Compilation Principle
编 译 原 理

第22讲：代码优化(2)

张献伟

xianweiz.github.io

DCS290, 5/23/2023

Quiz Questions

- Q1: what is 3-phase compilation? Benefits?

  Front-end, IR, back-end. Decouple language from machine (i.e., independent). Easy to commonly optimize and to extend.

- Q2: TAC of x + y * z + 5.

  $t_1 = y * z; t_2 = x + t_1; t_3 = t_2 + 5;$

- Q3: is the code SSA? If not, convert it.

```
a = x * y;
if a > 5: a = z;
b = a + 2;
```

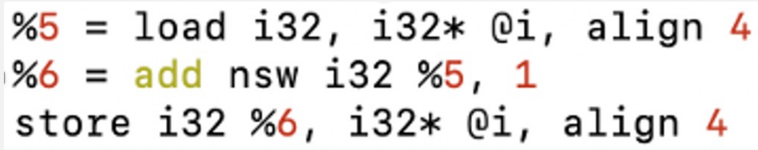
  No. x is assigned more than once.

  $a_1 = x * y;$ if $a_1 > 5: a_2 = z; b = PHI(a_1, a_2) + 2;$

- Q4: for the IR of S -> if (B) S_1 else S_2, where to place 'goto S.next'?

  S_1.code {goto S.next} else S_2.code: skip S_2 after executing S_1.

- Q5: explain the code.



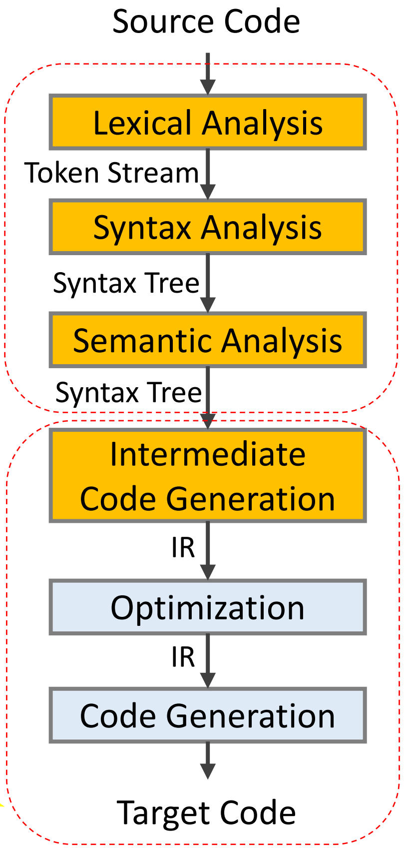
```
%5 = load i32, i32* @i, align 4
%6 = add nsw i32 %5, 1
store i32 %6, i32* @i, align 4
```

  i = i + 1:

  %5 = i; %6 = i + 1; i = %6

Types of Optimizations[分类]

- Compiler optimization is essentially a transformation[转换]
  - Delete / Add / Move / Modify something
- **Layout-related** transformations[布局相关]
  - Optimizes *where* in mem code and data is placed
  - Goal: maximize **spatial locality**[空间局部性]
    - Spatial locality: on an access, likelihood that nearby locations will also be accessed soon
    - Increases likelihood subsequent accesses will be faster
      - E.g. If access fetches cache line, later access can reuse
      - E.g. If access page faults, later access can reuse page
- **Code-related** transformations[代码相关]
  - Optimizes *what* code is generated
  - Goal: execute least number of most costly instructions

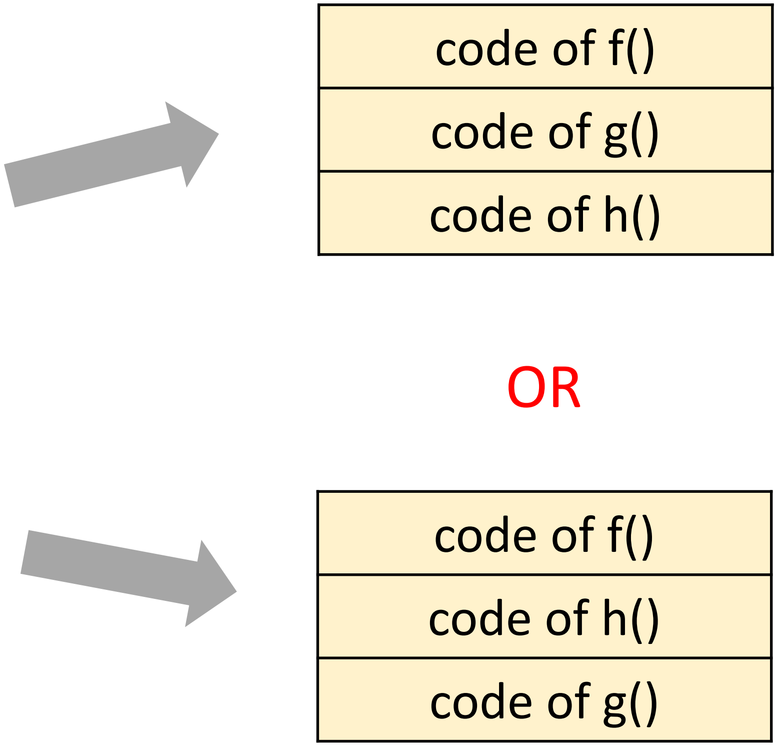


Focus

Layout-Related Opt.: Code

- Two ways to layout code for the below example

```
f() {
   …
   h();
   …
}
g() {
   …
}
h() {
  …
}
```



