

Gray Box Probing of GCC

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

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



1 July 2012

Outline

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



Outline

Notes



Part 1

Preliminaries

Notes

1 July 2012

Graybox Probing: Preliminaries

2/56

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

1 July 2012

Graybox Probing: Preliminaries

2/56

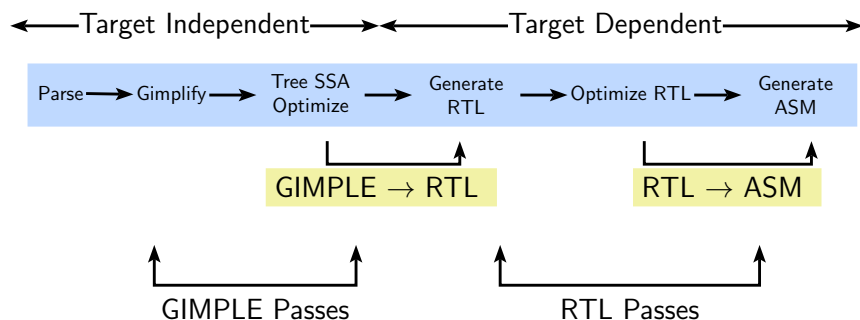
What is Gray Box Probing of GCC?

Notes



Basic Transformations in GCC

Transformation from a language to a *different* language



Transformation Passes in GCC 4.6.2

- 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 (eg. 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



Basic Transformations in GCC

Notes



Transformation Passes in GCC 4.6.2

Notes



Passes On GIMPLE in GCC 4.6.2

Pass Group	Examples	Number of passes
Lowering	GIMPLE IR, CFG Construction	10
Simple Interprocedural Passes (Non-LTO)	Conditional Constant Propagation, Inlining, SSA Construction	38
Regular Interprocedural Passes (LTO)	Constant Propagation, Inlining, Pointer Analysis	10
LTO generation passes		02
Other Intraprocedural Optimizations	Constant Propagation, Dead Code Elimination, PRE Value Range Propagation, Rename SSA	65
Loop Optimizations	Vectorization, Parallelization, Copy Propagation, Dead Code Elimination	28
Generating RTL		01
<i>Total number of passes on GIMPLE</i>		154

**Passes On RTL in GCC 4.6.2**

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

**Passes On GIMPLE in GCC 4.6.2**

Notes

**Passes On RTL in GCC 4.6.2**

Notes



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



Finding Out List of Optimizations

Notes



Producing the Output of GCC Passes

Notes



Total Number of Dumps

Optimization Level	Number of Dumps	Goals
Default	47	Fast compilation
O1	138	
O2	164	
O3	174	
Os	175	Optimize for space



Selected Dumps for Our Example Program

GIMPLE dumps (t)	138t.cplxlower0	163r.reginfo
001t.tu	143t.optimized	183r.outof_cfglayout
003t.original	224t.statistics	184r.split1
004t.gimple	ipa dumps (i)	186r.dfinit
006t.vcg	000i.cgraph	187r.mode_sw
009t.omplower	014i.visibility	188r.asmcons
010t.lower	015i.early_local_cleanups	191r.ira
012t.eh	044i.whole-program	194r.split2
013t.cfg	048i.inline	198r.pro_and_epilogue
017t.ssa	rtl dumps (r)	211r.stack
018t.veclover	144r.expand	212r.alignments
019t.inline_param1	145r.sibling	215r.mach
020t.einline	147r.initvals	216r.barriers
037t.release_ssa	148r.unshare	220r.shorten
038t.inline_param2	149r.vregs	221r.nothrow
044i.whole-program	150r.into_cfglayout	222r.final
048i.inline	151r.jump	223r.dfinish
		assembly



Total Number of Dumps

Notes

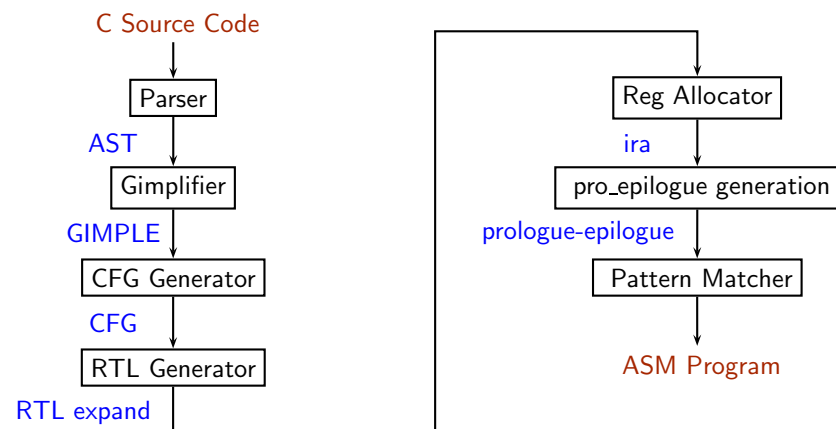


Selected Dumps for Our Example Program

Notes



Passes for First Level Graybox Probing of GCC



Lowering of abstraction!



Passes for First Level Graybox Probing of GCC

Notes



Part 2

Examining AST Dump

Notes

Generating Abstract Syntax Tree

```
$ gcc -fdump-tree-original-raw test.c
```



Abstract Syntax Tree

test.c

```
int a;
int main()
{
    a = 55;
}
```

test.c.003t.original

```
;; Function main (null)
;; enabled by /tree-original

@1 bind_expr      type: @2      body: @3
@2 void_type     name: @4      algn: 8
@3 modify_expr   type: @5      op 0: @6      op 1: @7
@4 type_decl    name: @8      type: @2
@5 integer_type name: @9      size: @10     algn: 32
                prec: 32      sign: signed  min : @11
                max : @12
@6 var_decl      name: @13     type: @5      srcp: t1.c:1
                size: @10     algn: 32      used: 1
@7 integer_cst   type: @5      low : @55
@8 identifier_node strg: void  lngt: 4
@9 type_decl    name: @14     type: @5
@10 integer_cst  type: @5      low : 32
@11 integer_cst  type: @5      high: -1      low : -2147483648
@12 integer_cst  type: @5      low : 2147483647
@13 identifier_node strg: a      lngt: 1
@14 identifier_node strg: int  lngt: 3
@15 integer_type name: @16     size: @17     algn: 64
                prec: 64      sign: unsigned min : @18
                max : @19
@16 identifier_node strg: bit_size_type lngt: 13
@17 integer_cst  type: @5      low : 64
@18 integer_cst  type: @5      low : 0
@19 integer_cst  type: @5      low : -1
```



Generating Abstract Syntax Tree

Notes



Abstract Syntax Tree

Notes



Part 3

Examining GIMPLE Dumps

Notes

1 July 2012

Graybox Probing: Examining GIMPLE Dumps

14/56

Gimplifier

- About GIMPLE
 - ▶ Three-address representation derived from GENERIC
 - Computation represented as a sequence of basic operations
 - Temporaries introduced to hold intermediate values
 - ▶ Control constructs are explicated into conditional jumps
- Examining GIMPLE Dumps
 - ▶ Examining translation of data accesses
 - ▶ Examining translation of control flow
 - ▶ Examining translation of function calls

1 July 2012

Graybox Probing: Examining GIMPLE Dumps

14/56

Gimplifier

Notes



GIMPLE: Composite Expressions Involving Local and Global Variables

test.c

```
int a;

int main()
{
  int x = 10;
  int y = 5;

  x = a + x * y;
  y = y - a * x;
}
```

test.c.004t.gimple

```
x = 10;
y = 5;
D.1954 = x * y;
a.0 = a;
x = D.1954 + a.0;
a.1 = a;
D.1957 = a.1 * x;
y = y - D.1957;
```

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



GIMPLE: Composite Expressions Involving Local and Global Variables

Notes



GIMPLE: 1-D Array Accesses

test.c

```
int main()
{
  int a[3], x;
  a[1] = a[2] = 10;
  x = a[1] + a[2];
  a[0] = a[1] + a[1]*x;
}
```

test.c.004t.gimple

```
a[2] = 10;
D.1952 = a[2];
a[1] = D.1952;
D.1953 = a[1];
D.1954 = a[2];
x = D.1953 + D.1954;
D.1955 = x + 1;
D.1956 = a[1];
D.1957 = D.1955 * D.1956;
a[0] = D.1957;
```



