11. Compilers

6.004x Computation Structures
Part 2 – Computer Architecture

Programming Languages

32-bit (4-byte) ADD instruction:

opcode	rc	ra	rb	(unused)
1 0 0 0 0 0	0 0 1 0 0	0 0 0 1 0	0 0 0 1 1	0 0 0 0 0 0 0 0 0 0 0

Means, to BETA, `Reg[4] ← Reg[2] + Reg[3]`

We'd rather write

`ADD(R2, R3, R4)` ***(Assembly)***

or better yet

`a = b + c;` ***(High-Level Language)***

High-Level Languages

Most algorithms are naturally expressed at a high level. Consider the following algorithm:

```
/* Compute greatest common divisor
 * using Euclid’s method
 */
int gcd(int a, int b) {
  int x = a;
  int y = b;
  while (x != y) {
    if (x > y) {
      x = x - y;
    } else {
      y = y - x;
    }
  }
  return x;
}
```

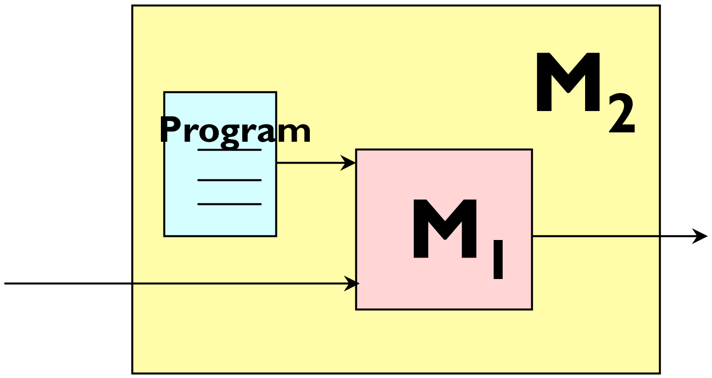
- 6.004 uses C, a common systems programming language. Modern popular alternatives include C++, Java, Python, and many others

- Advantages over assembly
  - Productivity (concise, readable, maintainable)
  - Correctness (type checking, etc)
  - Portability (run same program on different hardware)

- Disadvantages over assembly?
  - Efficiency?

Implementations: Interpretation vs compilation

Interpretation

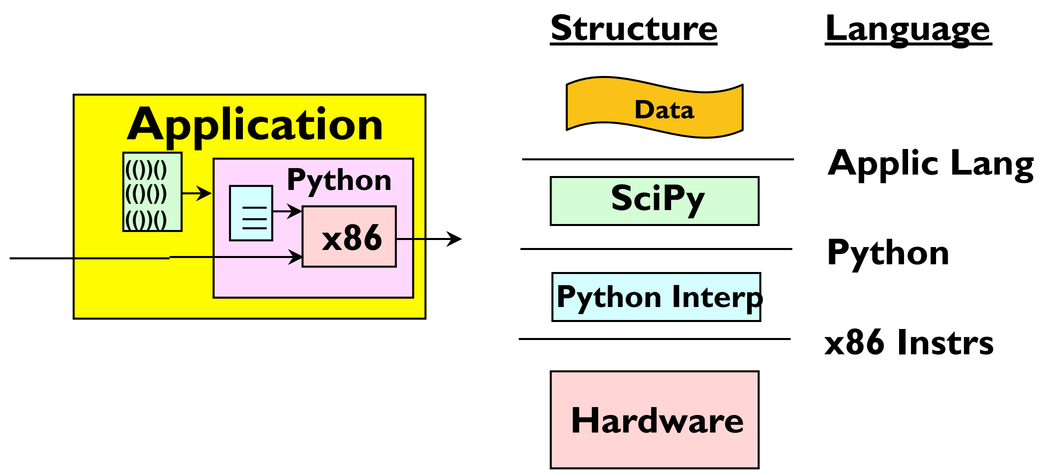


Model of *Interpretation:*

- Start with some hard-to-program machine, say M_1
- Write a single program for M_1 that mimics the behavior of some easier machine, M_2
- Result: a "virtual" M_2

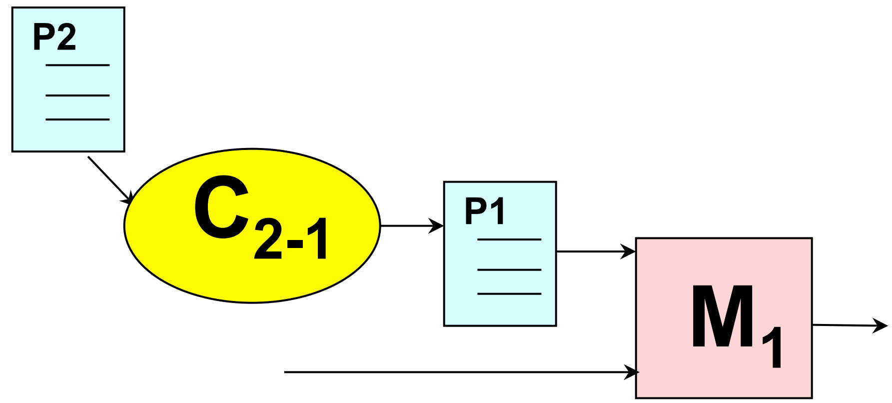
Layers of interpretation:

- Often we use several layers of interpretation to achieve desired behavior, e.g.:
- x86 CPU, running
  - Python, running
    - SciPy application, performing
      - Numerical analysys



Compilation

Model of *Compilation:*



- Given some hard-to-program machine, say M_1…
- Find some easier-to-program language L_2 (perhaps for a more complicated machine, M_2); write programs in that language
- Build a translator (compiler) that translates programs from M_2's language to M_1's language. May run on M_1, M_2, or some other machine.