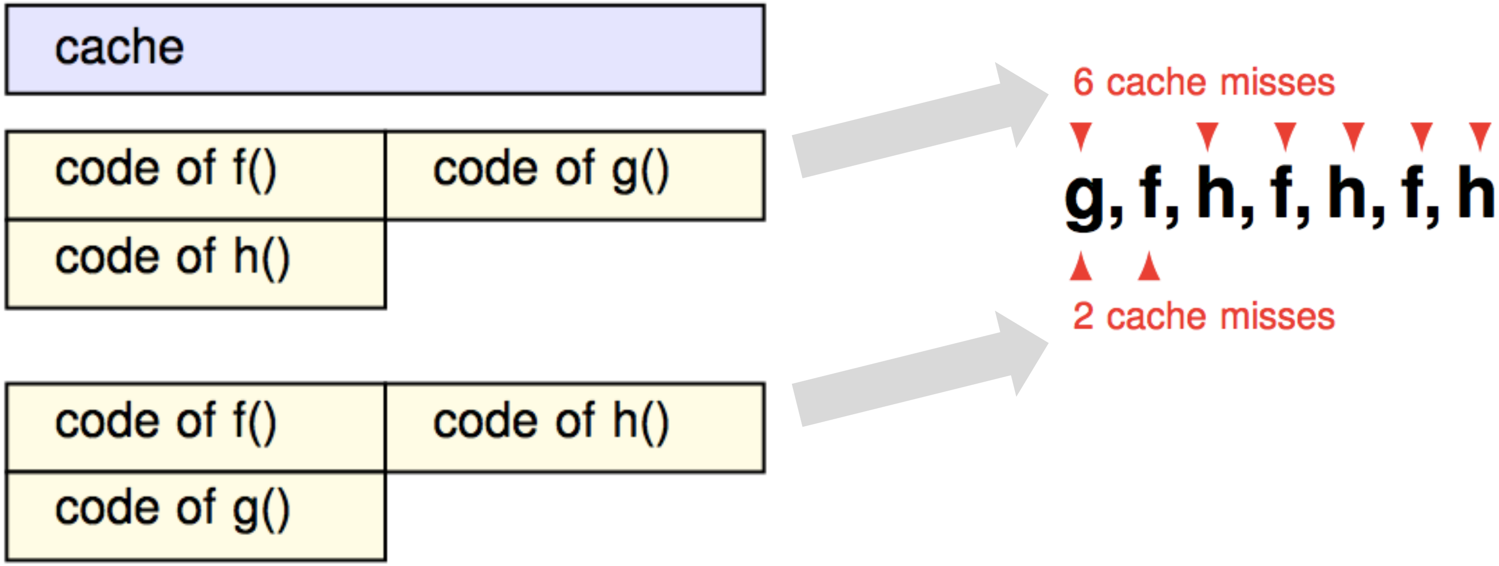
code of f()
code of g()
code of h()

OR

code of f()
code of h()
code of g()

Layout-Related Opt.: Code (cont.)

- Which code layout is better?
- Assume
  - data cache has one *N*-word line
  - the size of each function is *N/2*-word long
  - access sequence is “**g, f, h, f, h, f, h**”



cache

code of f()

code of g()

code of h()

code of f()

code of h()

code of g()

6 cache misses

g, f, h, f, h, f, h

2 cache misses

Layout-Related Opt.: Data

- Change the variable declaration order

```
struct S {
   int x1;
   int x2[200];
   int x3;
} obj[100];

for(...) {
   ... = obj[i].x1 + obj[i].x3;
}
```

```
struct S {
   int x1;
   int x3;
   int x2[200];
} obj[100];

for(...) {
   ... = obj[i].x1 + obj[i].x3;
}
```

- Improved spatial locality
  - Now x1 and x3 likely reside in same cache line
  - Access to x3 will always hit in the cache

Layout-Related Opt.: Data (cont.)

- Change AOS (array of structs) to SOA (struct of arrays)

```
struct S {
   int x;
   int y;
} points[100];

for(...) {
   ... = points[i].x * 2;
}
for(...) {
   ... = points[i].y * 2;
}
```

```
struct S {
   int x[100];
   int y[100];
} points;

for(...) {
   ... = points.x[i] * 2;
}
for(...) {
   ... = points.y[i] * 2;
}
```

- Improved spatial locality for accesses to ‘x’s and ‘y’s

Structure Peeling[结构分离]

```
struct S {
  int A;
  int B;
  int C;
};
```

A,C – Hot fields
B – Cold field

Peeled structures:

```
struct S.Hot {
  int A;
  int C;
};
```

```
struct S.Cold {
  int B;
};
```

https://llvm.org/devmtg/2014-10/Slides/Prashanth-DLO.pdf
https://llvm.org/devmtg/2021-02-28/slides/Prashantha-MLIR-LTO.pdf

Code-Related Optimizations

- Modifying code e.g. **strength reduction**[强度削减]

  A=2*a; ≡ A=a«1;

- Deleting code e.g. **dead code elimination**

  A=2; A=y; ≡ A=y;

- Moving code e.g. **code scheduling**

  A=x*y; B=A+1; C=y; ≡ A=x*y; C=y; B=A+1;

  (Now C=y; can execute while waiting for A=x*y;)