GIMPLE: 1-D Array Accesses

Notes



GIMPLE: 2-D Array Accesses

<pre>test.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]; }</pre>	<pre>test.c.004t.gimple a[0][0] = 7; a[1][1] = 8; a[2][2] = 9; D.1953 = a[0][0]; D.1954 = a[1][1]; x = D.1953 / D.1954; D.1955 = a[1][1]; D.1956 = a[2][2]; y = D.1955 % D.1956;</pre>
---	---

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

**GIMPLE: 2-D Array Accesses****Notes****GIMPLE: Use of Pointers**

<pre>test.c int main() { int **a,*b,c; b = &c; a = &b; **a = 10; /* c = 10 */ } ~</pre>	<pre>test.c.004t.gimple main () { int * D.1953; int * * a; int * b; int c; b = &c; a = &b; D.1953 = *a; *D.1953 = 10; }</pre>
--	---

Notes

GIMPLE: Use of Structures

<pre>test.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"; }</pre>	<pre>test.c.004t.gimple main () { void * D.1957; struct ad * D.1958; struct st * s; extern void * malloc (unsigned int); s = malloc (8); s->roll = 1; D.1957 = malloc (4); s->ct = D.1957; D.1958 = s->ct; D.1958->name = "Mumbai"; }</pre>
---	---

**GIMPLE: Use of Structures**

Notes

**GIMPLE: Pointer to Array**

<pre>test.c int main() { int *p_a, a[3]; p_a = &a[0]; *p_a = 10; *(p_a+1) = 20; *(p_a+2) = 30; }</pre>	<pre>test.c.004t.gimple main () { int * D.2048; int * D.2049; int * p_a; int a[3]; p_a = &a[0]; *p_a = 10; D.2048 = p_a + 4; *D.2048 = 20; D.2049 = p_a + 8; *D.2049 = 30; }</pre>
---	--

**GIMPLE: Pointer to Array**

Notes



GIMPLE: Translation of Conditional Statements

<pre>test.c int main() { int a=2, b=3, c=4; while (a<=7) { a = a+1; } if (a<=12) a = a+b+c; }</pre>	<pre>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>:</pre>
--	--



GIMPLE: Translation of Conditional Statements

Notes



GIMPLE: Translation of Loops

<pre>test.c int main() { int a=2, b=3, c=4; while (a<=7) { a = a+1; } if (a<=12) a = a+b+c; }</pre>	<pre>test.c.004t.gimple goto <D.1197>; <D.1196>: a = a + 1; <D.1197>: if (a <= 7) goto <D.1196>; else goto <D.1198>; <D.1198>:</pre>
--	--



GIMPLE: Translation of Loops

Notes



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>:

```

test.c.013t.cfg

```

<bb 5>:
  if (a <= 12)
    goto <bb 6>;
  else
    goto <bb 7>;
<bb 6>:
  D.1199 = a + b;
  a = D.1199 + c;
<bb 7>:
  return;

```



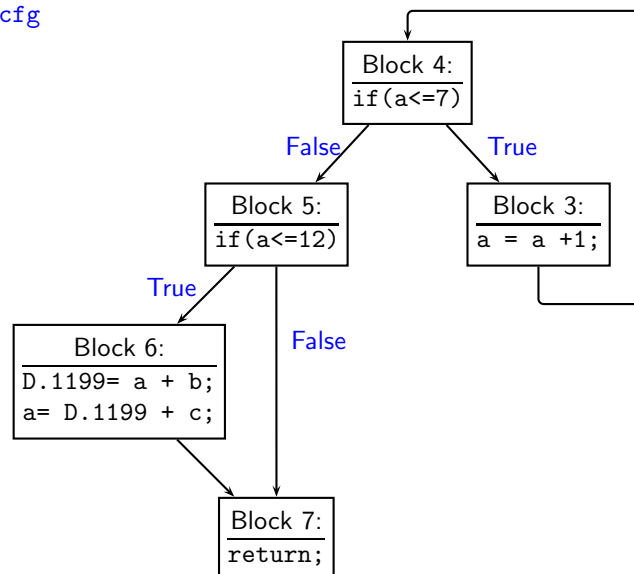
Control Flow Graph: Textual View

Notes



Control Flow Graph: Pictorial View

test.c.013t.cfg



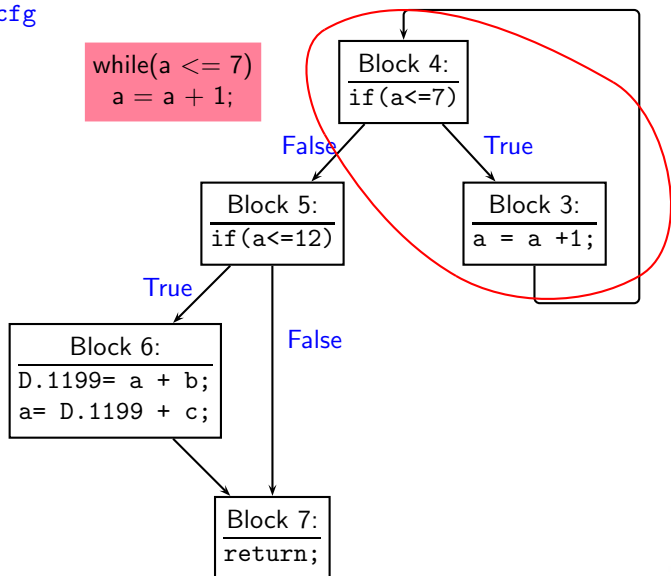
Control Flow Graph: Pictorial View

Notes



Control Flow Graph: Pictorial View

test.c.013t.cfg



Control Flow Graph: Pictorial View

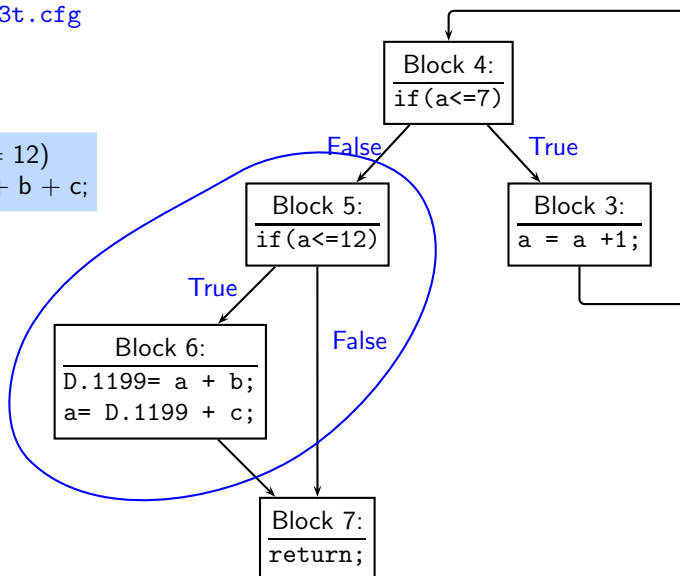
Notes



Control Flow Graph: Pictorial View

test.c.013t.cfg

if(a <= 12)
a = a + b + c;



Control Flow Graph: Pictorial View

Notes



GIMPLE: Function Calls and Call Graph

test.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);
}
```

test.c.000i.cgraph

```
printf/3(-1) @0xb73c7ac8 availability:not_availa
  called by: main/1 (1.00 per call)
  calls:
divide/2(-1) @0xb73c7a10 availability:not_availa
  called by: main/1 (1.00 per call)
  calls:
main/1(1) @0xb73c7958 availability:available 38
  called by:
  calls: printf/3 (1.00 per call)
        multiply/0 (1.00 per call)
        divide/2 (1.00 per call)
multiply/0(0) @0xb73c78a0 vailability:available
  called by: main/1 (1.00 per call)
  calls:
```

GIMPLE: Function Calls and Call Graph

Notes



GIMPLE: Function Calls and Call Graph

test.c

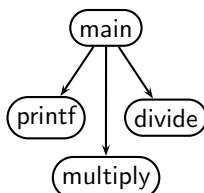
```
extern int divide(int, int);
int multiply(int a, int b)
{
    return a*b;
}

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

test.c.000i.cgraph

```
printf/3(-1)
  called by: main/1
  calls:
divide/2(-1)
  called by: main/1
  calls:
main/1(1)
  called by:
  calls: printf/3
        multiply/0
        divide/2
multiply/0(0)
  called by: main/1
  calls:
```

call graph



GIMPLE: Function Calls and Call Graph

Notes



GIMPLE: Call Graphs for Recursive Functions

test.c