Interpretation and compilation: two ways to execute high-level languages

- Both allow changes in the source program
- Both afford programming applications in platform (e.g., processor) independent languages
- Both are widely used in modern computer systems!

Interpretation vs Compilation

- Characteristic differences:

	Interpretation	Compilation
How it treats input "x+2"	Computes x+2	Generates a program that computes x+2
When it happens	During execution	Before execution
What it complicates/slows	Program execution	Program development
Decisions made at	Run time	Compile time

- Major choice we'll see repeatedly: do it at compile time or at run time?
  - Which is faster?
  - Which is more general?

Compilers

- Bare minimum for a functional compiler:

- Good compilers:
  - Produce meaningful errors on incorrect programs
    - Even better: meaningful warnings
  - Produce fast, optimized code

- This lecture:
  - Simple techniques to compile a C programs into assembly
  - Overview of how modern compilers work

A Simple Compilation Strategy

- Programs are sequences of statements, so repeatedly call `compile_statement(statement)`:
  - Unconditional: `expr;`
  - Compound: `{ statement_1; statement_2; ... }`
  - Conditional: `if (expr) statement_1; else statement_2;`
  - Iteration: `while (expr) statement;`

- Also need `compile_expr(expr)` to generate code to compute value of expr into a register
  - Constants: `1234`
  - Variables: `a`, `b[expr]`
  - Assignment: `a = expr`
  - Operations: `expr_1 + expr_2`, …
  - Procedure calls: `proc(expr,...)`

compile_expr(expr) ⇒ Rx

- Constants: 1234 ⇒ Rx
  - CMOVE(1234,Rx)
  - LD(c1,Rx)

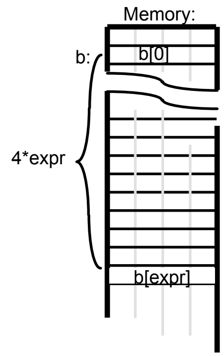
    …

    c1: LONG(123456)
- Variables: a ⇒ Rx
  - LD(a,Rx)

    . . .

    a: LONG(0)
- Assignment: a=expr ⇒ Rx
  - compile_expr(*expr*)⇒Rx

    ST(Rx,a)



- Variables: b[*expr*] ⇒ Rx
  - compile_expr(*expr*)⇒Rx

    MULC(Rx,*bsize*,Rx)

    LD(Rx,b,Rx)

    …

    // reserve array space

    b: . = . + *bsize***blen*
- Operations:

  $expr_1 + expr_2$ ⇒ Rx
  - compile_expr($expr_1$)⇒Rx

    compile_expr($expr_2$)⇒Ry

    ADD(Rx,Ry,Rx)

Compiling Expressions

C code:

```
int x, y;
y = (x-3)*(y+123456)
```

Beta assembly code:

```
x:  LONG(0)
y:  LONG(0)
c1: LONG(123456)
...
    LD(x, r1)
    CMOVE(3, r2)
    SUB(r1, r2, r1)      ⇒SUBC(r1,3,r1)
    LD(y, r2)
    LD(c1, r3)
    ADD(r2, r3, r2)
    MUL(r2, r1, r1)
    ST(r1, y)
```

```
compile_expr(y = (x-3)*(y+123456))
   compile_expr((x-3)*(y+123456))
      compile_expr(x-3)
         compile_expr(x)
            LD(x,r1)
         compile_expr(3)
            CMOVE(3,r2)
         SUB(r1,r2,r1)
      compile_expr(y+123456)
         compile_expr(y)
            LD(y,r2)
         compile_expr(123456)
            LD(c1,r3)
         ADD(r2,r3,r2)
      MUL(r1,r2,r1)
   ST(r1,y)
```

compile_statement

- Unconditional: *expr;*

  Beta assembly:

  compile_expr(*expr*)

- Compound: { *$statement_1$; $statement_2$; ...* }

  Beta assembly:

  compile_statement(*$statement_1$*)
  compile_statement(*$statement_2$*)
  …

compile_statement: Conditional

C code:

```
if (expr)
 statement;
```

Beta assembly:

compile_expr(*expr*)⇒Rx

```
BF(rx, Lendif)
```

compile_statement(*statement*)

```
Lendif:
```

C code:

```
if (expr)
  statement1;
else
  statement2;
```

Beta assembly:

compile_expr(*expr*)⇒Rx

```
BF(rx, Lelse)
```

compile_statement($statement_1$)

```
BR(Lendif)
Lelse:
```

compile_statement($statement_2$)

```
Lendif:
```

compile_statement: Iteration

C code:

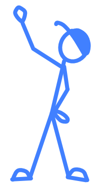
```
while (expr)
   statement;
```

Beta assembly:

```
Lwhile:
   compile_expr(expr)⇒Rx
   BF(rx, Lendwhile)
   compile_statement(statement)
   BR(Lwhile)
Lendwhile:
```

Better Beta assembly:

```
   BR(Ltest)
Lwhile:
   compile_statement(statement)
Ltest:
   compile_expr(expr)⇒Rx
   BT(rx, Lwhile)
```