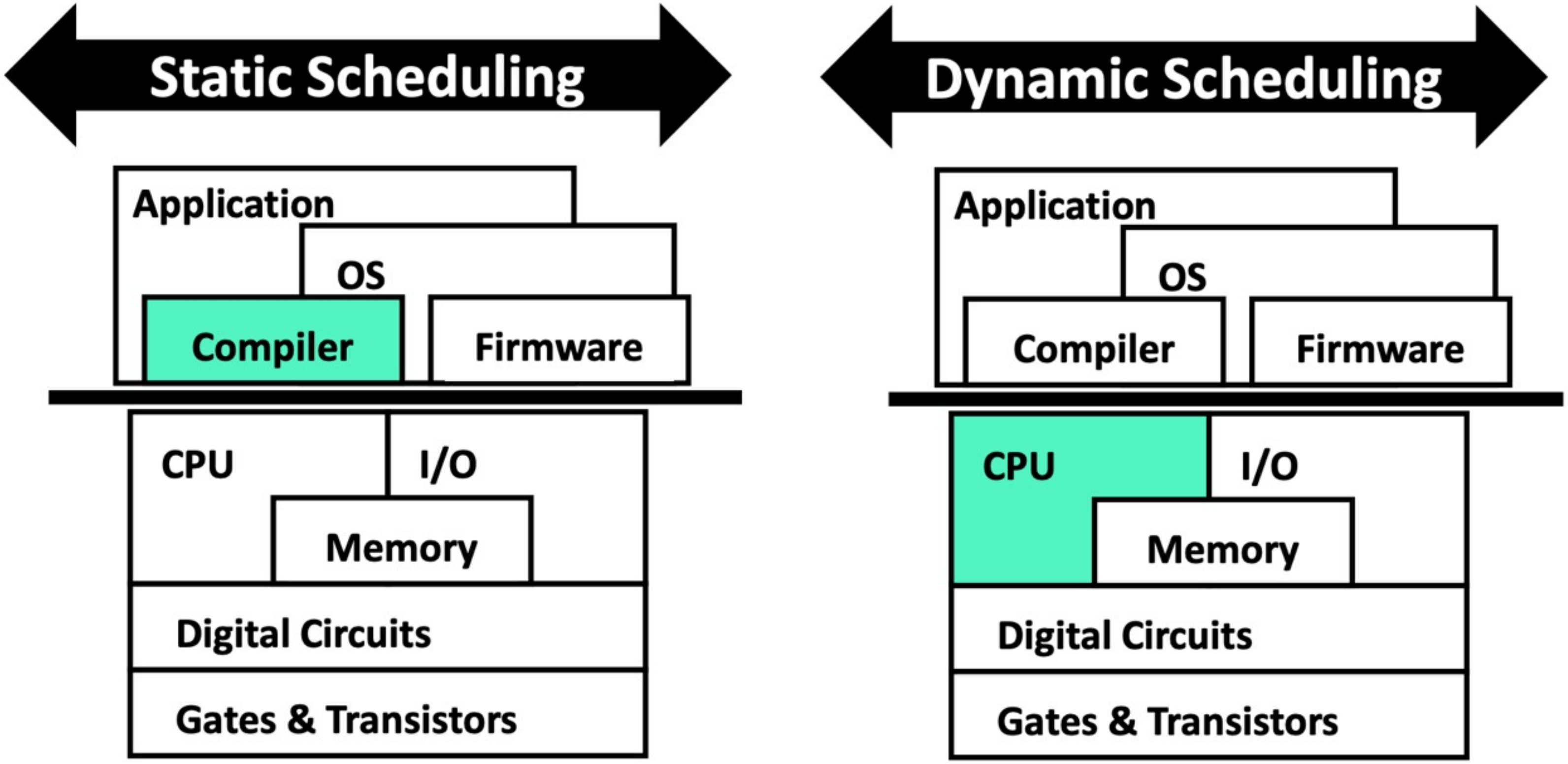
- Inserting code e.g. **data prefetching**[数据预取]

  ```
  while (p!=NULL)
  { process(p); p=p->next; }
  ≡
  while (p!=NULL)
  { prefetch(p->next); process(p); p=p->next; }
  ```

  (Now access to p->next is likely to hit in cache)

Detour: Instruction Scheduling[指令调度]

- Scheduling: act of finding independent instructions
  - Static: done at compile time by the compiler (sw)
  - Dynamic: done at runtime by the processor (hw)
    - Scoreboard, Tomasulo’s algorithm, Reorder Buffer (ROB)



https://acg.cis.upenn.edu/milom/mini-course-March-2013/lectures/08_scheduling.pdf
http://camelab.org/uploads/Main/lecture08-scoreboard.pdf

Detour: Compiler Tech. to Expose ILP

- Scheduling[调度]
  - To keep a pipeline full, parallelism among insts must be exploited by finding sequences of unrelated insts that can be overlapped in the pipeline[重叠]
  - To avoid a pipeline stall, the execution of a dependent inst must be separated from the source insts by a distance in clock cycles equal to the pipeline latency of that source inst[分隔]
- A compiler’s ability to perform the scheduling depends on
  - Amount of ILP in the program[程序特性]
  - Latencies of the functional units in the pipeline[硬件特性]
- Compiler can increase the amount of available of ILP by transforming loops[循环转换]

Detour: Loop Unrolling[循环展开]

