Workshop on Essential Abstractions in GCC

Graybox Probing for Machine Independent Optimizations

GCC Resource Center

(www.cse.iitb.ac.in/grc)

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

30 June 2011

Part 1

First Example Program

Outline

- Example 1
  - Constant Propagation
  - Copy Propagation
  - Dead Code Elimination
  - Loop unrolling
- Example 2
  - Partial Redundancy Elimination
  - Copy Propagation
  - Dead Code Elimination

Example Program 1

```
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

Compilation Command

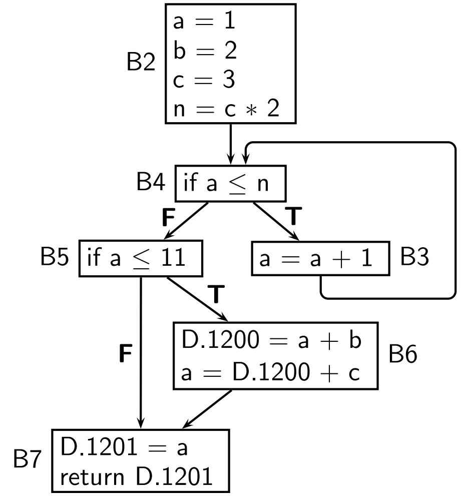
```
$gcc -fdump-tree-all  -O2 ccp.c
```


Example Program 1

Program ccp.c

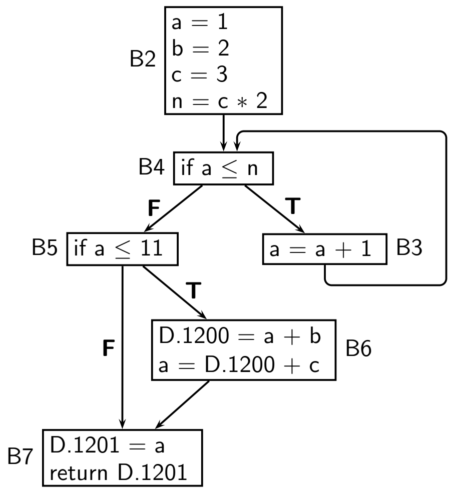
```
int main()
{ int a, b, c, n;

