Intermediate Code Generation

Ronghui Gu

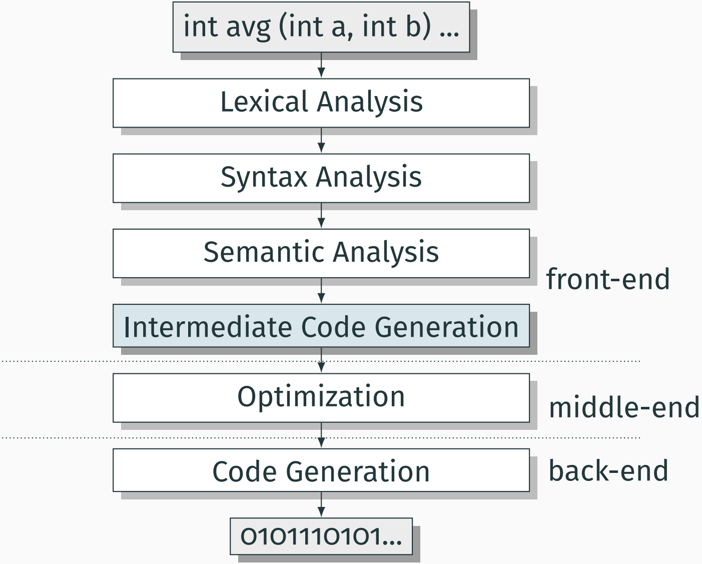
Spring 2020

Columbia University

* Course website: https://www.cs.columbia.edu/ rgu/courses/4115/spring2019

** These slides are borrowed from Prof. Edwards.

Intermediate Code Generation



Intermediate Code Generation

Intermediate Representation (IR):

- An abstract machine language
- Not specific to any particular machine
- Independent of source language

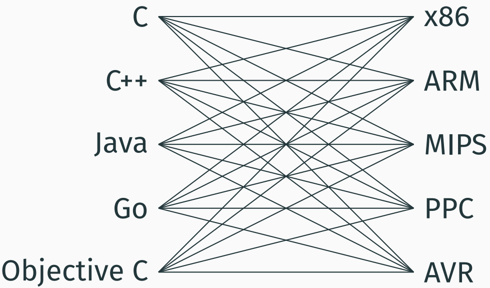
IR code generation is not necessary:

- Semantic analysis phase can generate assembly code directly.
- Hinders portability and modularity.

Intermediate Representation

Suppose we wish to build compilers for n source languages and m target machines.

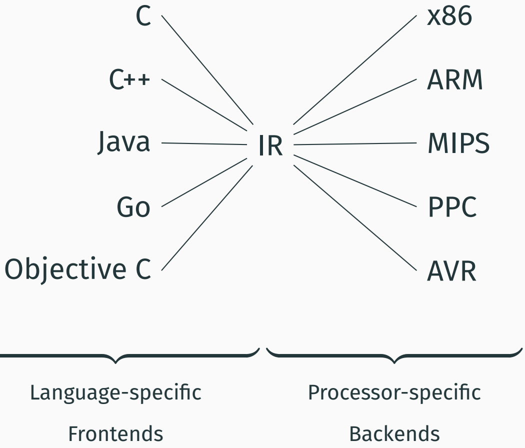
Case 1: no IR. Need $n \times m$ compilers.



Intermediate Representation

Suppose we wish to build compilers for n source languages and m target machines.

Case 2: IR present. Need just n front-ends and m back ends.



IR properties

- Must be convenient for semantic analysis phase to produce.
- Must be convenient to translate into real assembly code for all desired target machines.