- Simply replicates the loop body multiple times, adjusting the loop termination code[复制->调整]
  - Increases the number of insts relative to the branch and overhead insts[增加有效指令数]
  - Eliminates branches, thus allowing insts from different iterations to be scheduled together[消除分支, 共同调度]

```
Loop:  fld     f0, 0(x1)
       fadd.d  f4, f0, f2
       fsd     f4, 0(x1)
       fld     f6, -8(x1)
       fadd.d  f8, f6, f2
       fsd     f8, -8(x1)
       fld     f0, -16(x1)
       fadd.d  f12, f0, f2
       fsd     f12, -16(x1)
       fld     f14, -24(x1)
       fadd.d  f16, f14, f2
       fsd     f16, -24(x1)
       addi    x1, x1, -32
       bne     x1, x2, loop
```

```
Loop:  fld     f0, 0(x1)
       fld     f6, -8(x1)
       fld     f0, -16(x1)
       fld     f14, -24(x1)
       fadd.d  f4, f0, f2
       fadd.d  f8, f6, f2
       fadd.d  f12, f0, f2
       fadd.d  f16, f14, f2
       fsd     f4, 0(x1)
       fsd     f8, -8(x1)
       fsd     f12, -16(x1)
       fsd     f16, -24(x1)
       addi    x1, x1, -32
       bne     x1, x2, loop
```

A total of 14 clock cycles
(3.5 cycles per iter)

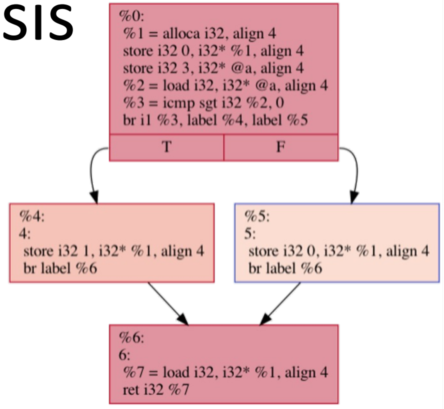
Detour: Unrolling Limitations[限制]

- The gains from loop unrolling are limited by
  - A decrease in the amount of overhead amortized with each unroll
    - Unrolled 4 times → 8 times: ½ cycle/iter → ¼ cycle/iter
  - Growth in code size caused by unrolling
    - May increase in the inst cache miss rate
    - May bring register pressure (more live values)
  - Compiler limitations
    - Sophisticated transformations increases the compiler complexity

```
Loop:  fld      f0, 0(x1)
       fld      f6, -8(x1)
       fld      f0, -16(x1)
       fld      f14, -24(x1)
       fadd.d   f4, f0, f2
       fadd.d   f8, f6, f2
       fadd.d   f12, f0, f2
       fadd.d   f16, f14, f2
       fsd      f4, 0(x1)
       fsd      f8, -8(x1)
       fsd      f12, -16(x1)
       fsd      f16, -24(x1)
       addi     x1, x1, -32
       bne      x1, x2, loop
```

Control-Flow Analysis[控制流分析]

- The compiling process has done lots of analysis
  - Lexical
  - Syntax
  - Semantic
  - IR



```
%0:
 %1 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 3, i32* @a, align 4
 %2 = load i32, i32* @a, align 4
 %3 = icmp sgt i32 %2, 0
 br i1 %3, label %4, label %5
```

```
%4:
4:
 store i32 1, i32* %1, align 4
 br label %6
```

```
%5:
5:
 store i32 0, i32* %1, align 4
 br label %6
```

```
%6:
6:
 %7 = load i32, i32* %1, align 4
 ret i32 %7
```

- But, it still doesn’t really know <u>how the program does what it does</u>
- **Control-flow analysis** helps compiler to figure out more info about how the program does its work
  - First construct a **control-flow graph** (CFG), which is a graph of the different possible paths program flow could take through a function
    - To build the graph, we first divide the code into basic blocks

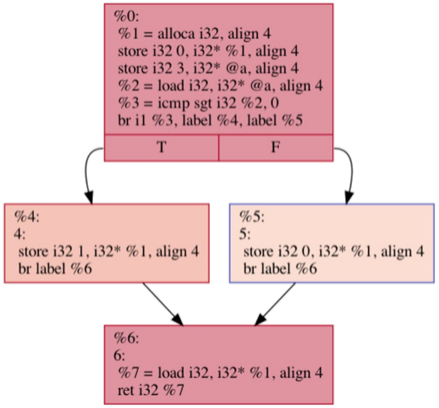
Basic Block[基本块]

- A **basic block** is a maximal sequence of instructions that
  - Except the first instruction, there are no other labels[只第一条入]
  - Except the last instruction, there are no jumps[只末一条出]
- Therefore, [进/出口唯一]
  - Can only jump into the beginning of a block
  - Can only jump out at the end of a block
- Are units of control flow that cannot be divided further
  - All instructions in basic block execute or none at all[all or nothing]

- Local optimizations are limited to scope of a basic block
- Global optimizations are across basic blocks

Control Flow Graph[控制流图]

- A **control flow graph** is a directed graph in which
  - Nodes are basic blocks
  - Edges represent flow of execution between basic blocks
    - ▫ Flow from end of one basic block to beginning of another
    - ▫ Flow can be result of a control flow divergence
    - ▫ Flow can be result of a control flow merge
  - Control statements introduce control flow edges
    - ▫ e.g. if-then-else, for-loop, while-loop, ...



```
%0:
 %1 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 3, i32* @a, align 4
 %2 = load i32, i32* @a, align 4
 %3 = icmp sgt i32 %2, 0
 br i1 %3, label %4, label %5
```

```
%4:
4:
 store i32 1, i32* %1, align 4
 br label %6
```