Saves an instruction each iteration

C code:

```
for (init; test; increment)
  statement;
```

Example:

```
for (i=0; i < 10; i = i + 1)
  sum = sum + b[i];
```

is equivalent to:

```
init;
while (test) {
    statement;
    increment;
}
```

Putting It All Together: Factorial

`int n = 20;` `int r = 0;`	`n: LONG(20)` `r: LONG(0)` `start:`
`r = 1;`	`CMOVE(1, r0)` `ST(r0, r)`
`while (n > 0) {`	`BR(test)` `loop:`
`r = r*n;`	`LD(r, r3)` `LD(n,r1)` `MUL(r1, r3, r3)` `ST(r3, r)`
`n = n-1;`	`LD(n,r1)` `SUBC(r1, 1, r1)` `ST(r1, n)`
`}`	`test:` `LD(n, r1)` `CMPLT(r31, r1, r2)` `BT(r2, loop)` `done:`

Easy translation

Slow code
(10 instructions
in the loop)

Optimization: keep values in regs

```
int n = 20,
int r;

r = 1;
```

```
n: LONG(20)
r: LONG(0)

start:
   CMOVE(1, r0)
   ST(r0, r)
   LD(n,r1)     | keep n in r1
   LD(r,r3)     | keep r in r3
```

```
while (n > 0)
 {
    r = r*n;
    n = n-1;
 }
```

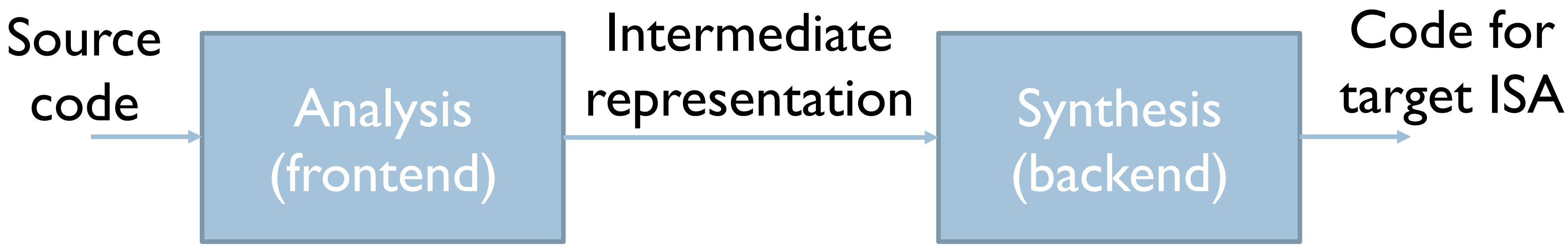
```
   BR(test)
loop:
   MUL(r1, r3, r3)
   SUBC(r1, 1, r1)
test:
   CMPLT(r31, r1, r2)
   BT(r2, loop)

done:
   ST(r1,n)     | save final n
   ST(r3,r)     | save final r
```

Optimization:
Keep n, r in registers
⇒ move LDs/STs
out of loop!

4 instructions in the loop

Anatomy of a Modern Compiler



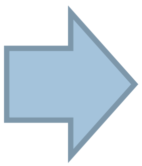
- Read source program
- Break it up into basic elements
- Check correctness, report errors
- Translate to generic intermediate representation (IR)

- Optimize IR
- Translate IR to ASM
- Optimize ASM

Frontend Stages

- Lexical analysis (scanning): Source → List of tokens

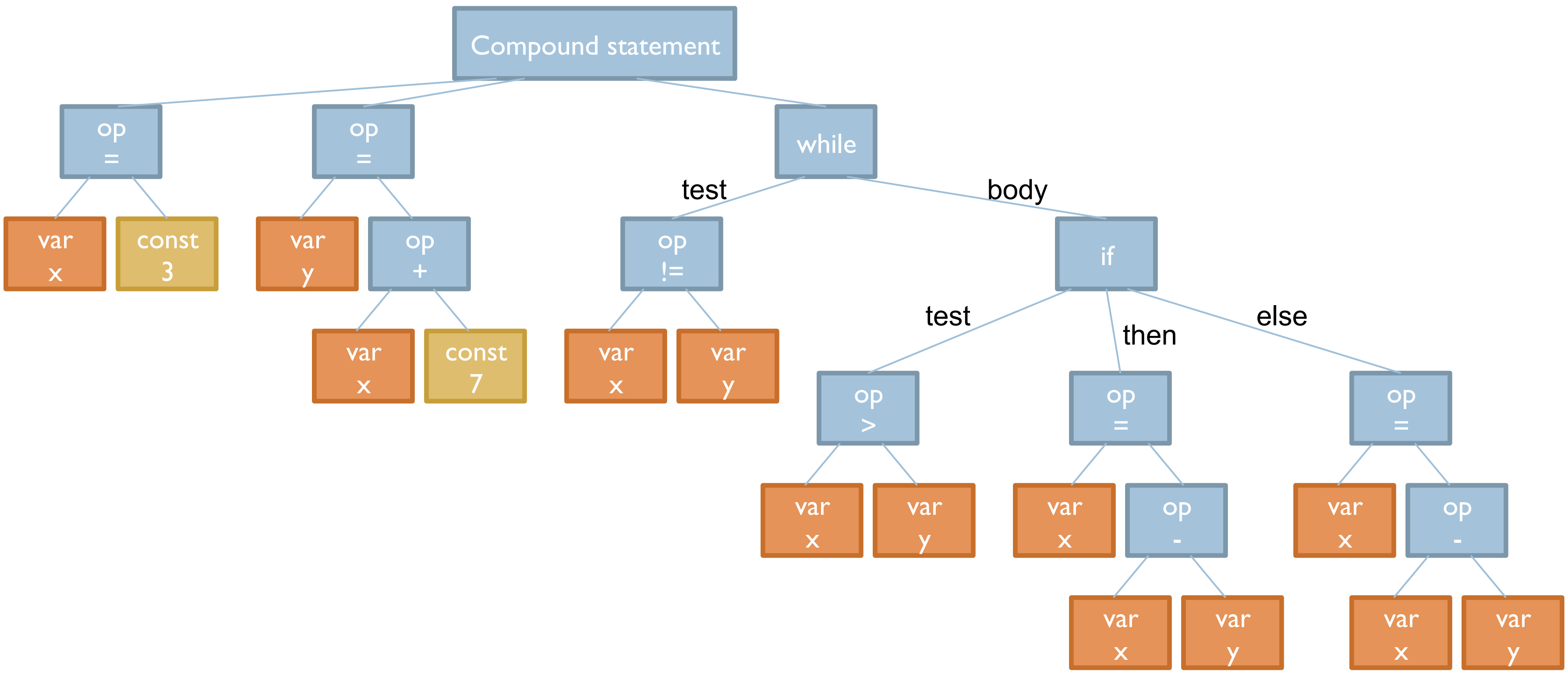
```
int x = 3;
int y = x + 7;
while (x != y) {
  if (x > y) {
    x = x - y;
  } else {
    y = y - x;
  }
}
```



```
("int", KEYWORD)
("x", IDENTIFIER)
("=", OPERATOR)
("3", INT_CONSTANT)
(";", SPECIAL_SYMBOL)
("int", KEYWORD)
("y", IDENTIFIER)
("=", OPERATOR)
("x", IDENTIFIER)
("+", OPERATOR)
("7", INT_CONSTANT)
(";", SPECIAL_SYMBOL)
("while", KEYWORD)
("(", SPECIAL_SYMBOL)
...
```