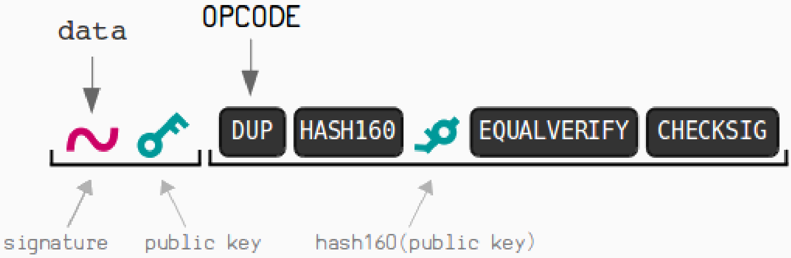
Intermediate Representations/Formats

Stack-Based IR: Java Bytecode

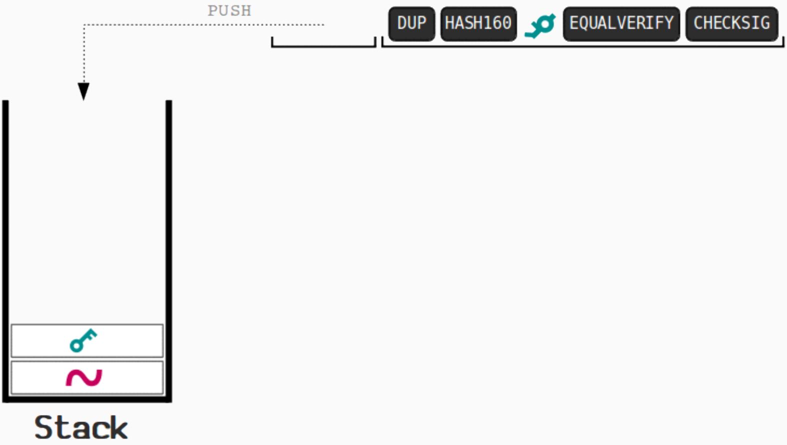
```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

```
Method int gcd(int, int)
  0 goto 19

  3 iload_1        // Push a
  4 iload_2        // Push b
  5 if_icmple 15 // if a <= b goto 15

  8 iload_1        // Push a
  9 iload_2        // Push b
 10 isub           // a - b
 11 istore_1       // Store new a
 12 goto 19

 15 iload_2        // Push b
 16 iload_1        // Push a
 17 isub           // b - a
 18 istore_2       // Store new b

 19 iload_1        // Push a
 20 iload_2        // Push b
 21 if_icmpne 3  // if a != b goto 3