  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



Control Flow Graph: Pictorial and Textual View

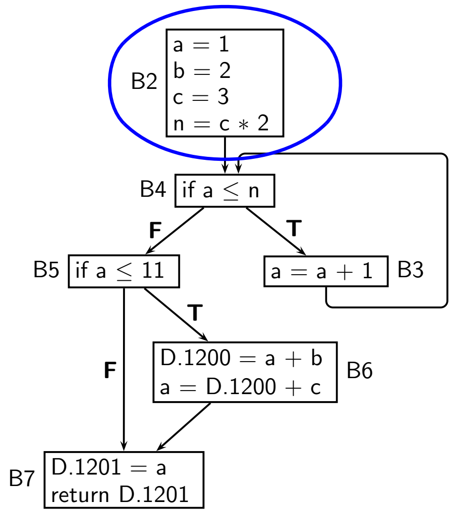
Control flow graph



Dump file ccp.c.013t.cfg

Control Flow Graph: Pictorial and Textual View

Control flow graph

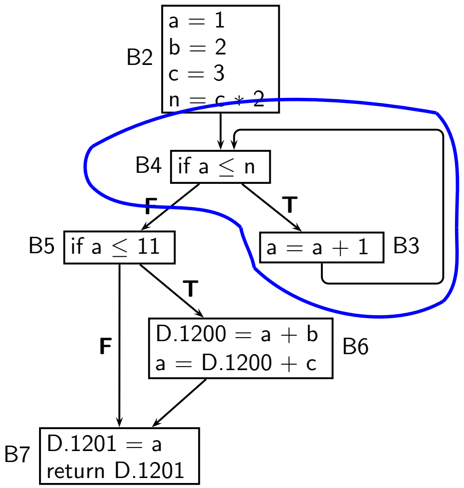


Dump file ccp.c.013t.cfg

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

Control Flow Graph: Pictorial and Textual View

Control flow graph



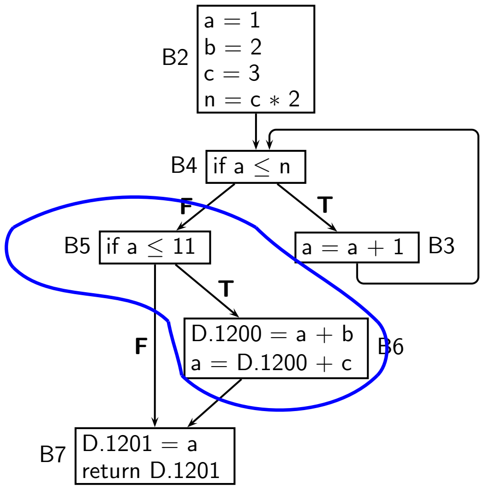
Dump file ccp.c.013t.cfg

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

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


30 June 2011 Machine Independent Optimizations: First Example Program 5/29

Control Flow Graph: Pictorial and Textual View

Control flow graph



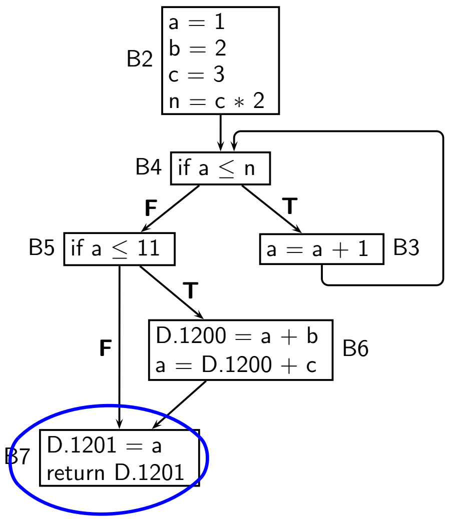
Dump file ccp.c.013t.cfg

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

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


30 June 2011 Machine Independent Optimizations: First Example Program 5/29

Control Flow Graph: Pictorial and Textual View

Control flow graph

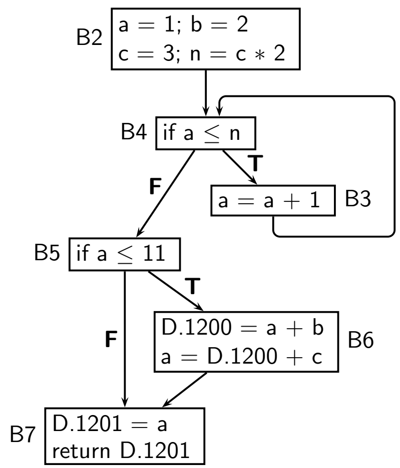


Dump file ccp.c.013t.cfg

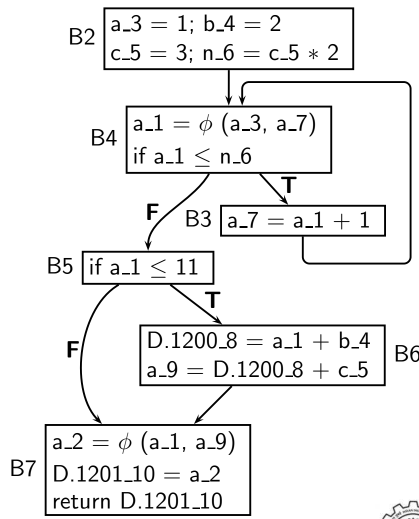
```
<bb 7>:
D.1201 = a;
return D.1201;
```