```

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

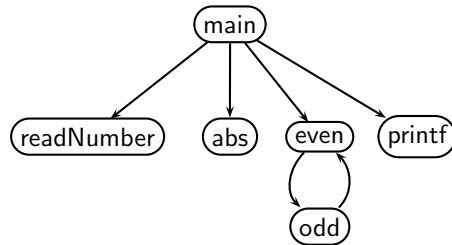
int odd(int n)
{ if (n == 1) return 1;
  else return (!even(n-1));
}

main()
{ int n;

  n = abs(readNumber());
  if (even(n))
    printf ("n is even\n");
  else printf ("n is odd\n");
}

```

call graph



GIMPLE: Call Graphs for Recursive Functions

Notes



Inspect GIMPLE When in Doubt (1)

```

int x=2,y=3;
x = y++ + ++x + ++y;

```

What are the values of x and y?

x = 10 , y =5

$$\left| \begin{array}{l} x \\ y \\ (y+x) \\ (y+x)+y \end{array} \right| \begin{array}{l} 3 \\ 3 \\ 6 \\ \end{array}$$



Inspect GIMPLE When in Doubt (1)

Notes



Inspect GIMPLE When in Doubt (1)

```
int x=2,y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10 , y =5

x	3
y	4
(y + x)	6
(y + x) + y	



Inspect GIMPLE When in Doubt (1)

```
int x=2,y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10 , y =5

x	3
y	5
(y + x)	6
(y + x) + y	



Inspect GIMPLE When in Doubt (1)

Notes



Inspect GIMPLE When in Doubt (1)

Notes



Inspect GIMPLE When in Doubt (1)

```
int x=2,y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10 , y =5

x	3
y	5
(y + x)	6
(y + x) + y	11



Inspect GIMPLE When in Doubt (1)

```
int x=2,y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10 , y =5

x	3
y	5
(y + x)	6
(y + x) + y	11

```
x = 2;
y = 3;
x = x + 1; /* 3 */
D.1572 = y + x; /* 6 */
y = y + 1; /* 4 */
x = D.1572 + y; /* 10 */
y = y + 1; /* 5 */
```



Inspect GIMPLE When in Doubt (1)

Notes



Inspect GIMPLE When in Doubt (1)

Notes



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 :

```
int * (* (*MYVAR) (int) ) [10];
```

Hint: Use `-fdump-tree-original-raw-verbose` option. The dump to see is `003t.original`



Part 4

Examining RTL Dumps

Inspect GIMPLE When in Doubt (2)

Notes



Notes

RTL for i386: Arithmetic Operations (1)

Translation of $a = a + 1$

Dump file: [test.c.144r.expand](#)

```
(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))
```



RTL for i386: Arithmetic Operations (1)

Notes

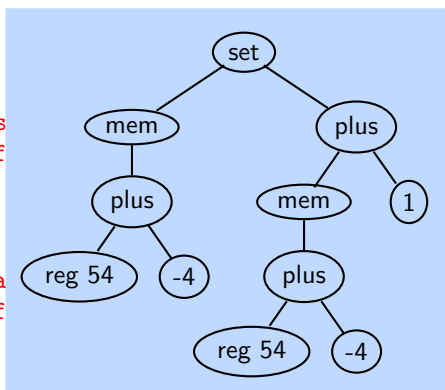


RTL for i386: Arithmetic Operations (1)

Translation of $a = a + 1$

Dump file: [test.c.144r.expand](#)

```
(insn 12 11 13 4 (parallel [
  ( set (mem/c/i:SI
    (plus:SI
      (reg/f:SI 54 virtual-s
      (const_int -4 [0xffff
    (plus:SI
      (mem/c/i:SI
        (plus:SI
          (reg/f:SI 54 virtua
          (const_int -4 [0xff
        (const_int 1 [0x1])))
      (clobber (reg:CC 17 flags))
    ]) t.c:24 -1 (nil))
```



RTL for i386: Arithmetic Operations (1)

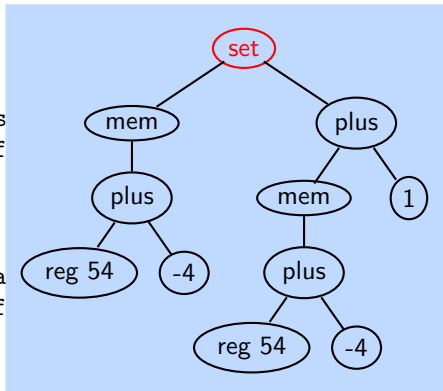
Notes



RTL for i386: Arithmetic Operations (1)

Translation of $a = a + 1$ Dump file: [test.c.144r.expand](#)

```
(insn 12 11 13 4 (parallel [
  ( set (mem/c/i:SI
    (plus:SI
      (reg/f:SI 54 virtual-s
        (const_int -4 [0xffff]
      (plus:SI
        (mem/c/i:SI
          (plus:SI
            (reg/f:SI 54 virtua
              (const_int -4 [0xff
                (const_int 1 [0x1])))
            (clobber (reg:CC 17 flags))
          ]) t.c:24 -1 (nil))
        ])
```



RTL for i386: Arithmetic Operations (1)

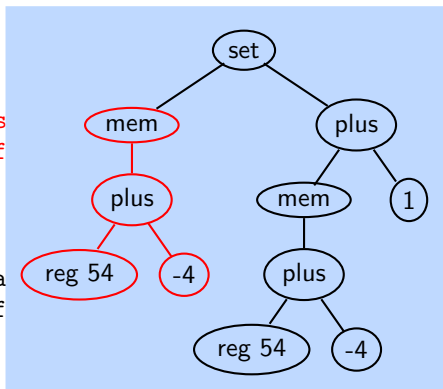
Notes



RTL for i386: Arithmetic Operations (1)

Translation of $a = a + 1$ Dump file: [test.c.144r.expand](#)

```
(insn 12 11 13 4 (parallel [
  ( set (mem/c/i:SI
    (plus:SI
      (reg/f:SI 54 virtual-s
        (const_int -4 [0xffff]
      (plus:SI
        (mem/c/i:SI
          (plus:SI
            (reg/f:SI 54 virtua
              (const_int -4 [0xff
                (const_int 1 [0x1])))
            (clobber (reg:CC 17 flags))
          ]) t.c:24 -1 (nil))
        ])
```



RTL for i386: Arithmetic Operations (1)

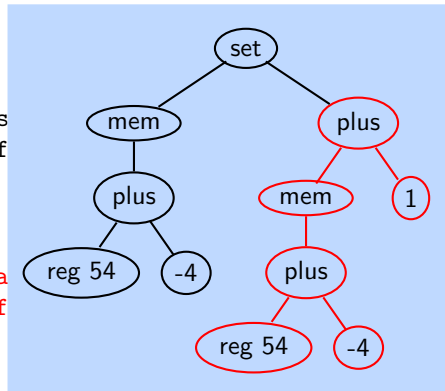
Notes



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

```
(insn 12 11 13 4 (parallel [
  ( set (mem/c/i:SI
    (plus:SI
      (reg/f:SI 54 virtual-s
        (const_int -4 [0xffff
          (plus:SI
            (mem/c/i:SI
              (plus:SI
                (reg/f:SI 54 virtua
                  (const_int -4 [0xff
                    (const_int 1 [0x1])))
              (clobber (reg:CC 17 flags))
            ]) t.c:24 -1 (nil))
        ])
```



RTL for i386: Arithmetic Operations (1)

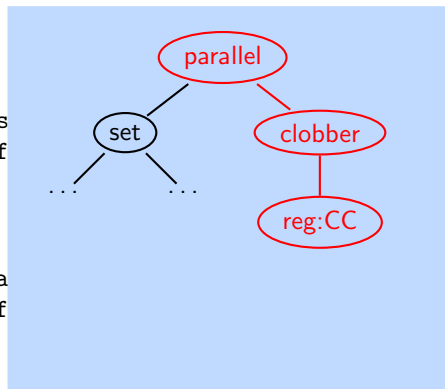
Notes



RTL for i386: Arithmetic Operations (1)

Translation of $a = a + 1$ Dump file: `test.c.144r.expand`side-effect of plus may
modify condition code register
non-deterministically