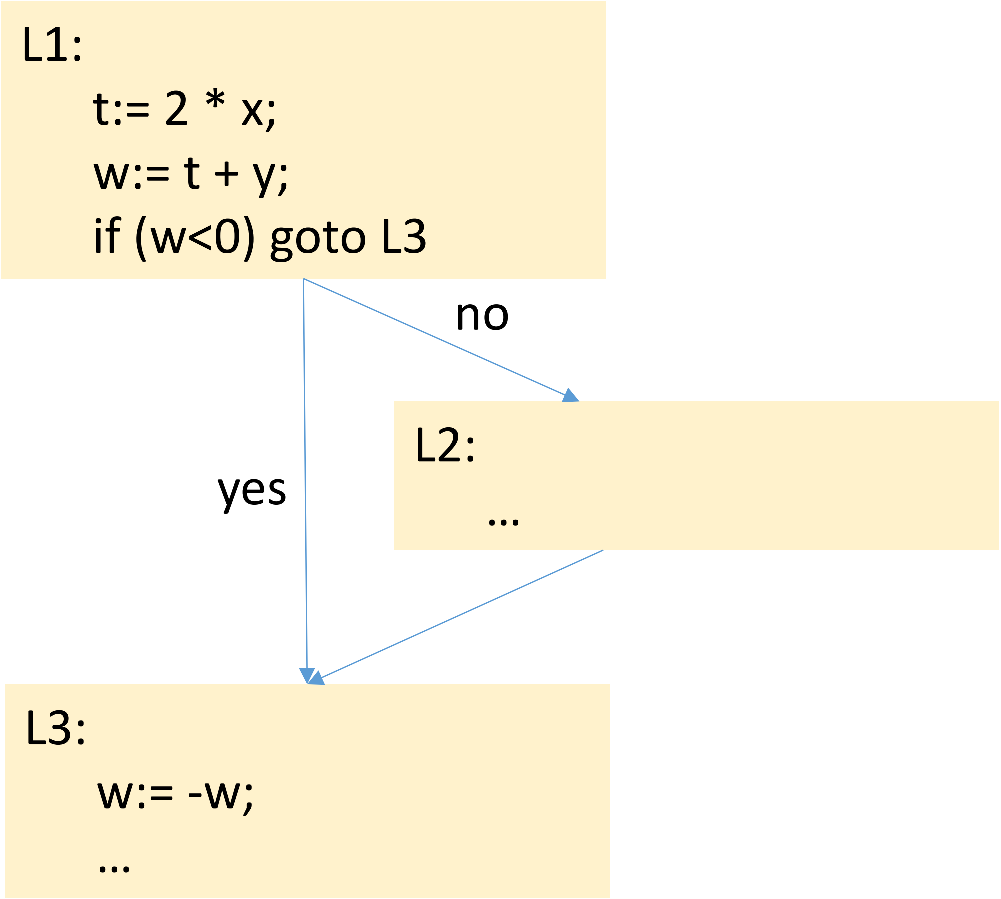
```
%5:
5:
 store i32 0, i32* %1, align 4
 br label %6
```

```
%6:
6:
 %7 = load i32, i32* %1, align 4
 ret i32 %7
```

- CFG is widely used to represent a function
- CFG is widely used for program analysis, especially for global analysis/optimization

Example

```
L1:
     t:= 2 * x;
     w:= t + y;
     if (w<0) goto L3
L2:
     …
L3:
     w:= -w
      …
```



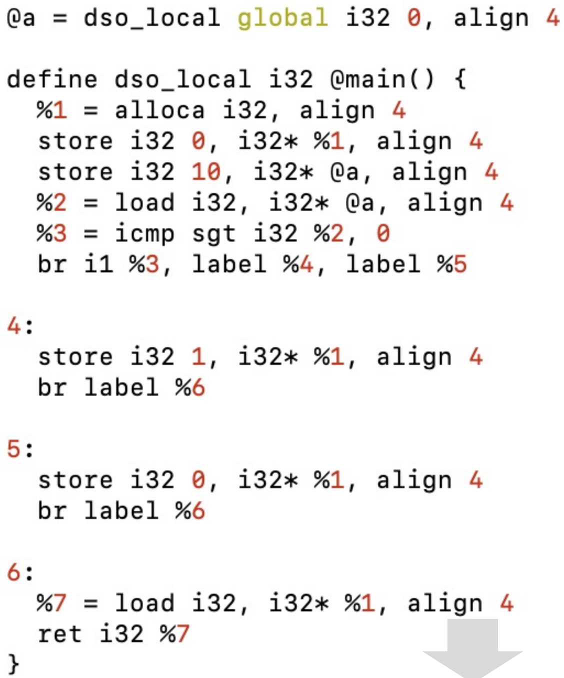
```
L1:
     t:= 2 * x;
     w:= t + y;
     if (w<0) goto L3
```

```
L2:
     …
```

```
L3:
     w:= -w;
     …
```