Single Static Assignment (SSA) Form

Control flow graph

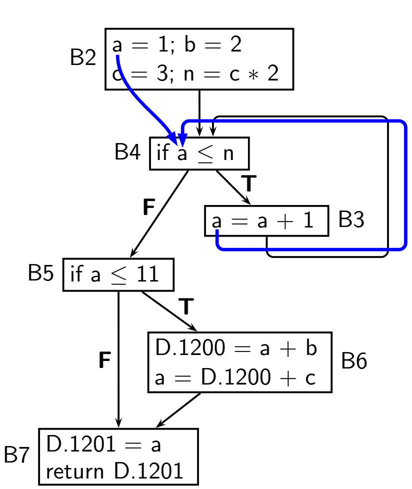


SSA Form

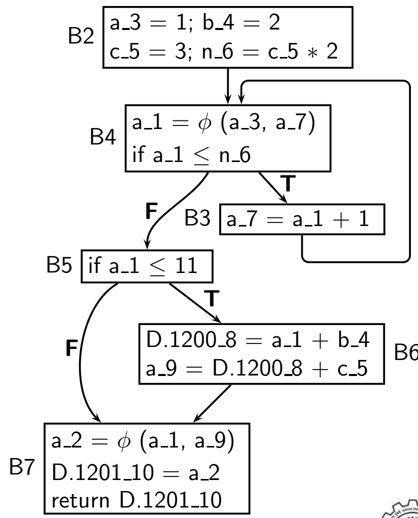


Single Static Assignment (SSA) Form

Control flow graph

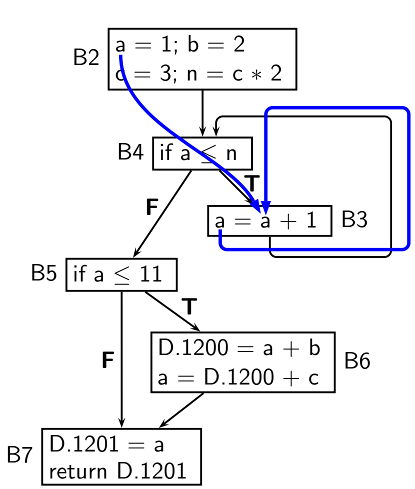


SSA Form

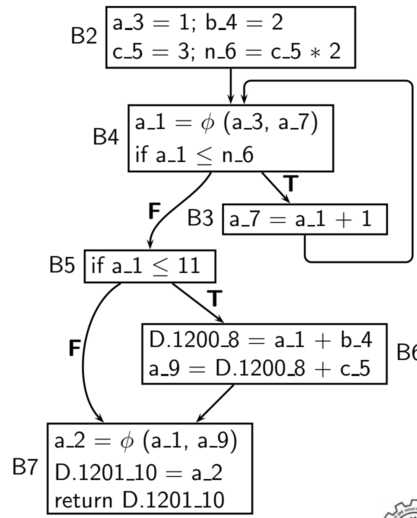


Single Static Assignment (SSA) Form

Control flow graph

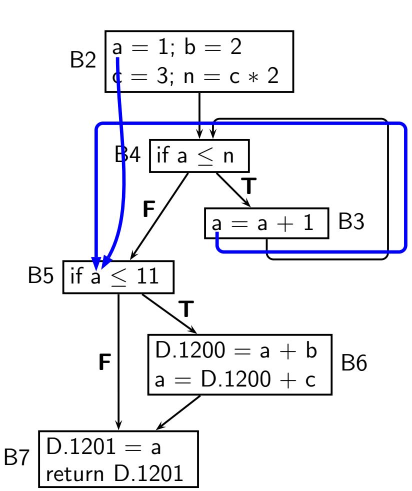


SSA Form

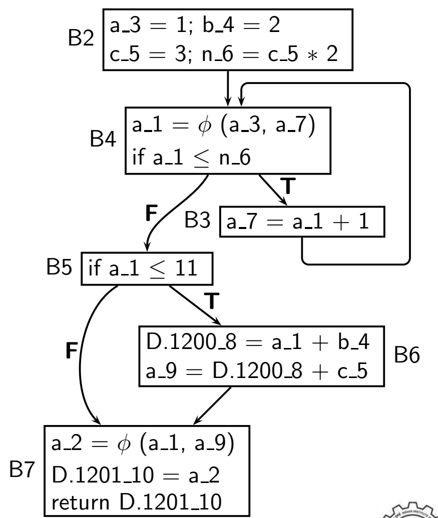


Single Static Assignment (SSA) Form

Control flow graph

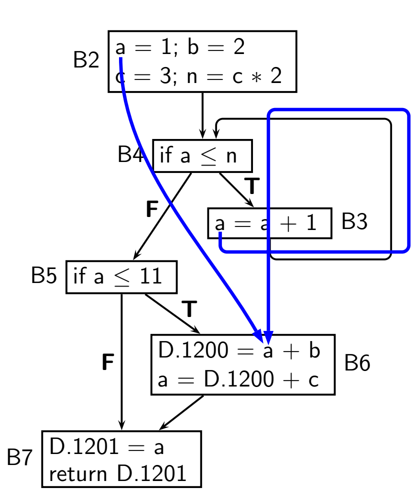