```
(insn 12 11 13 4 (parallel [
  ( set (mem/c/i:SI
    (plus:SI
      (reg/f:SI 54 virtual-s
        (const_int -4 [0xffff
          (plus:SI
            (mem/c/i:SI
              (plus:SI
                (reg/f:SI 54 virtua
                  (const_int -4 [0xff
                    (const_int 1 [0x1])))
              (clobber (reg:CC 17 flags))
            ]) t.c:24 -1 (nil))
        ])
```



RTL for i386: Arithmetic Operations (1)

Notes



RTL for i386: Arithmetic Operations (1)

Translation of $a = a + 1$

Dump file: test.c.144r.exp

Output with slim suffix

```

(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))

```



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))

```

Current Instruction



RTL for i386: Arithmetic Operations (1)

Notes



Additional Information in RTL

Notes



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])
    (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))
```

Previous Instruction



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])
    (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))
```

Next Instruction



Additional Information in RTL

Notes



Additional Information in RTL

Notes



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))
```

Basic Block



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



Additional Information in RTL

Notes



Additional Information in RTL

Notes



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])
    (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))
```

memory reference that does not trap



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])
    (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))
```

scalar that is not a part of an aggregate



Additional Information in RTL

Notes



Additional Information in RTL

Notes



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])
    (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))
```

register that holds a pointer



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])
    (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))
```

single integer



Additional Information in RTL

Notes



Additional Information in RTL

Notes



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))

(insn 12 11 13 4 (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))

(insn 13 12 14 4 (set
  (mem/c/i:SI (symbol_ref:SI ("a")
    <var_decl 0xb7d8d000 a>) [0 a+0 S4 A32])
  (reg:SI 63 [ a.1 ])) 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`

```
(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))

(insn 12 11 13 4 (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))

(insn 13 12 14 4 (set
  (mem/c/i:SI (symbol_ref:SI ("a")
    <var_decl 0xb7d8d000 a>) [0 a+0 S4 A32])
  (reg:SI 63 [ a.1 ])) t.c:26 -1 (nil))
```

Load a into reg64



RTL for i386: Arithmetic Operations (2)

Notes



RTL for i386: Arithmetic Operations (2)

Notes



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+
    ])) t.c:26 -1 (nil))

(insn 12 11 13 4 (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))

(insn 13 12 14 4 (set
  (mem/c/i:SI (symbol_ref:SI ("a")
    <var_decl 0xb7d8d000 a>) [0 a+0 S4 A32])
  (reg:SI 63 [ a.1 ])) t.c:26 -1 (nil))
```

Load a into reg64
reg63 = reg64 + 1



RTL for i386: Arithmetic Operations (2)

Notes



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+
    ])) t.c:26 -1 (nil))

(insn 12 11 13 4 (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))

(insn 13 12 14 4 (set
  (mem/c/i:SI (symbol_ref:SI ("a")
    <var_decl 0xb7d8d000 a>) [0 a+0 S4 A32])
  (reg:SI 63 [ a.1 ])) t.c:26 -1 (nil))
```

Load a into reg64
reg63 = reg64 + 1
store reg63 into a



RTL for i386: Arithmetic Operations (2)

Notes



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+
    )
  Load a into reg64
  reg63 = reg64 + 1
  store reg63 into a

  (insn 12 11 13 4 (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))
  Output with slim suffix
  r64:SI=['a']
  {r63:SI=r64:SI+0x1;
    clobber flags:CC;
  }
  ['a']=r63:SI

  (insn 13 12 14 4 (set
    (mem/c/i:SI (symbol_ref:SI ("a")
      <var_decl 0xb7d8d000 a>) [0 a+0 S4 A32])
    (reg:SI 63 [ a.1 ])) t.c:26 -1 (nil))
```



RTL for i386: Arithmetic Operations (2)

Notes



RTL for i386: Arithmetic Operations (3)

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

Dump file: [test.c.144r.expand](#)

```
(insn 10 9 11 4 (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))
```



RTL for i386: Arithmetic Operations (3)

Notes



RTL for i386: Arithmetic Operations (3)

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

Dump file: test.c.144r.expand

```
(insn 10 9 11 4 (parallel [
  (set
    (mem/c/i:SI
      (reg/f:SI 53 virtual-incoming-
        (plus:SI
          (mem/c/i:SI
            (reg/f:SI 53 virtual-incoming-
              (const_int 1 [0x1])))
          (clobber (reg:CC 17 flags))
        ] t1.c:25 -1 (nil))
```

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



RTL for i386: Arithmetic Operations (3)

Notes



RTL for i386: Arithmetic Operations (3)

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

Dump file: test.c.144r.expand

```
(insn 10 9 11 4 (parallel [
  (set
    (mem/c/i:SI
      (reg/f:SI 53 virtual-incoming-
        (plus:SI
          (mem/c/i:SI
            (reg/f:SI 53 virtual-incoming-
              (const_int 1 [0x1])))
          (clobber (reg:CC 17 flags))
        ] t1.c:25 -1 (nil))
```

Access through argument
pointer register instead of
frame pointer register
No offset required?
Output with slim suffix
{ [r53:SI]=[r53:SI]+0x1;
clobber flags:CC;
}



RTL for i386: Arithmetic Operations (3)

Notes



RTL for i386: Arithmetic Operation (4)

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

Dump file: [test.c.144r.expand](#)

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



RTL for i386: Arithmetic Operation (4)

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

Dump file: [test.c.144r.expand](#)

```
(insn 10 9 11 4 (parallel [
  (set
    (mem/c/i:SI
      (plus:SI
        (reg/f:SI 53 virtual-
        (const_int 4 [0x4])))
    (plus:SI
      (mem/c/i:SI
        (plus:SI
          (reg/f:SI 53 virtu
          (const_int 4 [0x4]
        (const_int 1 [0x1])))
      (clobber (reg:CC 17 flags))
]) 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+0x4]=[r53:SI+0x4]+0x1;
  clobber flags:CC;
}
```



RTL for i386: Arithmetic Operation (4)

Notes



RTL for i386: Arithmetic Operation (4)

Notes



RTL for spim: Arithmetic Operations

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

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

```
r39=stack($fp - 4)
r40=r39+1
stack($fp - 4)=r40
```

```
(insn 7 6 8 4 (set (reg:SI 39)
  (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
    (const_int -4 [...])) [...])) -1 (nil))
(insn 8 7 9 4 test.c:6 (set (reg:SI 40)
  (plus:SI (reg:SI 39)
    (const_int 1 [...]))) -1 (nil))
(insn 9 8 10 4 test.c:6 (set
  (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
    (const_int -4 [...])) [...])
  (reg:SI 40)) test.c:6 -1 (nil))
```

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



RTL for spim: Arithmetic Operations

Notes



RTL for i386: Control Flow

What does this represent?

```
(jump_insn 15 14 16 4 (set (pc)
  (if_then_else (lt (reg:CCGC 17 flags)
    (const_int 0 [0x0]))
    (label_ref 12)
    (pc))) p1.c:6 -1 (nil)
  (nil)
  -> 12)
```

$pc = r17 < 0 ? label(12) : pc$



RTL for i386: Control Flow

Notes



RTL for i386: Control Flow

Translation of `if (a > b) { /* something */ }`

Dump file: [test.c.144r.expand](#)

```
(insn 8 7 9 (set (reg:SI 61)
  (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
    (const_int -8 [0xffffffff8])) [0 a+0 S4 A32])) test.c:7 -1 (nil))
(insn 9 8 10 (set (reg:CCGC 17 flags)
  (compare:CCGC (reg:SI 61)
    (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
      (const_int -4 [0xffffffffc])) [0 b+0 S4 A32]))) test.c:7 -1 (nil))
(jump_insn 10 9 0 (set (pc)
  (if_then_else (le (reg:CCGC 17 flags)
    (const_int 0 [0x0]))
    (label_ref 13)
    (pc))) test.c:7 -1 (nil)
-> 13)
```



Part 5

Examining Assembly Dumps

RTL for i386: Control Flow

Notes



Notes

i386 Assembly

Dump file: test.s

```

        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;
}

```

**i386 Assembly**

Notes

**i386 Assembly**

Dump file: test.s

```

        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;
}

```

**i386 Assembly**

Notes



i386 Assembly

Dump file: test.s

```

        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;
}

```

**i386 Assembly**

Notes

**i386 Assembly**

Dump file: test.s

```

        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;
}

```

**i386 Assembly**

Notes



Examining GIMPLE Optimization

Notes

Example Program for Observing Optimizations

```
int main()
{ int a, b, c, n;

  a = 1;
  b = 2;
  c = 3;
  n = c*2;
  while (a <= n)
  {
    a = a+1;
  }
  if (a < 12)
    a = a+b+c;
  return a;
}
```