Frontend Stages

- Lexical analysis (scanning): Source → Tokens
- Syntactic analysis (parsing): Tokens → Syntax tree



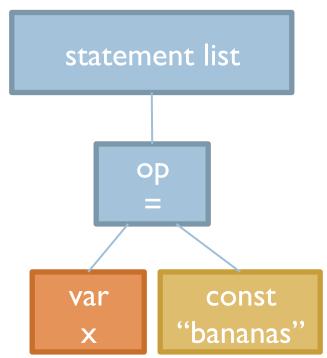
Frontend Stages

- Lexical analysis (scanning): Source → Tokens
- Syntactic analysis (parsing): Tokens → Syntax tree
- Semantic analysis (mainly, type checking)

Consider:

```
int x = “bananas”;
```

Syntax OK
Semantically (meaning) WRONG

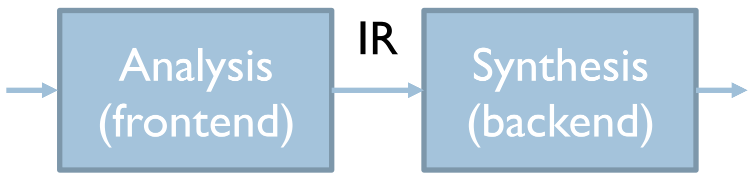


Var	Type
x	int

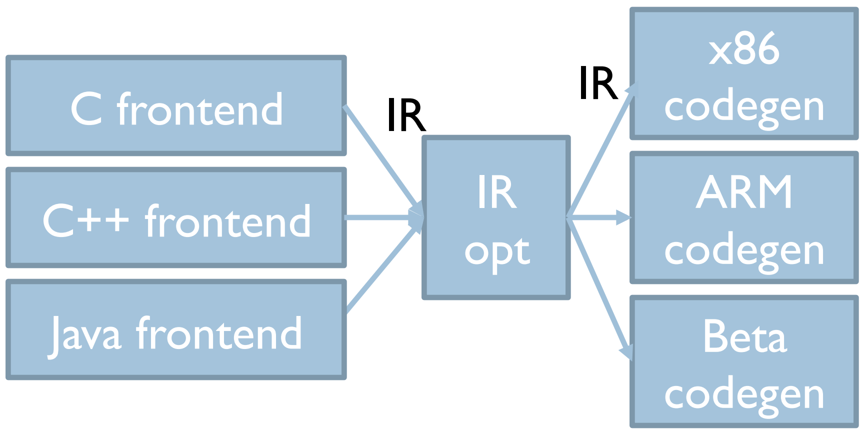
```
Line 1: error, invalid conversion from string constant to int
```

Intermediate Representation (IR)

- Internal compiler language that is:
  - Language-independent
  - Machine-independent
  - Easy to optimize



- Why yet another language?
  - Assembly does not have enough info to optimize it well
  - Enables modularity and reuse