SSA Form

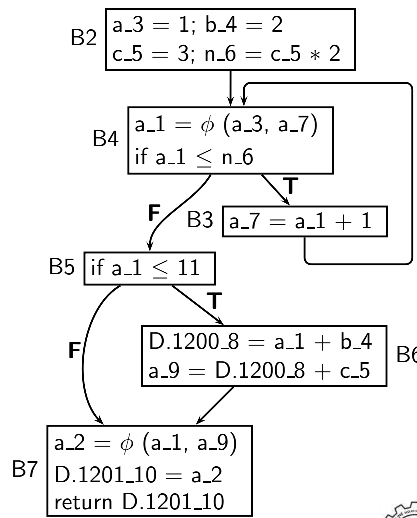


Single Static Assignment (SSA) Form

Control flow graph

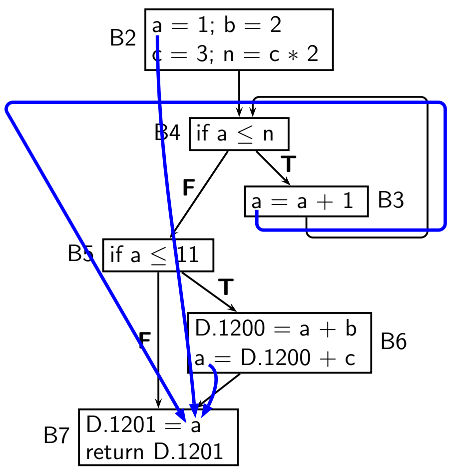


SSA Form

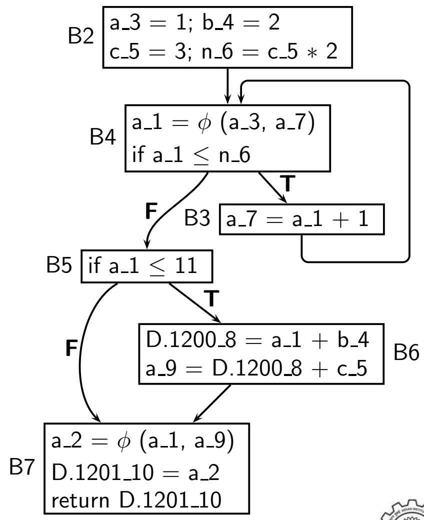


Single Static Assignment (SSA) Form

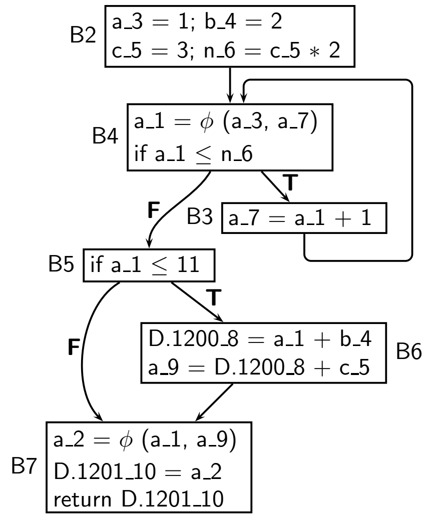
Control flow graph



SSA Form



Properties of SSA Form

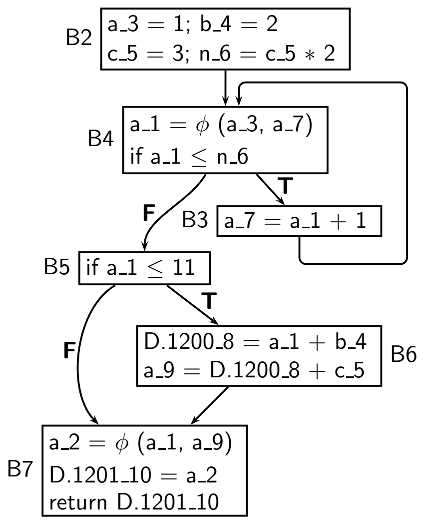


- A ϕ function is a multiplexer or a selection function
- Every use of a variable corresponds to a unique definition of the variable
