL for Function Calls in Spim**

- **Calling function**
  - Allocate memory for return value (push)
  - Store mandatory callee save registers (push)
  - Set frame pointer
  - Allocate local variables (push)
  - Execute code
  - Put result in return value space
  - Deallocate local variables (pop)
  - Load callee save registers (pop)
  - Return

- **Called function**
  - Allocate memory for actual parameters on stack
  - Copy actual parameters
  - Allocate local variables (push)
  - Execute code
  - Deallocate local variables (pop)
  - Return

**Dump file: test.c.197r.pro_and_epilogue**

```asm
(insn 17 3 18 2 (set (mem:SI (reg/f:SI 29 $sp) [0 S4 A8]) (reg:SI 31 $ra)) test.c:2 -1 (nil))

(insn 18 17 19 2 (set (mem:SI (plus:SI (reg/f:SI 29 $sp) (const_int -4)) [....])) test.c:2 -1 (nil))

(sw $ra, 0($sp))
(sw $sp, 4($sp))
(sw $fp, 8($sp))
(addi $sp,$fp,32)
```

**Prologue and Epilogue: Spim**

- Allocate memory for return value (push)
- Store mandatory callee save registers (push)
- Set frame pointer
- Allocate local variables (push)
- Execute code
- Put result in return value space
- Deallocate local variables (pop)
- Load callee save registers (pop)
- Return
Part 4

Examining Assembly Dumps

### i386 Assembly

#### Dump file: test.s

```assembly
jmp .L2
.L3:
    addl $1, -4(%ebp)
.L2:
    cmpl $7, -4(%ebp)
jle .L3
cmpl $12, -4(%ebp)
jg .L6
movl -8(%ebp), %edx
movl -4(%ebp), %eax
addl %edx, %eax
addl -12(%ebp), %eax
movl %eax, -4(%ebp)
.L6:
```

```
while (a <= 7)
{
a = a+1;
}
if (a <= 12)
{
a = a+b+c;
}
```

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Dump file: test.s

```assembly
jmp .L2
.L3:
    addl $1, -4(%ebp)
.L2:
    cmpl $7, -4(%ebp) jle .L3
    cmpl $12, -4(%ebp) jg .L6
    movl -8(%ebp), %edx
    movl -4(%ebp), %eax
    addl %edx, %eax
    addl -12(%ebp), %eax
    movl %eax, -4(%ebp)
.L6:
```

while (a <= 7) {
    a = a+1;
}
if (a <= 12) {
    a = a+b+c;
}

Conclusions

Gray Box Probing of GCC: Conclusions

- Source code is transformed into assembly by lowering the abstraction level step by step to bring it close to the machine
- This transformation can be understood to a large extent by observing inputs and output of the different steps in the transformation
- In gcc, the output of almost all the passes can be examined
- The complete list of dumps can be figured out by the command `man gcc`