Common IR: Control Flow Graph

- Assignments:

```
x = a op b
```

Variable (x); Variable or constant (a, b); op: +, -, *, …

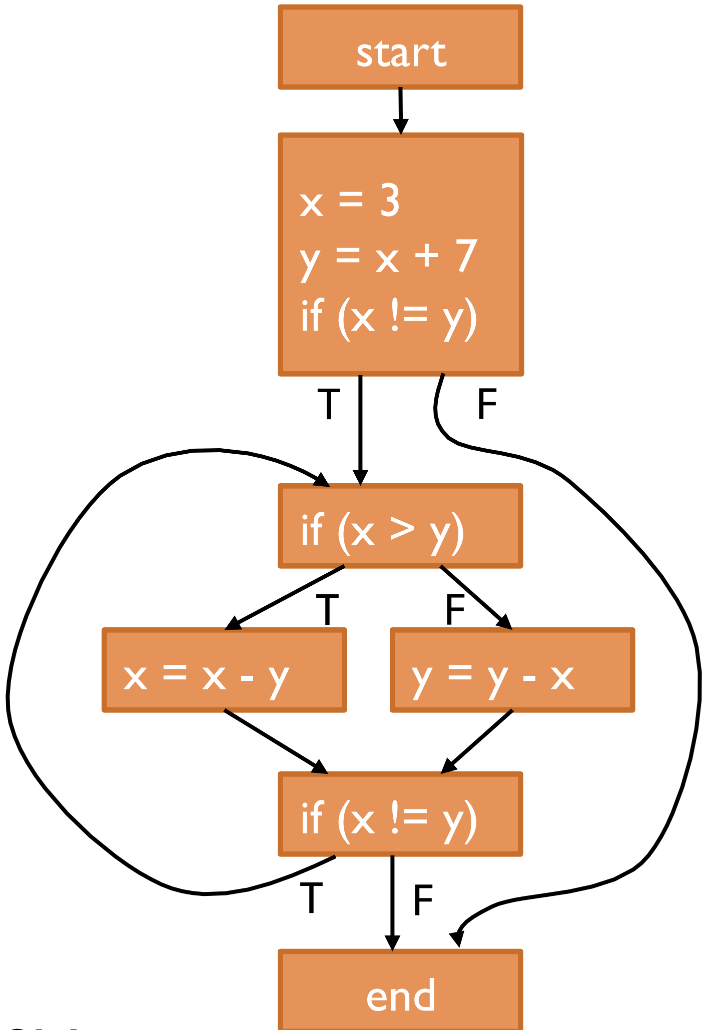
- Basic block: Sequence of assignments with an optional branch at the end

```
x = 3
y = x + 7
if (x != y)
```

- Control flow graph:
  - Nodes: Basic blocks
  - Edges: branches between basic blocks

Control Flow Graph for GCD

```
int x = 3;
int y = x + 7;
while (x != y) {
  if (x > y) {
    x = x - y;
  } else {
    y = y - x;
  }
}
```



Looks like a high-level FSM…

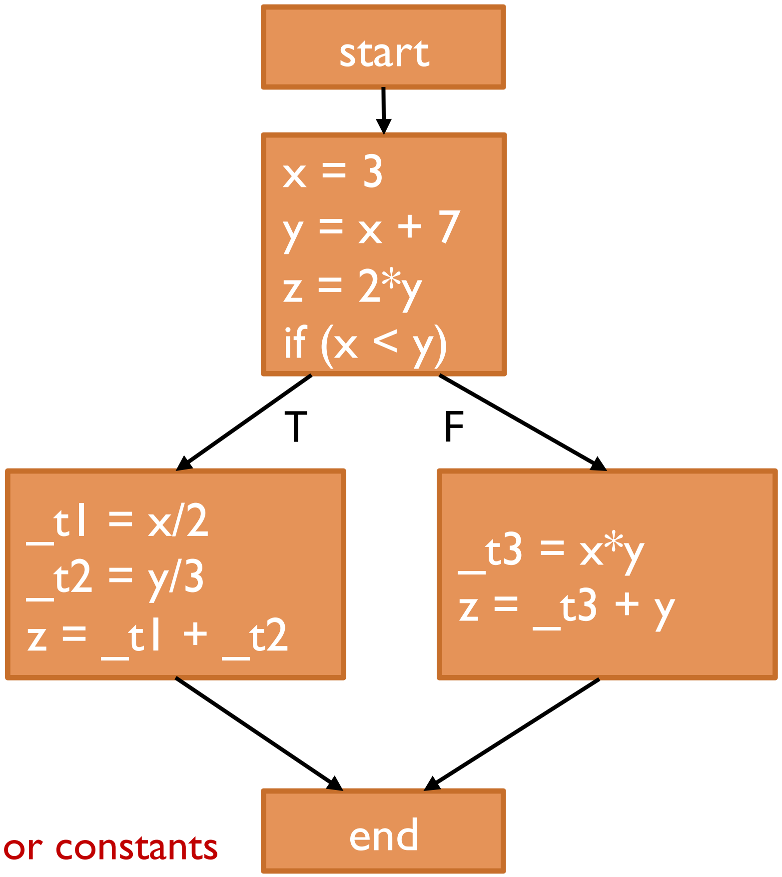
IR Optimization

- Perform a set of passes over the CFG
  - Each pass does a specific, simple task over the CFG
  - By repeating multiple simple passes on the CFG over and over, compilers achieve very complex optimizations

- Example optimizations:
  - Dead code elimination: Eliminate assignments to variables that are never used, or basic blocks that are never reached
  - Constant propagation: Identify variables that are constant, substitute the constant elsewhere
  - Constant folding: Compute and substitute constant expressions