LLVM CFG

- $clang -emit-llvm -S ../tester/functional/027_if2.sysu.c



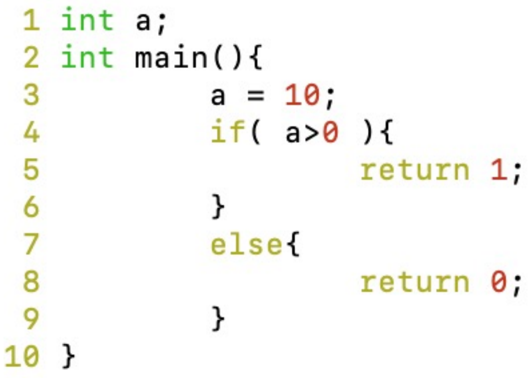
```
@a = dso_local global i32 0, align 4

define dso_local i32 @main() {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 10, i32* @a, align 4
  %2 = load i32, i32* @a, align 4
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %4, label %5

4:
  store i32 1, i32* %1, align 4
  br label %6

5:
  store i32 0, i32* %1, align 4
  br label %6

6:
  %7 = load i32, i32* %1, align 4
  ret i32 %7
}
```



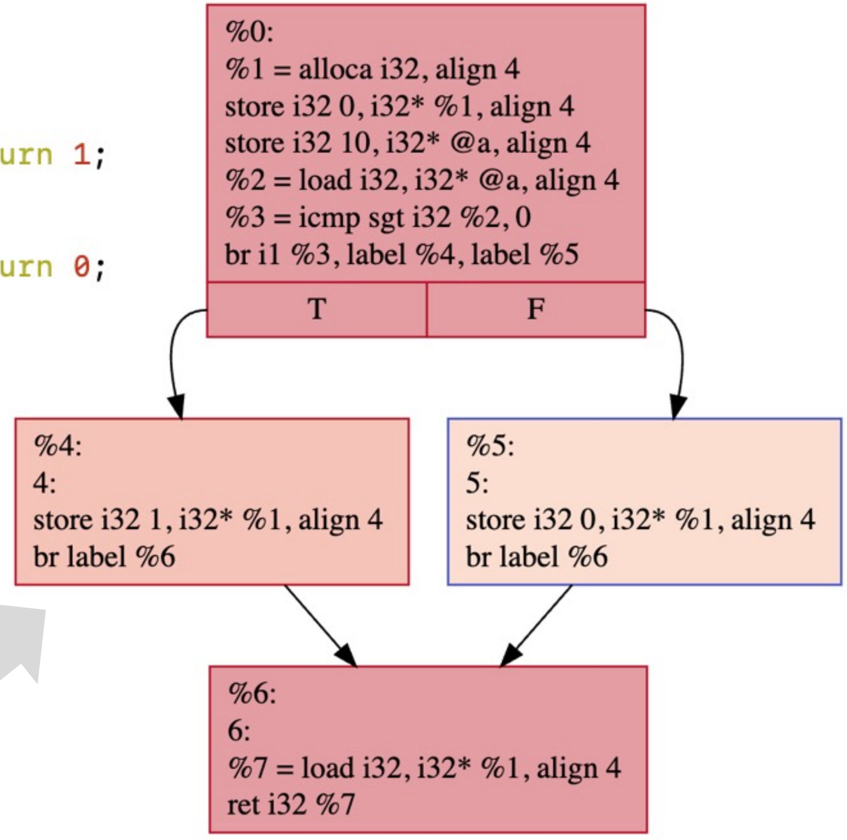
```
int a;
int main(){
        a = 10;
        if( a>0 ){
                return 1;
        }
        else{
                return 0;
        }
}
```

$opt -dot-cfg 027_if2.sysu.ll [→ .main.dot]

```
digraph "CFG for 'main' function" {
        label="CFG for 'main' function";

        Node0x2a784a90 [shape=record,color="#b70d28ff", style=filled, fillcolor="#b
70d2870",label="{%0:\l  %1 = alloca i32, align 4\l  store i32 0, i32* %1, align 4\l
  store i32 10, i32* @a, align 4\l  %2 = load i32, i32* @a, align 4\l  %3 = icmp sg
t i32 %2, 0\l  br i1 %3, label %4, label %5\l|{<s0>T|<s1>F}}"];
        Node0x2a784a90:s0 -> Node0x2a784c70;
        Node0x2a784a90:s1 -> Node0x2a784cc0;
        Node0x2a784c70 [shape=record,color="#b70d28ff", style=filled, fillcolor="#e
8765c70",label="{%4:\l4:                                              \l  store i
32 1, i32* %1, align 4\l  br label %6\l}"];
        Node0x2a784c70 -> Node0x2a784e50;
        Node0x2a784cc0 [shape=record,color="#3d50c3ff", style=filled, fillcolor="#f
7b39670",label="{%5:\l5:                                              \l  store i
32 0, i32* %1, align 4\l  br label %6\l}"];
        Node0x2a784cc0 -> Node0x2a784e50;
        Node0x2a784e50 [shape=record,color="#b70d28ff", style=filled, fillcolor="#b
70d2870",label="{%6:\l6:                                              \l  %7 = lo
ad i32, i32* %1, align 4\l  ret i32 %7\l}"];
}
```