 24 iload_1        // Push a
 25 ireturn        // Return a
```

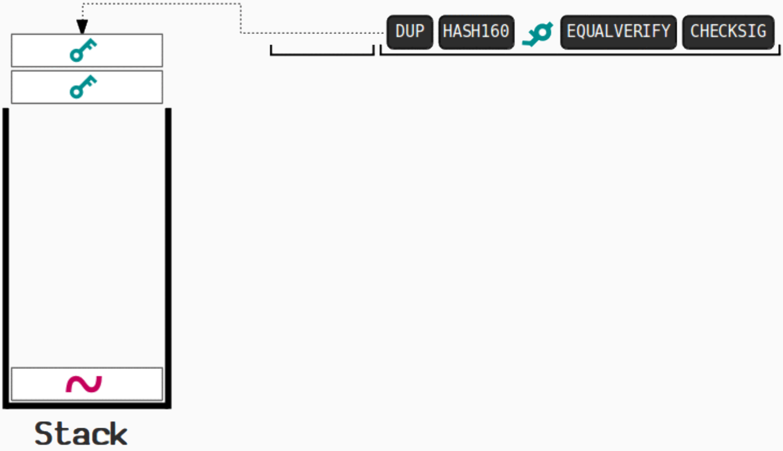
Stack-Based IR: Bitcoin Script



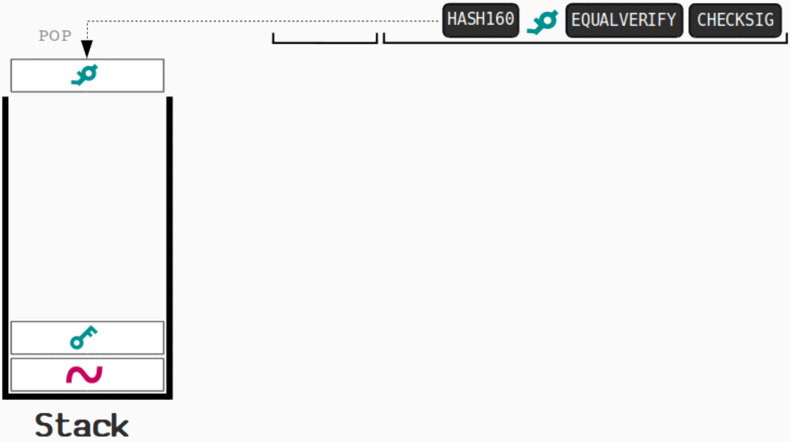
Stack-Based IR: Bitcoin Script



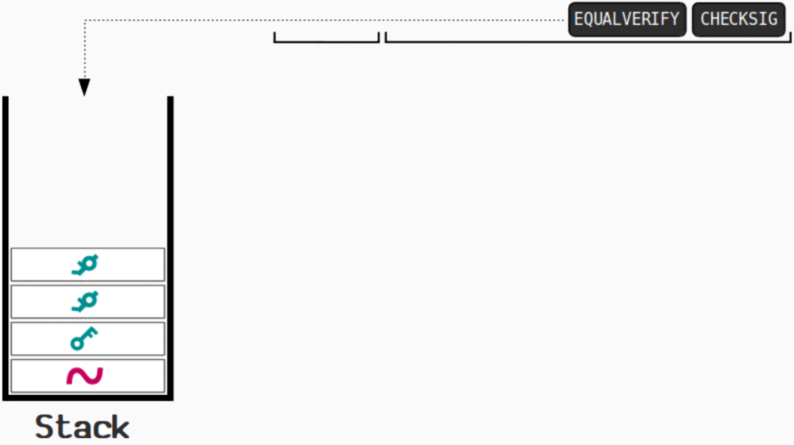
Stack-Based IR: Bitcoin Script



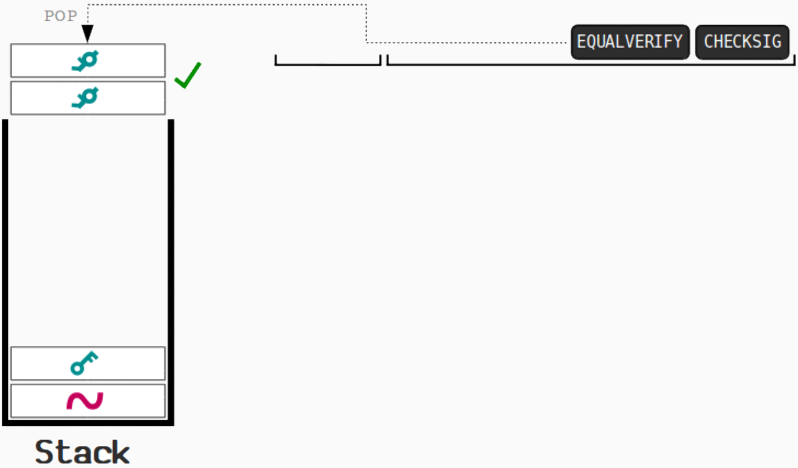
Stack-Based IR: Bitcoin Script



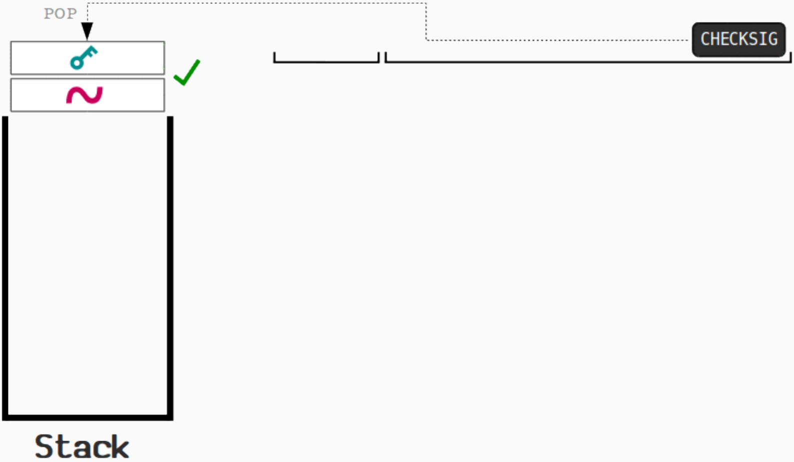
Stack-Based IR: Bitcoin Script



Stack-Based IR: Bitcoin Script



Stack-Based IR: Bitcoin Script



Stack-Based IRs

Advantages:

- Trivial translation of expressions
- Trivial interpreters
- No problems with exhausting registers
- Often compact

Disadvantages:

- Semantic gap between stack operations and modern register machines
- Hard to see what communicates with what
- Difficult representation for optimization

Register-Based IR: Mach SUIF