Example IR Optimizations

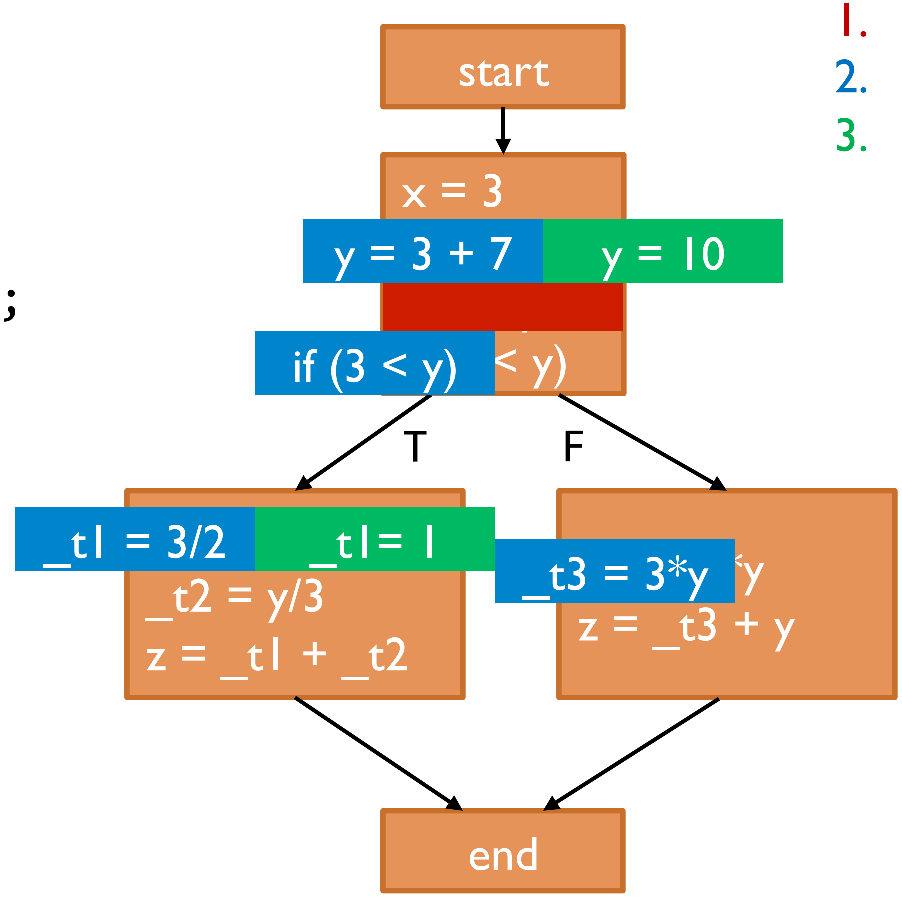
```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
  z = x/2 + y/3;
} else {
  z = x*y + y;
}
```



NOTE: Expressions with > 2 vars or constants broken down in multiple assignments, using temporary variables

Example IR Optimizations

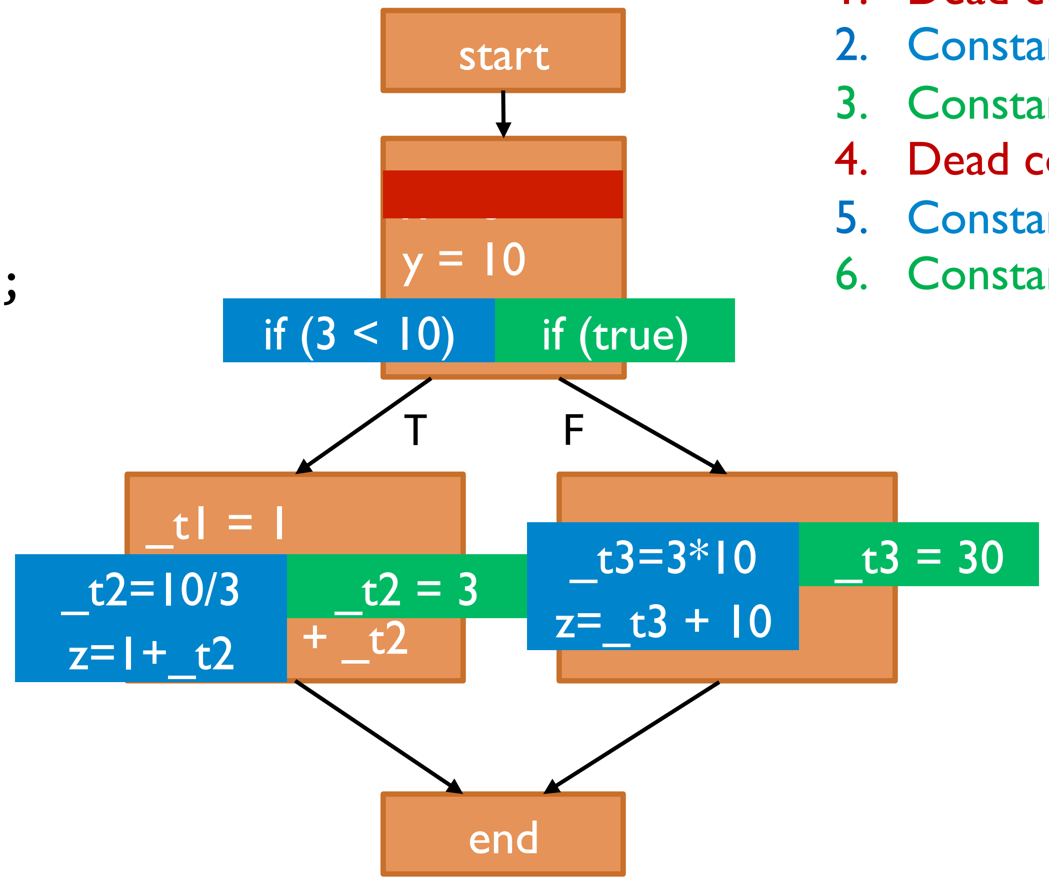
```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
  z = x/2 + y/3;
} else {
  z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding

Example IR Optimizations

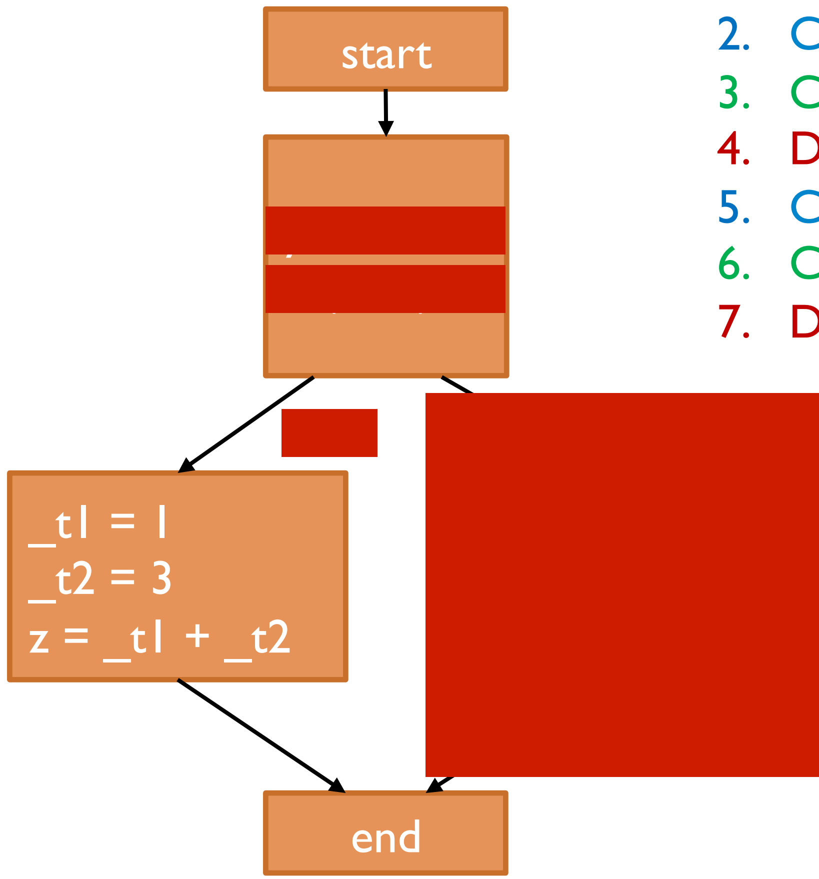
```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
  z = x/2 + y/3;
} else {
  z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding

Example IR Optimizations

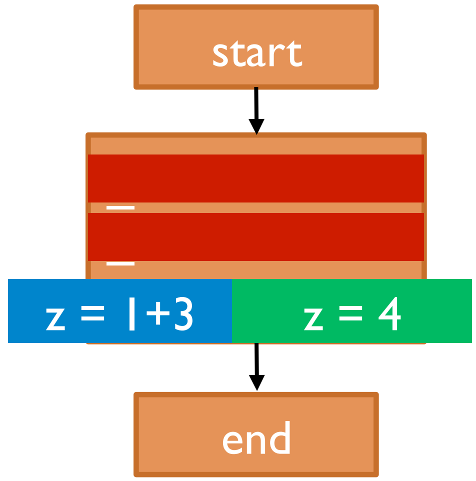
```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
  z = x/2 + y/3;
} else {
  z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim

Example IR Optimizations

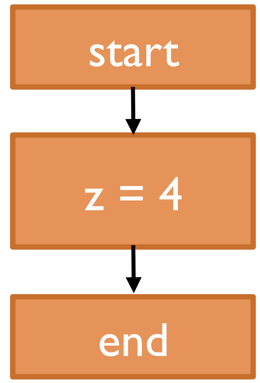
```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
  z = x/2 + y/3;
} else {
  z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding
10. Dead code elim

Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
  z = x/2 + y/3;
} else {
  z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding
10. Dead code elim
11. Constant propagation
12. Constant folding
13. Dead code elim
14. Constant propagation
15. Constant folding

No changes in 13,14, 15 → DONE

Dumb repetition of simple transformations on CFGs

⇩

Extremely powerful optimizations

More optimizations by adding passes: Common subexpression elimination, loop-invariant code motion, loop unrolling…

Code Generation

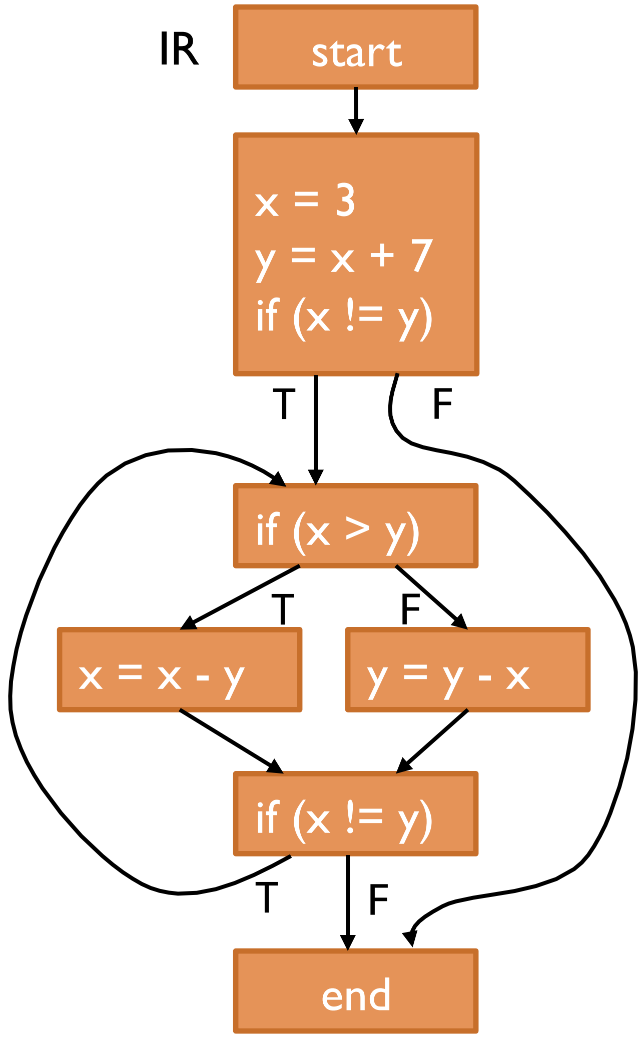
- Translate generated IR to assembly
- Register allocation: Map variables to registers
  - If variables > registers, map some to memory, and load/store them when needed
- Translate each assignment to instructions
  - Some assignments may require > 1 instr if our ISA doesn't have op
- Emit each basic block: label, assignments, and branches
- Lay out basic blocks, removing superfluous jumps
- ISA and CPU-specific optimizations
  - e.g., if possible, reorder instructions to improve performance