- For every use, the definition is guaranteed to appear on every path leading to the use

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

SSA Form: Pictorial and Textual View

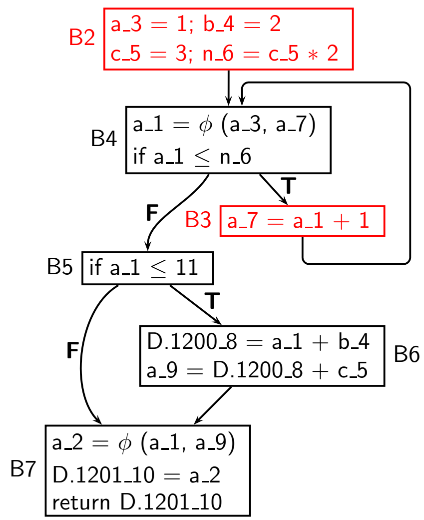
CFG in SSA form



Dump file ccp.c.017t.ssa

SSA Form: Pictorial and Textual View

CFG in SSA form

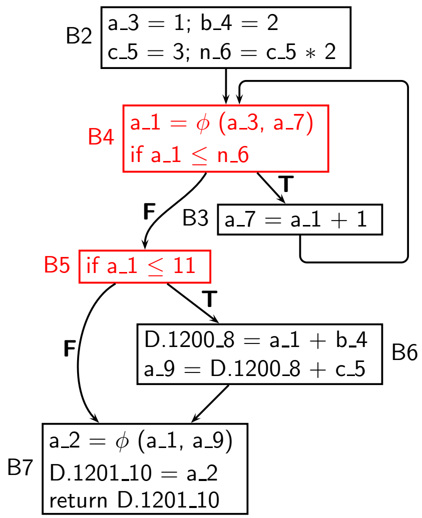


Dump file ccp.c.017t.ssa

```
<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

CFG in SSA form

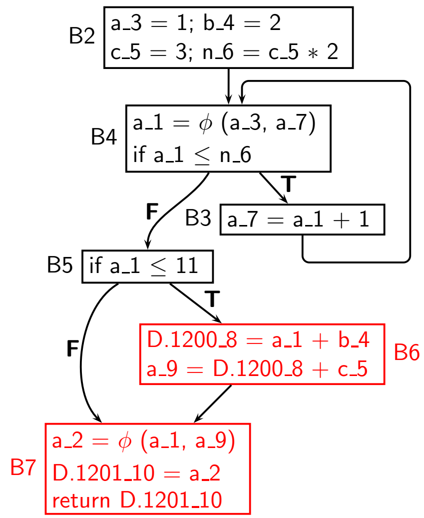


Dump file ccp.c.017t.ssa