http://viz-js.com/



CFG for 'main' function

Construct CFG

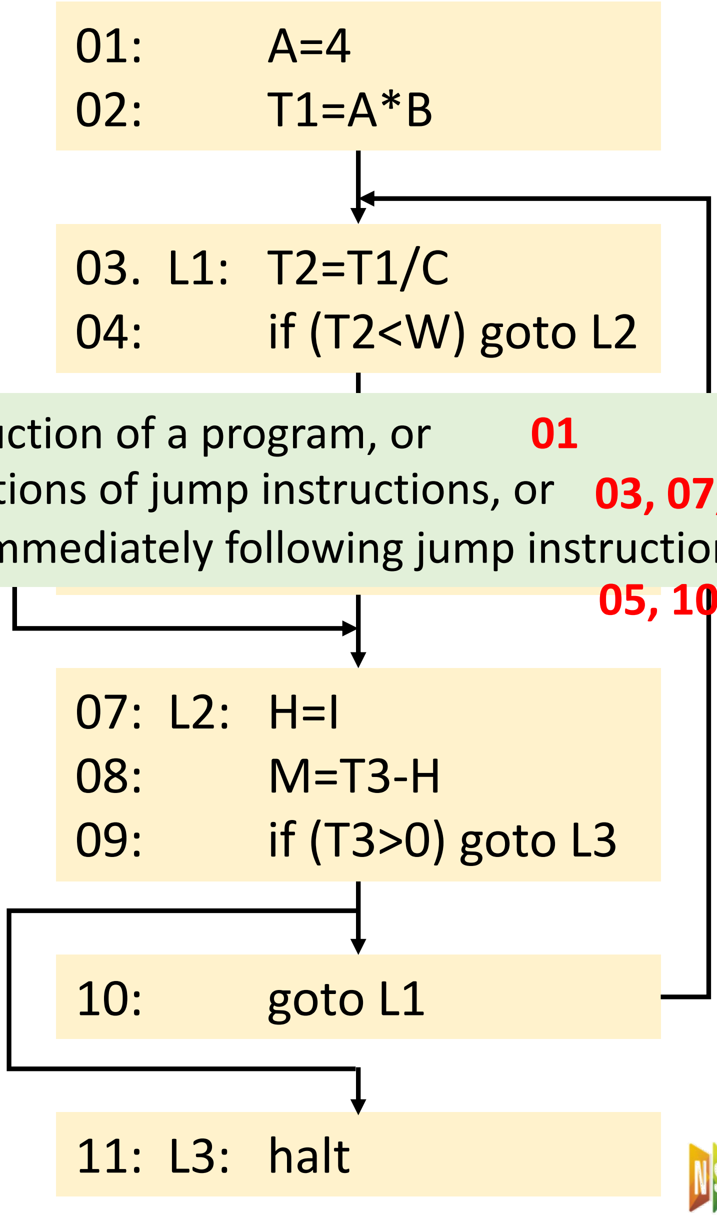
- Step 1: partition code into basic blocks[分解为基本块]
  - Identify **leader** instructions that are
    - ☐ the first instruction of a program, or[首条指令]
    - ☐ target instructions of jump instructions, or[跳转目标]
    - ☐ instructions immediately following jump instructions[紧跟跳转]
  - A basic block consists of a leader instruction and subsequent instructions before the next leader
- Step 2: add an edge between basic blocks B1 and B2 if[连接基本块]
  - B2 follows B1, and B1 may “fall through” to B2[相邻]
    - ☐ B1 ends with a conditional jump to another basic block[若条件假，到达B2]
    - ☐ B1 ends with a non-jump instruction (B2 is a target of a jump)[无跳转，B1顺序执行到达B2]
    - ☐ Note: if B1 ends in an unconditional jump, cannot fall through[B1无条件跳转，会绕开B2]
  - B2 doesn’t follow B1, but B1 ends with a jump to B2[不相邻，但B2是B1的跳转目标]

Example

- Partition code into basic blocks
  - Identify leader instructions
- Add edges between basic blocks

```
01:        A=4
02:        T1=A*B
03.  L1:   T2=T1/C
04:        if (T2<W) goto L2
05:        M=T1*K
06:        T3=M+1
07:  L2:   H=I
08:        M=T3-H
09:        if (T3>0) goto L3
10:        goto L1
11:  L3:   halt
```

- ○ the first instruction of a program, or **01**
- ○ target instructions of jump instructions, or **03, 07, 11**
- ○ instructions immediately following jump instructions **05, 10, 11**



01: A=4
02: T1=A*B

03. L1: T2=T1/C
04: if (T2<W) goto L2

07: L2: H=I
08: M=T3-H
09: if (T3>0) goto L3

10: goto L1

11: L3: halt

Local and Global Optimizations

- Local optimizations[局部优化]
  - Optimizations performed exclusively within a basic block
  - Typically the easiest, never consider any control flow info
    - All instructions in scope executed exactly once
  - Examples:
    - constant folding[常量折叠]
    - common subexpression elimination[删除公共子表达式]

- Global optimizations[全局优化]
  - Optimizations performed across basic blocks
    - Scope can contain if / while / for statements
    - Some insts may not execute, or even execute multiple times
  - Note: global here doesn’t mean across the entire program
    - We usually optimize one function at a time

Local Optimization: Examples

- Common subexpression elimination[公共子表达式删除]
  - Two operations are common if they produce the same result
    - It is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it[避免重复计算]
- Dead code elimination[无用代码删除]
  - If an instruction's result is never used, the instruction is considered "dead" and can be removed from the instruction stream[结果从不使用]

$y = x + z;$
$y = x * x + (x/3)$
$z = x * x + y;$

$y = x + z;$
$t_1 = x * x$
$t_2 = x / 3$
$y = t_1 + t_2$
$t_3 = x * x$
$z = t_3 + y;$