- What does this program return?
- 12
- We use this program to illustrate various shades of the following optimizations:
Constant propagation, Copy propagation, Loop unrolling, Dead code elimination



Example Program for Observing Optimizations

Notes



Compilation Command

```
$gcc -fdump-tree-all -O2 ccp.c
```



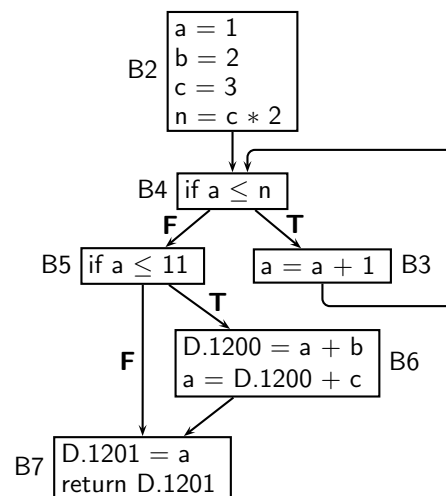
Example Program 1

Program ccp.c

```
int main()
{ int a, b, c, n;

  a = 1;
  b = 2;
  c = 3;
  n = c*2;
  while (a <= n)
  {
    a = a+1;
  }
  if (a < 12)
    a = a+b+c;
  return a;
}
```

Control flow graph



Compilation Command

Notes



Example Program 1

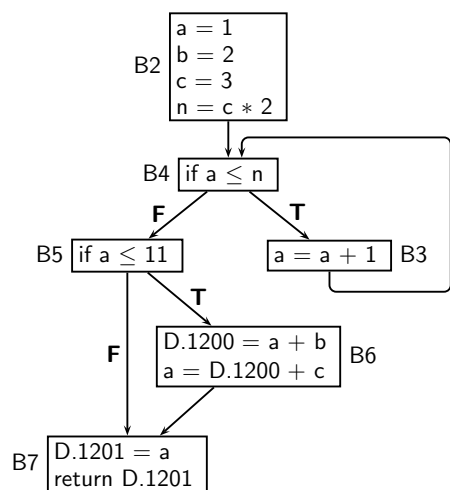
Notes



Control Flow Graph: Pictorial and Textual View

Control flow graph

Dump file ccp.c.013t.cfg



Control Flow Graph: Pictorial and Textual View

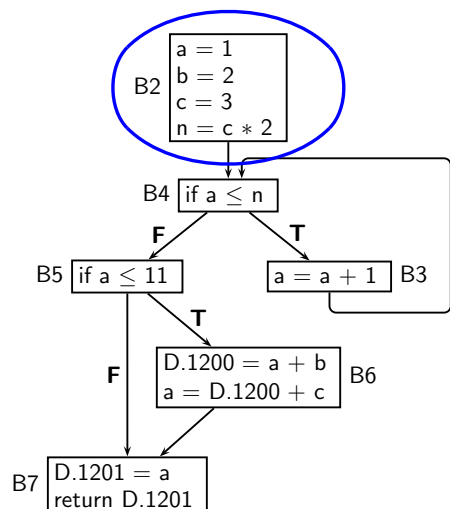
Notes



Control Flow Graph: Pictorial and Textual View

Control flow graph

Dump file ccp.c.013t.cfg



```

<bb 2>:
a = 1;
b = 2;
c = 3;
n = c * 2;
goto <bb 4>;
  
```



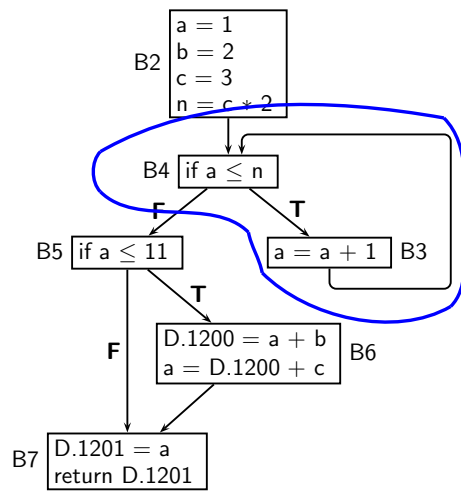
Control Flow Graph: Pictorial and Textual View

Notes



Control Flow Graph: Pictorial and Textual View

Control flow graph



Dump file ccp.c.013t.cfg

```

<bb 3>:
a = a + 1;

<bb 4>:
if (a <= n)
  goto <bb 3>;
else
  goto <bb 5>;
  
```



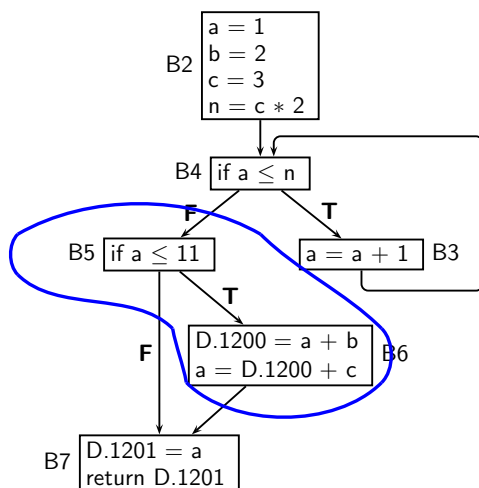
Control Flow Graph: Pictorial and Textual View

Notes



Control Flow Graph: Pictorial and Textual View

Control flow graph



Dump file ccp.c.013t.cfg

```

<bb 5>:
if (a <= 11)
  goto <bb 6>;
else
  goto <bb 7>;

<bb 6>:
D.1200 = a + b;
a = D.1200 + c;
  
```



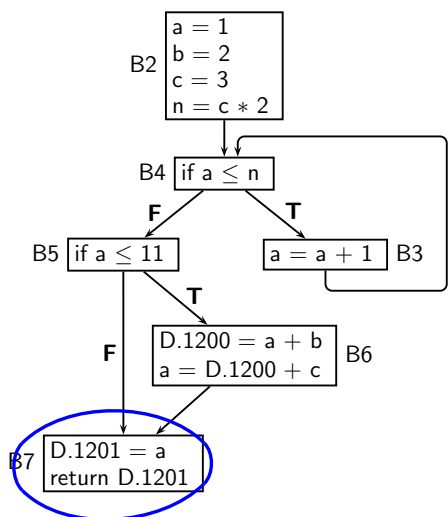
Control Flow Graph: Pictorial and Textual View

Notes



Control Flow Graph: Pictorial and Textual View

Control flow graph



Dump file ccp.c.013t.cfg

```

<bb 7>:
D.1201 = a;
return D.1201;
  
```



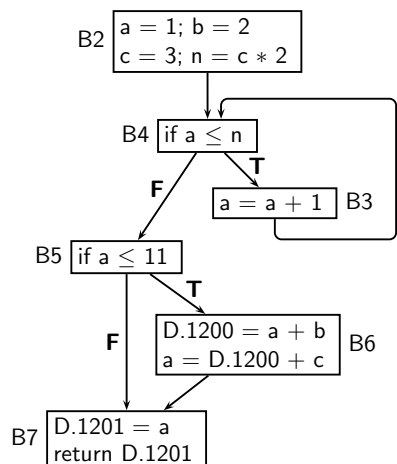
Control Flow Graph: Pictorial and Textual View