```
<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

CFG in SSA form



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

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

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

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;
```


Copy Renamimg

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;
```


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

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;
```

Second Level Copy Propagation

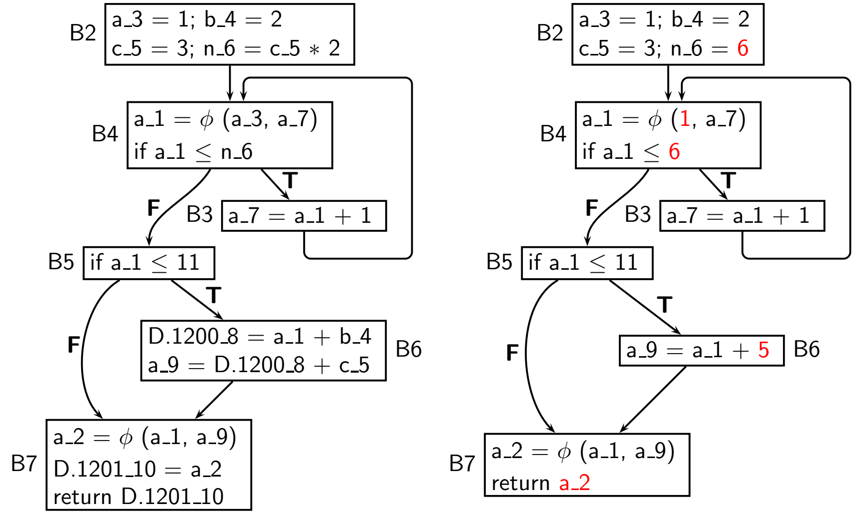
Input dump: ccp.c.023t.ccp1

```
<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;
```

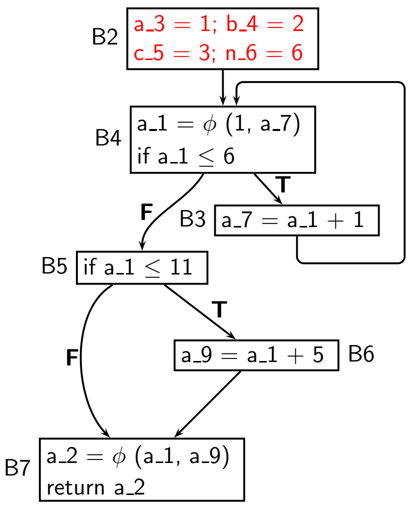
Output dump: ccp.c.027t.copyprop1

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

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


The Result of Copy Propagation and Renaming

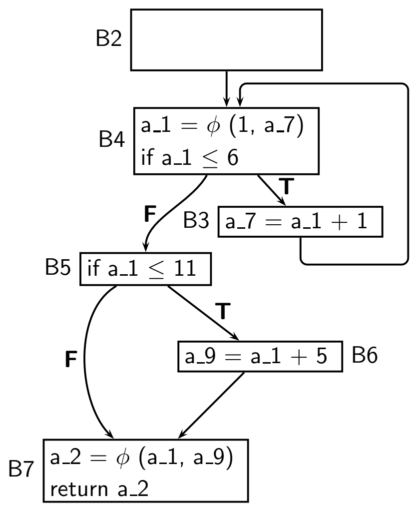


The Result of Copy Propagation and Renaming



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

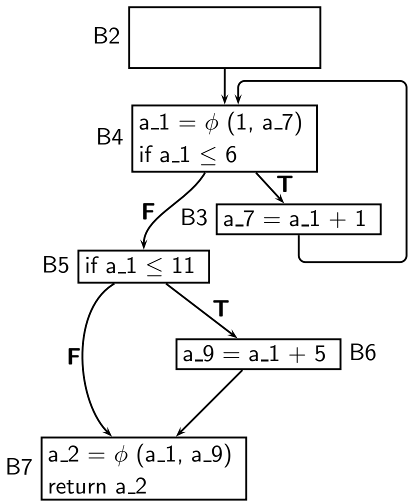
Dead Code Elimination Using Control Dependence

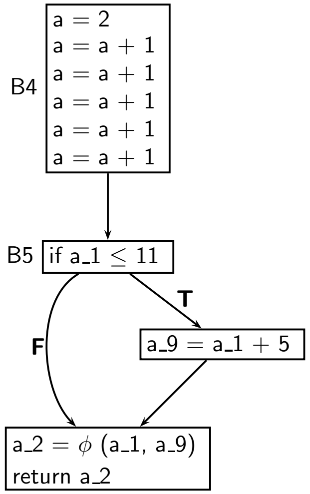


Dump file ccp.c.029t.cddce1