~~$y = x + z;$~~
$t_1 = x * x$
$t_2 = x / 3$
$y = t_1 + t_2$
~~$t_3 = x * x$~~
$z =$ **t_1** $+ y;$

DAG of Basic Blocks

- Many important techniques for local optimization begin by transforming a BB into a DAG (directed acyclic graph)[无环有向图]

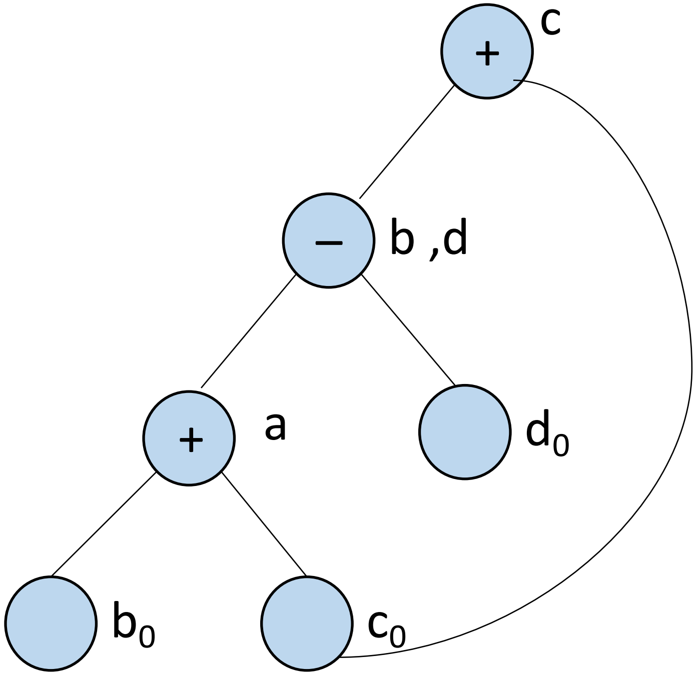
- To construct a DAG for a BB as follows
  - Create a node for each of the initial values of the variables appearing in the BB[为变量初始值创建节点，叶子]
  - Create a node *N* associated with each statement *s* within the block[为声明语句创建节点，中间]
    - The children of *N* are those nodes corresponding to statements that are the last definitions, prior to *s*, of the operands used by *s*
    - Label *N* by the operator applied at *s*[用运算符标注节点]
  - Certain nodes are designated output nodes[某些为输出节点]
    - These are the nodes whose variables are *live on exit* from the block (i.e., their values may be used later, in another block of the flow graph)

Example: DAG

- (3) c = b + c
  - *b* refers to the node labelled '-'
    - Most recent definition of *b*

- (4) d = a - d
  - Operator and children are the same as the 2nd statement
    - Reuse the node

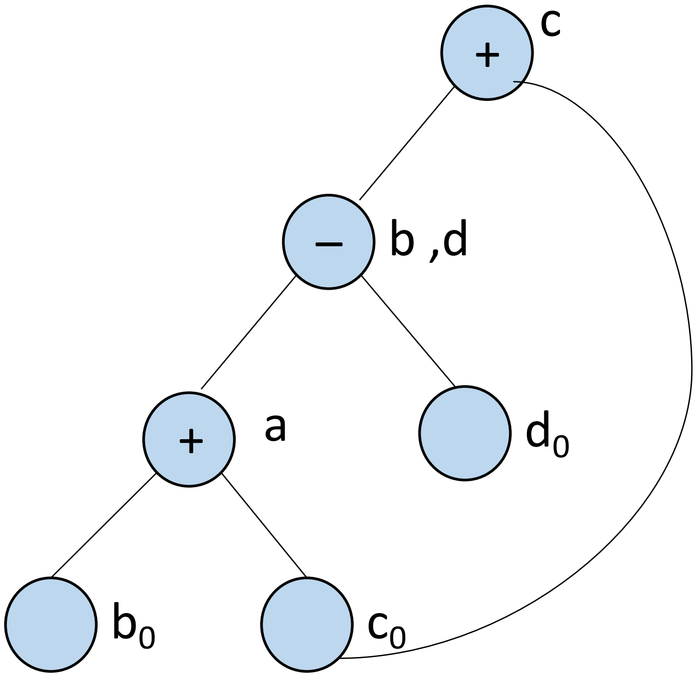
(1) a = b + c
(2) b = a - d
(3) c = b + c
(4) d = a - d



Local Opt.: Elimination

- If *c* is not live on exit from the block
  - No need to keep *c = b + c*
- If both *b* and *d* are live
  - Remove either (2) or (4) : **common subexpr elimination**
  - Add a 4th statement to copy one to the other
- If only *a* is live on exit
  - Then remove nodes from the DAG correspond to dead code
    - c -> b,d -> d_0
  - This is actually **dead code elimination**

(1) a = b + c
(2) b = a - d
(3) c = b + c
(4) d = a - d



Local Opt.: Elimination (cont.)

- When finding common subexprs, we really are finding exprs that are <u>guaranteed</u> to compute the same value, no matter how that value is computed[过于严苛]
  - Thus miss the fact that (1) and (4) are the same
    - ▫ $b + c = (b - d) + (c + d) = b_0 + c_0$
- **Solution**: algebraic identities[代数恒等式]

(1) a = b + c
(2) b = b - d
(3) c = c + d
(4) e = b + c