Notes

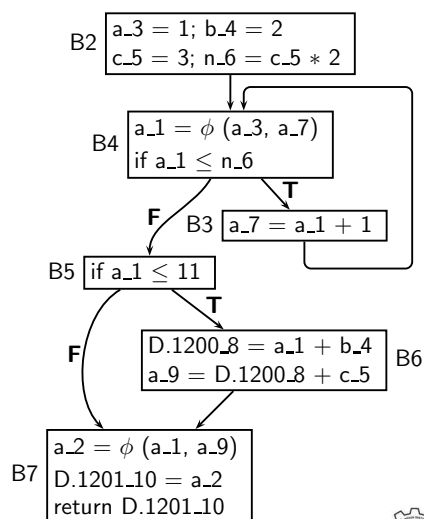


Single Static Assignment (SSA) Form

Control flow graph



SSA Form



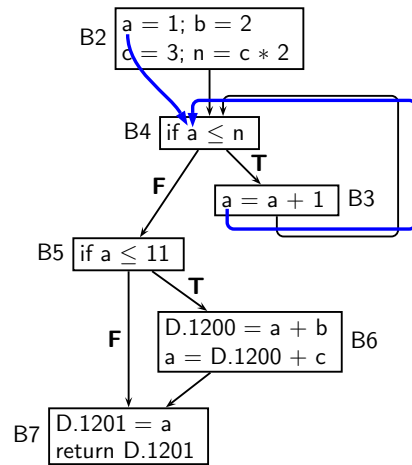
Single Static Assignment (SSA) Form

Notes

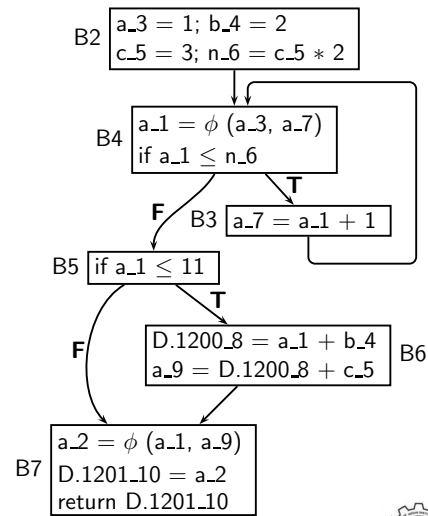


Single Static Assignment (SSA) Form

Control flow graph



SSA Form



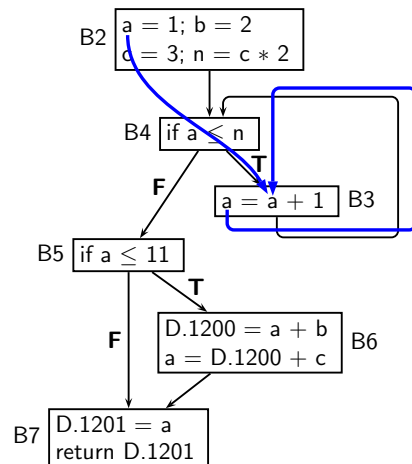
Single Static Assignment (SSA) Form

Notes

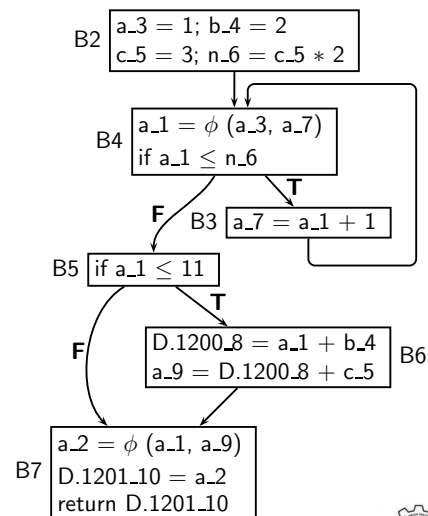


Single Static Assignment (SSA) Form

Control flow graph



SSA Form



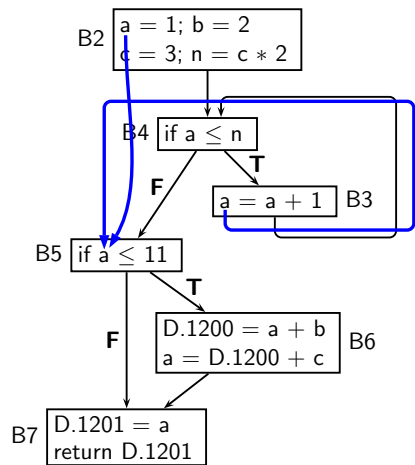
Single Static Assignment (SSA) Form

Notes

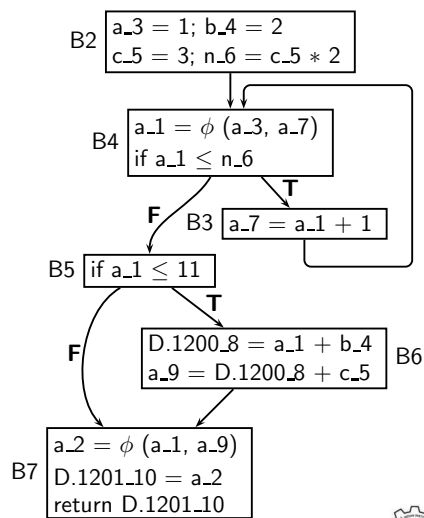


Single Static Assignment (SSA) Form

Control flow graph



SSA Form



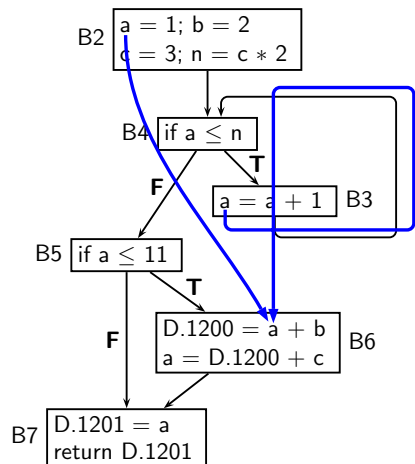
Single Static Assignment (SSA) Form

Notes

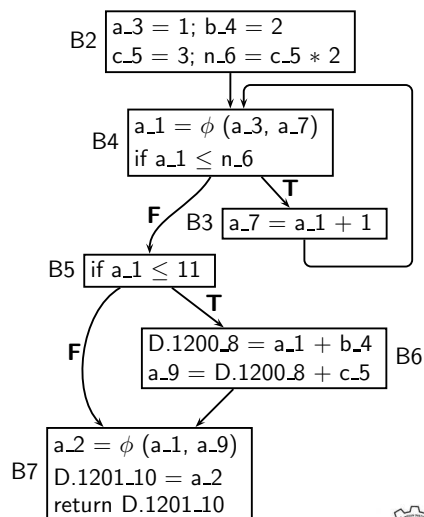


Single Static Assignment (SSA) Form

Control flow graph



SSA Form



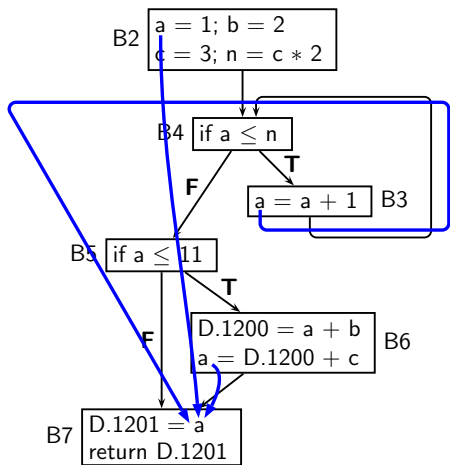
Single Static Assignment (SSA) Form

Notes

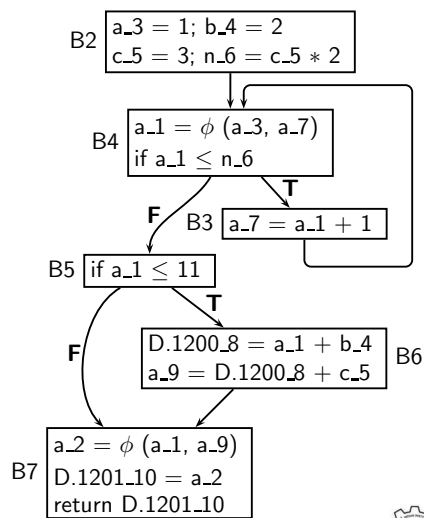


Single Static Assignment (SSA) Form

Control flow graph



SSA Form

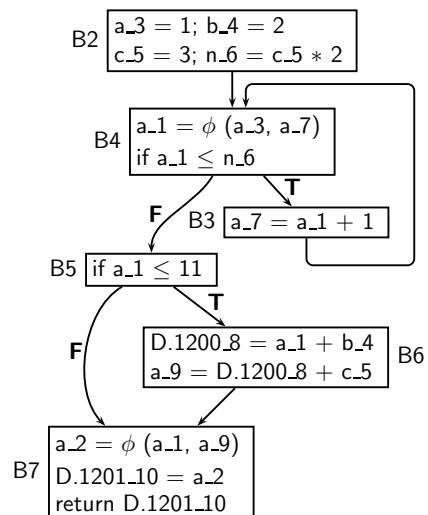


Single Static Assignment (SSA) Form

Notes



Properties of SSA Form



- A ϕ function is a multiplexer or a selection function
- Every use of a variable corresponds to a unique definition of the variable
- For every use, the definition is guaranteed to appear on every path leading to the use