```
<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;
```

Loop Unrolling

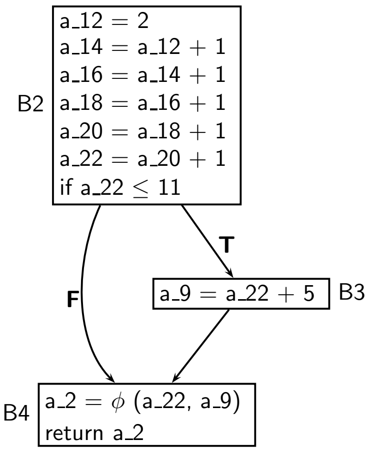




Complete Unrolling of Inner Loops

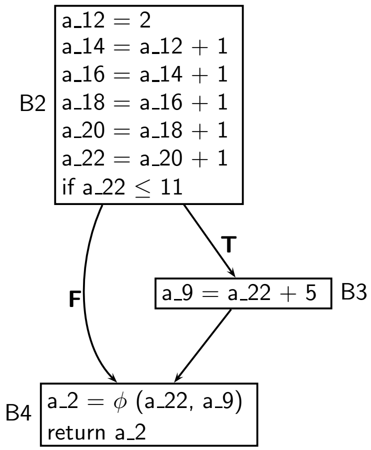
Dump file: ccp.c.058t.cunrolli

```
<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;
```



Another Round of Constant Propagation

Input



Dump file: ccp.c.059t.ccp2

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

}
```


Part 2

Second Example Program

Example Program 2

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

   do
   {
      a = b+c;
   }
   while (a <= n);

   return a;
}
```

We use this program to illustrate the following optimizations:

Partial Redundancy Elimination, Copy Propagation, Dead Code Elimination

Compilation Command

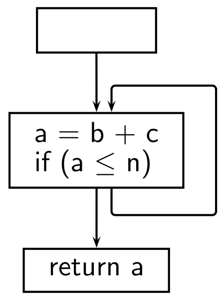
```
$gcc -fdump-tree-all  -O2 -S ccp.c
```


Example Program 2

loop.c

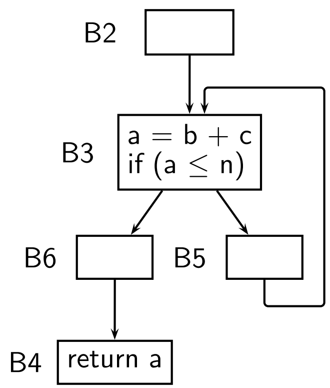
```
int f(int b, int c, int n)
{ int a;

   do
   {
      a = b+c;
   }
   while (a <= n);

   return a;
}
```

Control Flow Graph



Dump of Input to PRE Pass

Control Flow Graph



loop.c.091t.crited

```
<bb 2>:

<bb 3>:
  a_3 = b_1(D) + c_2(D);
  if (a_3 <= n_4(D)) goto <bb 5>;
  else goto <bb 6>;