```c
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}
```

```
gcd:
gcd._gcdTmp0:
  sne    $vr1.s32 <- gcd.a,gcd.b
  seq    $vr0.s32 <- $vr1.s32,0
  btrue $vr0.s32,gcd._gcdTmp1  // if !(a != b) goto Tmp1

  sl     $vr3.s32 <- gcd.b,gcd.a
  seq    $vr2.s32 <- $vr3.s32,0
  btrue $vr2.s32,gcd._gcdTmp4  // if !(a < b) goto Tmp4

  mrk    2, 4    // Line number 4
  sub    $vr4.s32 <- gcd.a,gcd.b
  mov    gcd._gcdTmp2 <- $vr4.s32
  mov    gcd.a <- gcd._gcdTmp2  // a = a - b
  jmp    gcd._gcdTmp5
gcd._gcdTmp4:
  mrk    2, 6
  sub    $vr5.s32 <- gcd.b,gcd.a
  mov    gcd._gcdTmp3 <- $vr5.s32
  mov    gcd.b <- gcd._gcdTmp3  // b = b - a
gcd._gcdTmp5:
  jmp    gcd._gcdTmp0

gcd._gcdTmp1:
  mrk    2, 8
  ret    gcd.a  // Return a
```

Register-Based IRs

Most common type of IR

Advantages:

- Better representation for register machines
- Dataflow is usually clear

Disadvantages:

- Slightly harder to synthesize from code
- Less compact
- More complicated to interpret

Three-Address Code & Static Single Assignment

Most register-based IRs use **three-address code**: Arithmetic instructions have (up to) three operands: two sources and one destination.

SSA Form: each variable in an IR is assigned exactly once

C code:

```
int gcd(int a, int b)
{
  while (a != b)
    if (a < b)
      b -= a;
    else
      a -= b;
  return a;
}
```

Three-Address:

```
WHILE:  t = a != b
        bz DONE, t
        t = a < b
        bz ELSE, t
        b = b - a
        jmp LOOP
ELSE:   a = a - b
LOOP:   jmp WHILE
DONE:   ret a
```

SSA:

```
WHILE:  t1 = a1 != b1
        bz DONE, t1
        t2 = a1 < b1
        bz ELSE, t2
        b1 = b1 - a1
        jmp LOOP
ELSE:   a1 = a - b1
LOOP:   jmp WHILE
DONE:   ret a1
```

Three-Address Code

What is an "Address" in Three-Address Code?

- Name: (from the source program) e.g., x, y, z
- Constant: (with explicit primitive type) e.g., 1, 2, 'a'
- Compiler-generated temporary: ("register") e.g., t1, t2, t3

Instructions of Three-Address Code

- x = y op z: where op is a binary operation
- x = op y: where op is a unary operation
- x = y: copy operation
- jmp L: unconditional jump to label L
- bz L, x: jump to L if x is zero
- bnz L, x: jump to L if x is not zero
- param x, call L, y, return z: function calls

Three-Address Code (TAC) Generation

Goal: take statements (AST) and produce a sequence of TAC.

Example:

```
a := b + c * d;
```

TAC:

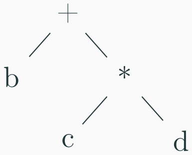
```
t1 = c * d
t2 = b + t1
a = t1
```

Translate expressions and statements

Translating Expressions

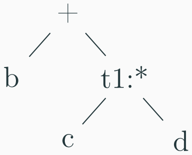
Example

b + c * d



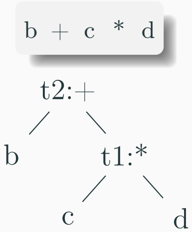
Example

b + c * d



t1 = c * d

Example



t1 = c * d

t2 = b + t1

Algorithm: Syntax-Directed Translation (SDT)

For each expression E, we'll synthesize two attributes:

- E.addr: the name of the variable (often a temporary variable)
- E.code: the IR instructions generated from E

SDT: each semantic rule corresponds to actions computing two attributes with the following auxiliary functions:

- Call NewTemp to create a new temporary variable
- Call Gen: to print a new three-address instruction

  Gen(t, "=", x, op, y) $\Rightarrow$ "t = x op y"

Syntax-Directed Translation (SDT)

CFG rule: $E_0 \rightarrow$ **id**

Actions:

E_0.addr := **id**

E_0.code := “” empty string

We do not consider scopes here.

Example: E_0 = ID(“a”)

E_0.addr := “a”

E_0.code := “” empty string

Syntax-Directed Translation (SDT)

CFG rule: $E_0 \rightarrow E_1 + E_2$

Actions:

E_0.addr := NewTemp()

E_0.code :=E_1.code || E_2.code ||

Gen(E_0.addr, “=”, “add”, E_1.addr, “,”, E_2.addr)

Example: a + b

E_0 = PLUS (E_1, E_2) E_1 = ID(“a”) E_2 = ID(“b”)

E_1.addr := “a” E_1.code := “”

E_2.addr := “b” E_2.code := “”

E_0.addr := “t1”

E_0.code := “t1 = add a, b”

Syntax-Directed Translation (SDT)

Example: b + c * d

E_0 = PLUS (E_1, E_2) E_1 = ID(“b”)

E_2 = MUL (ID(“c”), ID(“d”))

E_0.code :=E_1.code || E_2.code ||

Gen(E_0.addr, “=”, E_1.addr, “+”, E_2.addr)

Syntax-Directed Translation (SDT)

Example: b + c * d

E_0 = PLUS (E_1, E_2) E_1 = ID("b")

E_2 = MUL (ID("c"), ID("d"))

E_0.code := E_1.code || E_2.code ||

Gen(E_0.addr, "=", E_1.addr, "+", E_2.addr)

Syntax-Directed Translation (SDT)

Example: b + c * d

E_0 = PLUS (E_1, E_2) E_1 = ID("b")

E_2 = MUL (ID("c"), ID("d"))

E_0.code := "" || E_2.code ||

Gen(E_0.addr, "=", E_1.addr, "+", E_2.addr)

E_1.addr = "b"

Syntax-Directed Translation (SDT)

Example: b + c * d

E_0 = PLUS (E_1, E_2) E_1 = ID("b")

E_2 = MUL (ID("c"), ID("d"))

E_0.code := "" || "t1 = c * d"||

Gen(E_0.addr, "=", E_1.addr, "+", E_2.addr)

E_1.addr = "b" E_2.addr = "t1"

Syntax-Directed Translation (SDT)

Example: b + c * d

E_0 = PLUS (E_1, E_2) E_1 = ID("b")

E_2 = MUL (ID("c"), ID("d"))

E_0.code := "" || "t1 = c * d"||

Gen(NewTemp(), "=", E_1.addr, "+", E_2.addr)

E_1.addr = "b" E_2.addr = "t1"

Syntax-Directed Translation (SDT)

Example: b + c * d

E_0 = PLUS (E_1, E_2) E_1 = ID("b")

E_2 = MUL (ID("c"), ID("d"))

E_0.code := "" || "t1 = c * d"||

Gen("t2", "=", E_1.addr, "'+", E_2.addr)

E_1.addr = "b" E_2.addr = "t1"

Syntax-Directed Translation (SDT)

Example: b + c * d

E_0 = PLUS (E_1, E_2) E_1 = ID("b")

E_2 = MUL (ID("c"), ID("d"))

E_0.code := "" || "t1 = c * d"||

Gen("t2", "=", "b", "+", "t1")

Syntax-Directed Translation (SDT)

Example: b + c * d

E_0 = PLUS (E_1, E_2) E_1 = ID("b")

E_2 = MUL (ID("c"), ID("d"))

E_0.code := "" || "t1 = c * d"||

"t2 = b + t1"

Translating Statements

Assignment

CFG rule: $S \to \textbf{id} := E$

Actions:

S.code :=E.code || Gen(**id**, “=”, E.addr)

Example: a := b + c

S = ASG (ID(“a”), E) E =PLUS(ID(“b”), ID(“c”))

E.code := “t1 = b + c” E.addr := “t1’

S.code := “t1 = b + c” || “a = t1”

IF Statement

AST: IF(E, S)

Generated IR:

```
    E.code
    bz Label_End, E.addr
    S.code
Label_End:
```

Example: if (a > b) { c = a - b }

```
    t1 = a > b
    bz Label_End, t1
    c = a - b
Label_End:
```

IF-ELSE Statement

AST: IFELSE(E, S_1, S_2)

Generated IR:

```
        E.code
        bz Label_Else, E.addr
        S_1.code
        jmp Label_End
    Label_Else:
        S_2.code
    Label_End:
```

IF-ELSE Statement

Example: if (a > b) { c = a - b } { c = b - a }

```
    t1 = a > b
    bz Label_Else, t1
    c = a - b
    jmp Label_End
Label_Else:
    c = b - a
Label_End:
```

Loop

AST: WHILE(E, S)

Generated IR:

```
Label_While:
    E.code
    bz Label_End, E.addr
    S.code
    jmp Label_While
Label_End:
```

Function Calls

$f\ (E_1, \cdots, E_n)$

Generated IR:

E_n.code
E_{n-1}.code
$\ldots$
E_1.code
param E_n.addr
$\ldots$
param E_1.addr
call f, n

Function Calls

$f\,(E_1, \cdots, E_n)$

Generated IR:

E_n.code
E_{n-1}.code
$\ldots$
E_1.code
param E_n.addr   how to pass parameters?
$\ldots$
param E_1.addr
call f, n

And One More Thing...

```
int x;     where is this x stored? what is x.addr?

int main () {
    x = 4 ;
    int y;     where is this y stored? what is y.addr?
...
}
```

Basic Blocks

A **Basic Block** is a sequence of IR instructions with two properties:

1. The first instruction is the only entry point (no other branches in; can only start at the beginning)
2. Only the last instruction may affect control (no other branches out)

∴ If any instruction in a basic block runs, they all do

Typically "arithmetic and memory instructions, then branch"

```
ENTER:  t2 = t1 + 1
        t3 = t2 < 10
        bz NEXT, t3
```

Basic Blocks and Control-Flow Graphs

```
WHILE:  t1 = a1 != b1      ◀
        bz DONE, t1
        t2 = a1 < b1       ◀
        bz ELSE, t2
        b1 = b1 – a1       ◀
        jmp LOOP
ELSE:   a1 = a1 – b1       ◀
LOOP:   jmp WHILE          ◀
DONE:   ret a1             ◀
```

- Leaders: branch targets & after conditional branch

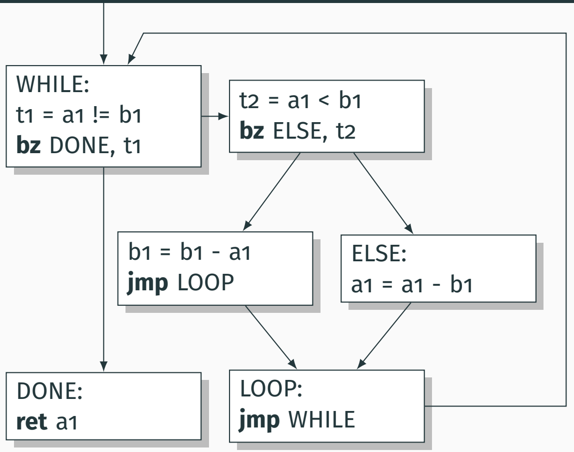
Basic Blocks and Control-Flow Graphs

```
WHILE:  t1 = a1 != b1      ◀
        bz DONE, t1
--------------------------
        t2 = a1 < b1       ◀
        bz ELSE, t2
--------------------------
        b1 = b1 – a1       ◀
        jmp LOOP
--------------------------
ELSE:   a1 = a1 – b1       ◀
--------------------------
LOOP:   jmp WHILE          ◀
--------------------------
DONE:   ret a1             ◀
```

- Leaders: branch targets & after conditional branch
- Basic blocks: start at a leader; end before next

Basic Blocks and Control-Flow Graphs

```
WHILE:  t1 = a1 != b1
        bz DONE, t1
-----------------------
        t2 = a1 < b1
        bz ELSE, t2
-----------------------
        b1 = b1 − a1
        jmp LOOP
-----------------------
ELSE:   a1 = a1 − b1
-----------------------
LOOP:   jmp WHILE
-----------------------
DONE:   ret a1
```



- Leaders: branch targets & after conditional branch
- Basic blocks: start at a leader; end before next
- Basic Blocks are nodes of the Control-Flow Graph

The LLVM IR

Three-address code instructions; Static single-assignment; Explicit control-flow graph; Local names start with %; Types throughout; User-defined functions