SSA construction algorithm is expected to insert as few ϕ functions as possible to ensure the above properties

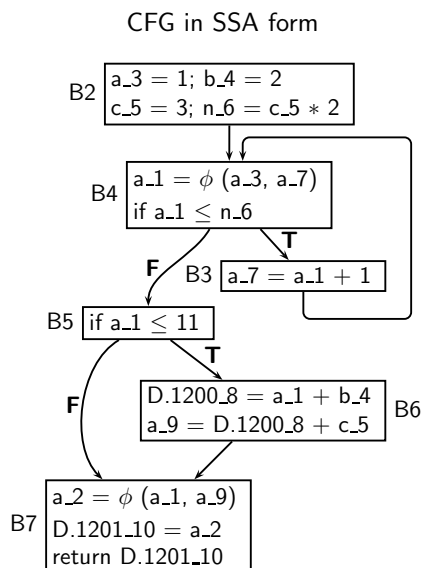


Properties of SSA Form

Notes



SSA Form: Pictorial and Textual View



Dump file ccp.c.017t.ssa

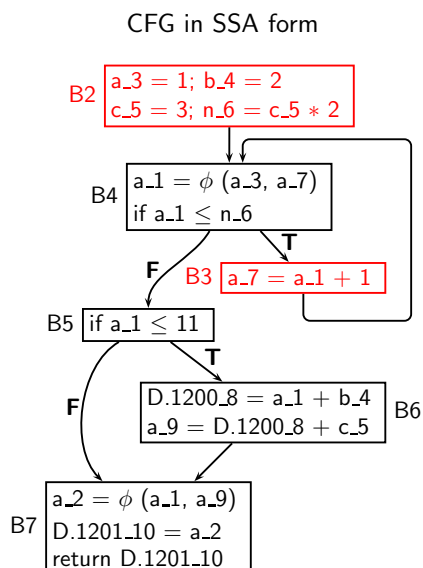


SSA Form: Pictorial and Textual View

Notes



SSA Form: Pictorial and Textual View



Dump file ccp.c.017t.ssa

```

<bb 2>:
a_3 = 1;
b_4 = 2;
c_5 = 3;
n_6 = c_5 * 2;
goto <bb 4>;

<bb 3>:
a_7 = a_1 + 1;
  
```

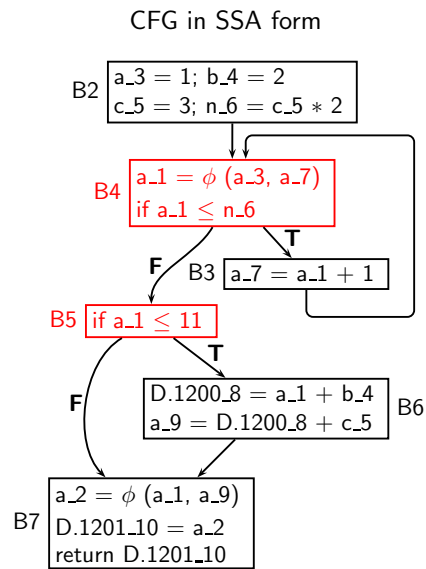


SSA Form: Pictorial and Textual View

Notes



SSA Form: Pictorial and Textual View



Dump file ccp.c.017t.ssa

```

<bb 4>:
# a_1 = PHI <a_3(2), a_7(3)>
if (a_1 <= n_6)
  goto <bb 3>;
else
  goto <bb 5>;

<bb 5>:
if (a_1 <= 11)
  goto <bb 6>;
else
  goto <bb 7>;
  
```

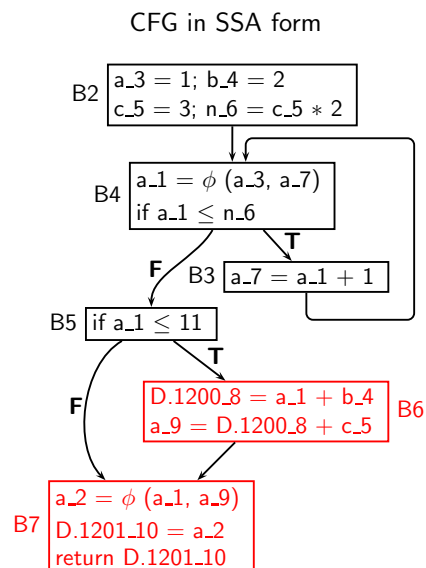


SSA Form: Pictorial and Textual View

Notes



SSA Form: Pictorial and Textual View



Dump file ccp.c.017t.ssa

```

<bb 6>:
D.1200_8 = a_1 + b_4;
a_9 = D.1200_8 + c_5;

<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
D.1201_10 = a_2;
return D.1201_10;
  
```

Notes



A Comparison of CFG and SSA Dumps

Dump file ccp.c.013t.cfg

```
<bb 2>:
  a = 1;
  b = 2;
  c = 3;
  n = c * 2;
  goto <bb 4>;

<bb 3>:
  a = a + 1;
```

Dump file ccp.c.017t.ssa

```
<bb 2>:
  a_3 = 1;
  b_4 = 2;
  c_5 = 3;
  n_6 = c_5 * 2;
  goto <bb 4>;

<bb 3>:
  a_7 = a_1 + 1;
```



A Comparison of CFG and SSA Dumps

Notes



A Comparison of CFG and SSA Dumps

Dump file ccp.c.013t.cfg

```
<bb 4>:
  if (a <= n)
    goto <bb 3>;
  else
    goto <bb 5>;

<bb 5>:
  if (a <= 11)
    goto <bb 6>;
  else
    goto <bb 7>;
```

Dump file ccp.c.017t.ssa

```
<bb 4>:
  # a_1 = PHI <a_3(2), a_7(3)>
  if (a_1 <= n_6)
    goto <bb 3>;
  else
    goto <bb 5>;

<bb 5>:
  if (a_1 <= 11)
    goto <bb 6>;
  else
    goto <bb 7>;
```



A Comparison of CFG and SSA Dumps

Notes



A Comparison of CFG and SSA Dumps

Dump file ccp.c.013t.cfg

```
<bb 6>:
D.1200 = a + b;
a = D.1200 + c;
```

```
<bb 7>:
D.1201 = a;
return D.1201;
```

Dump file ccp.c.017t.ssa

```
<bb 6>:
D.1200_8 = a_1 + b_4;
a_9 = D.1200_8 + c_5;
```

```
<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
D.1201_10 = a_2;
return D.1201_10;
```



A Comparison of CFG and SSA Dumps

Notes



Copy Renaming

Input dump: ccp.c.017t.ssa

```
<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
D.1201_10 = a_2;
return D.1201_10;
```

Output dump: ccp.c.022t.copyrename1

```
<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
a_10 = a_2;
return a_10;
```



Copy Renaming

Notes



First Level Constant and Copy Propagation

Input dump: ccp.c.022t.copyrename1

```
<bb 2>:
  a_3 = 1;
  b_4 = 2;
  c_5 = 3;
  n_6 = c_5 * 2;
  goto <bb 4>;

<bb 3>:
  a_7 = a_1 + 1;

<bb 4>:
  # a_1 = PHI < a_3(2), a_7(3)>
  if (a_1 <= n_6)
    goto <bb 3>;
  else
    goto <bb 5>;
```

Output dump: ccp.c.023t.ccp1

```
<bb 2>:
  a_3 = 1;
  b_4 = 2;
  c_5 = 3;
  n_6 = 6;
  goto <bb 4>;

<bb 3>:
  a_7 = a_1 + 1;

<bb 4>:
  # a_1 = PHI < 1(2), a_7(3)>
  if (a_1 <= 6)
    goto <bb 3>;
  else
    goto <bb 5>;
```



First Level Constant and Copy Propagation

Notes



First Level Constant and Copy Propagation

Input dump: ccp.c.022t.copyrename1

```
<bb 2>:
  a_3 = 1;
  b_4 = 2;
  c_5 = 3;
  n_6 = 6;
  goto <bb 4>;

...

<bb 6>:
  D.1200_8 = a_1 + b_4;
  a_9 = D.1200_8 + c_5;
```

Output dump: ccp.c.023t.ccp1

```
<bb 2>:
  a_3 = 1;
  b_4 = 2;
  c_5 = 3;
  n_6 = 6;
  goto <bb 4>;

...

<bb 6>:
  D.1200_8 = a_1 + 2;
  a_9 = D.1200_8 + 3;
```