<bb 5>:
  goto <bb 3>;

<bb 6>:

<bb 4>:
  # a_6 = PHI <a_3(6)>
  return a_6;
```

Input and Output of PRE Pass

loop.c.091t.crited

```
<bb 2>:

<bb 3>:
  a_3 = b_1(D) + c_2(D);
  if (a_3 <= n_4(D))
       goto <bb 5>;
  else goto <bb 6>;

<bb 5>:
  goto <bb 3>;

<bb 6>:
<bb 4>:
  # a_6 = PHI <a_3(6)>
  return a_6;
```

loop.c.092t.pre

```
<bb 2>:
  pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
  a_3 = pretmp.2_7;
  if (a_3 <= n_4(D))
       goto <bb 5>;
  else goto <bb 6>;

<bb 5>:
  goto <bb 3>;

<bb 6>:
<bb 4>:
  # a_6 = PHI <a_3(6)>
  return a_6;
```


Copy Propagation after PRE

loop.c.092t.pre

```
<bb 2>:
  pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
  a_3 = pretmp.2_7;
  if ( a_3 <= n_4(D))
       goto <bb 5>;
  else goto <bb 6>;

<bb 5>:
  goto <bb 3>;

<bb 6>:
<bb 4>:
  # a_6 = PHI <a_3(6)>
  return a_6;
```

loop.c.097t.copyprop4

```
<bb 2>:
  pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
  a_3 = pretmp.2_7;
  if ( n_4(D) >= pretmp.2_7)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;

<bb 5>:
  # a_8 = PHI <pretmp.2_7(3)>
  a_6 = a_8;
  return a_8;
```


Dead Code Elimination

loop.c.097t.copyprop4

```
<bb 2>:
  pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
  a_3 = pretmp.2_7;
  if (n_4(D) >= pretmp.2_7)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;

<bb 5>:
  # a_8 = PHI <pretmp.2_7(3)>
  a_6 = a_8;
  return a_8;
```

loop.c.098t.dceloop1

```
<bb 2>:
  pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
  if (n_4(D) >= pretmp.2_7)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;

<bb 5>:
  # a_8 = PHI <pretmp.2_7(3)>
  return a_8;
```


Redundant ϕ Function Elimination and Copy Propagation

loop.c.098t.dceloop1

```
<bb 2>:
  pretmp.2_7 = b_1(D) + c_2(D);

<bb 3>:
  if (n_4(D) >= pretmp.2_7)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;

<bb 5>:
  # a_8 = PHI <pretmp.2_7(3)>
  return a_8;
```

loop.c.125t.phicprop2

```
<bb 2>:
  pretmp.2_7 = c_2(D) + b_1(D);
  if (n_4(D) >= pretmp.2_7)
    goto <bb 4>;
  else
    goto <bb 3>;

<bb 3>:
  return pretmp.2_7;

<bb 4>:
  goto <bb 4>;
```

Final Assembly Program

loop.c.125t.phicprop2

```
<bb 2>:
  pretmp.2_7 = c_2(D) + b_1(D);
  if (n_4(D) >= pretmp.2_7)
    goto <bb 4>;
  else
    goto <bb 3>;

<bb 3>:
  return pretmp.2_7;

<bb 4>:
  goto <bb 4>;
```

loop.s

```
     movl    8(%esp), %eax
     addl    4(%esp), %eax
     cmpl    %eax, 12(%esp)
     jge     .L2
     rep
     ret
.L2:
.L3:
     jmp    .L3
```

Why infinite loop?

Infinite Loop in Example Program 2

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

   do
   {
      a = b+c;
   }
   while (a <= n);

   return a;
}
```

The program does not terminate unless a > n

Part 3

Conclusions

Conclusions

- GCC performs many machine independent optimizations
- The dumps of optimizations are easy to follow, particularly at the GIMPLE level
- 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)