```
int add(int x, int y)
{
  return x + y;
}
```

```
define i32 @add(i32 %x, i32 %y) {
entry:
  %x1 = alloca i32
  store i32 %x, i32* %x1
  %y2 = alloca i32
  store i32 %y, i32* %y2
  %x3 = load i32* %x1
  %y4 = load i32* %y2
  %tmp = add i32 %x3, %y4
  ret i32 %tmp
}
```

The LLVM IR

i32: 32-bit signed integer type

alloca: Allocate space on the stack; return a pointer

store: Write a value to an address

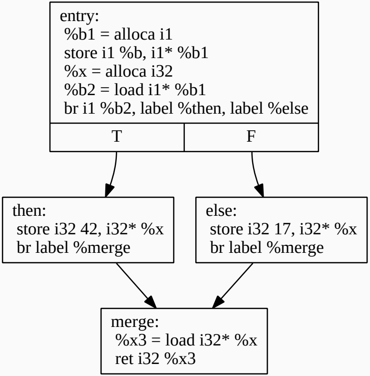
load: Read a value from an address

add: Add two values to produce a third

ret: Return a value to the caller

Basic Blocks

```
int cond(bool b) {
  int x;
  if (b) x = 42;
  else   x = 17;
  return x;
}
```



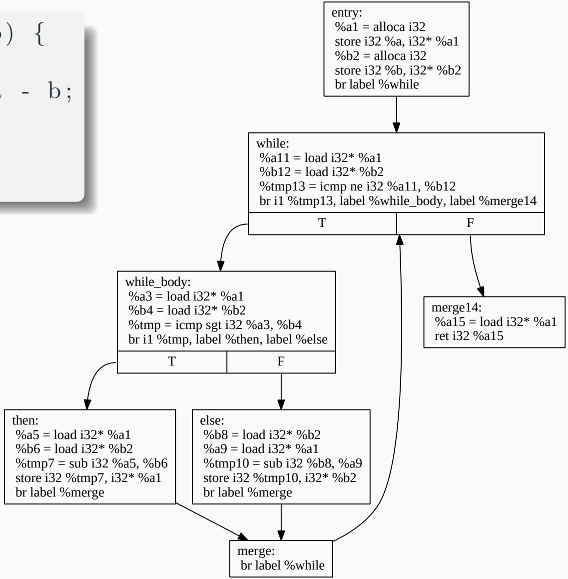
CFG for 'cond' function

```
define i32 @cond(i1 %b) {
entry:
  %b1 = alloca i1
  store i1 %b, i1* %b1
  %x = alloca i32
  %b2 = load i1* %b1
  br i1 %b2, label %then, label %else

merge:          ; preds = %else, %then
  %x3 = load i32* %x
  ret i32 %x3

then:           ; preds = %entry
  store i32 42, i32* %x
  br label %merge

else:           ; preds = %entry
  store i32 17, i32* %x
  br label %merge
}
```

```c
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}
```

```llvm
define i32 @gcd(i32 %a, i32 %b) {
entry:
  %a1 = alloca i32
  store i32 %a, i32* %a1
  %b2 = alloca i32
  store i32 %b, i32* %b2
  br label %while
while:                        ; preds = %merge, %entry
  %a11 = load i32* %a1
  %b12 = load i32* %b2
  %tmp13 = icmp ne i32 %a11, %b12
  br i1 %tmp13, label %while_body, label %merge14
while_body:                   ; preds = %while
  %a3 = load i32* %a1
  %b4 = load i32* %b2
  %tmp = icmp sgt i32 %a3, %b4
  br i1 %tmp, label %then, label %else
merge:                        ; preds = %else, %then
  br label %while
then:                         ; preds = %while_body
  %a5 = load i32* %a1
  %b6 = load i32* %b2
  %tmp7 = sub i32 %a5, %b6
  store i32 %tmp7, i32* %a1
  br label %merge
else:                         ; preds = %while_body
  %b8 = load i32* %b2
  %a9 = load i32* %a1
  %tmp10 = sub i32 %b8, %a9
  store i32 %tmp10, i32* %b2
  br label %merge
merge14:                      ; preds = %while
  %a15 = load i32* %a1
  ret i32 %a15
}
```

```
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}
```



CFG for 'gcd' function