First Level Constant and Copy Propagation

Notes



Second Level Copy Propagation

Input dump: ccp.c.023t.ccp1

```
<bb 6>:
D.1200_8 = a_1 + 2;
a_9 = D.1200_8 + 3;

<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
a_10 = a_2;
return a_10;
```

Output dump: ccp.c.027t.copyprop1

```
<bb 6>:
a_9 = a_1 + 5;

<bb 7>:
# a_2 = PHI <a_1(5), a_9(6)>
return a_2;
```

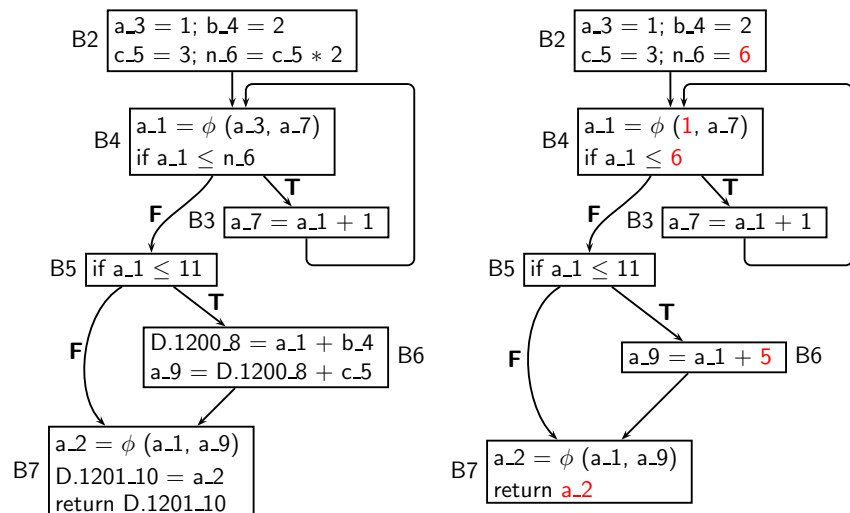


Second Level Copy Propagation

Notes



The Result of Copy Propagation and Renaming

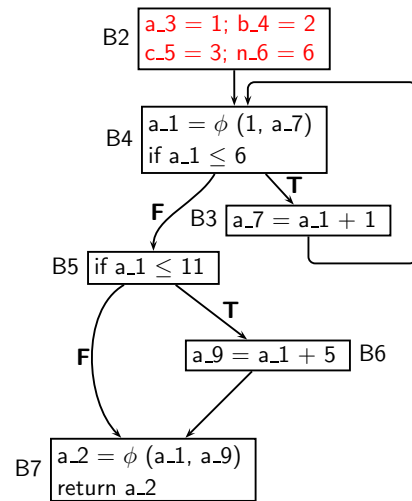


The Result of Copy Propagation and Renaming

Notes



The Result of Copy Propagation and Renaming



- No uses for variables a_3, b_4, c_5, and n_6
- Assignments to these variables can be deleted

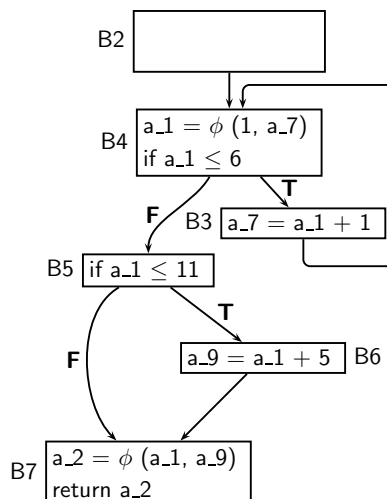


The Result of Copy Propagation and Renaming

Notes



Dead Code Elimination Using Control Dependence



Dump file [ccp.c.029t.cddce1](#)

```

<bb 2>:
  goto <bb 4>;
<bb 3>:
  a_7 = a_1 + 1;
<bb 4>:
  # a_1 = PHI <1(2), a_7(3)>
  if (a_1 <= 6) goto <bb 3>;
  else goto <bb 5>;
<bb 5>:
  if (a_1 <= 11) goto <bb 6>;
  else goto <bb 7>;
<bb 6>:
  a_9 = a_1 + 5;
<bb 7>:
  # a_2 = PHI <a_1(5), a_9(6)>
  return a_2;

```

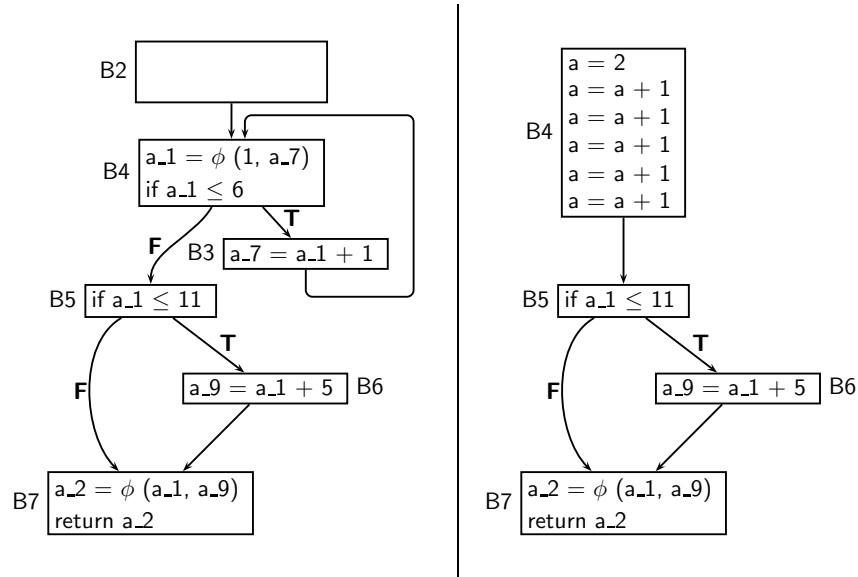


Dead Code Elimination Using Control Dependence

Notes



Loop Unrolling



Loop Unrolling

Notes



Complete Unrolling of Inner Loops

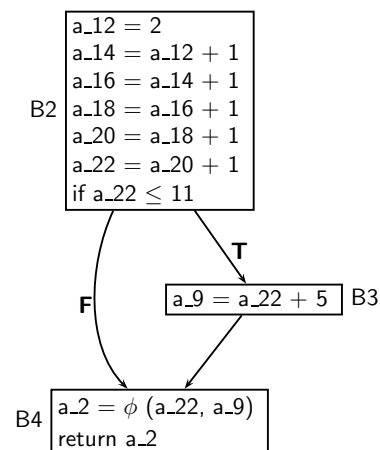
Dump file: [ccp.c.058t.cunrolli](#)

```

<bb 2>:
  a_12 = 2;
  a_14 = a_12 + 1;
  a_16 = a_14 + 1;
  a_18 = a_16 + 1;
  a_20 = a_18 + 1;
  a_22 = a_20 + 1;
  if (a_22 <= 11) goto <bb 3>;
  else goto <bb 4>;

<bb 3>:
  a_9 = a_22 + 5;

<bb 4>:
  # a_2 = PHI <a_22(2), a_9(3)>
  return a_2;
  
```



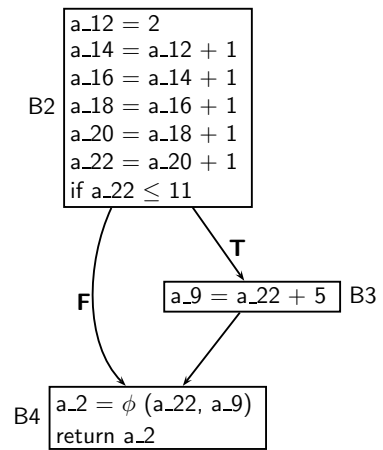
Complete Unrolling of Inner Loops

Notes



Another Round of Constant Propagation

Input

Dump file: `ccp.c.059t.ccp2`

```

main ()
{
  <bb 2>:
  return 12;
}
  
```



Part 7

Conclusions

Another Round of Constant Propagation

Notes



Notes

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
- It is easy to prepare interesting test cases and observe the effect of transformations
- One optimization often leads to another
Hence GCC performs many optimizations repeatedly
(eg. copy propagation, dead code elimination)



Gray Box Probing of GCC: Conclusions

Notes