Putting It All Together: GCD

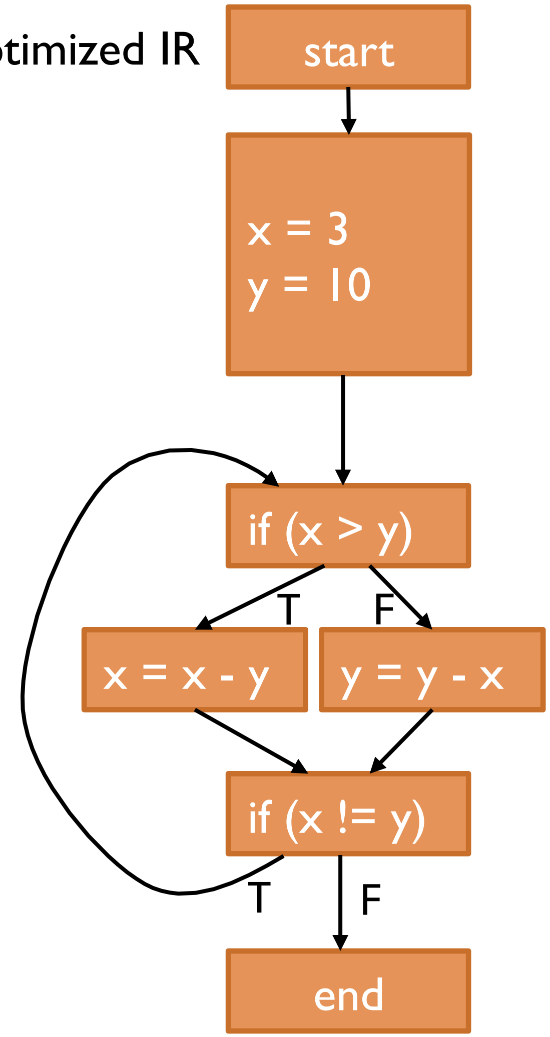
Source code

```
int x = 3;
int y = x + 7;
while (x != y) {
  if (x > y) {
    x = x - y;
  } else {
    y = y - x;
  }
}
```

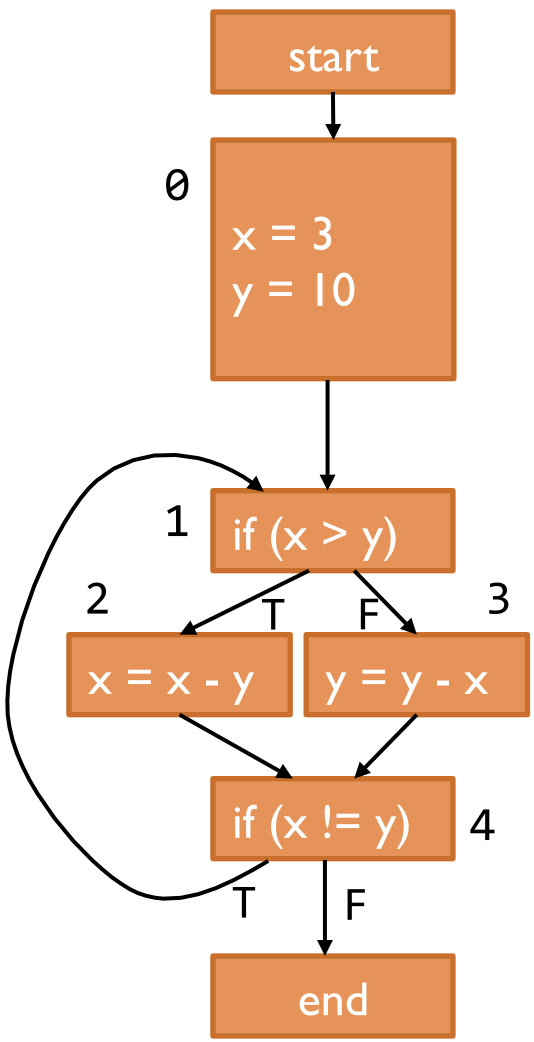
IR



Optimized IR



Putting It All Together: GCD



1. Allocate registers:
`x: R0, y: R1`

2. Produce each basic block:

```
BBL0:    CMOVE(3, R0)
         CMOVE(10, R1)
         BR(BBL1)

BBL1:    CMPLT(R1, R0, R2)
         BT(R2, BBL2)
         BR(BBL3)

BBL2:    SUB(R0, R1, R0)
         BR(BBL4)

BBL3:    SUB(R1, R0, R1)
         BR(BBL4)

BBL4:    CMPEQ(R1, R0, R2)
         BT(R2, end)
         BR(BBL1)
end:
```

3. Lay out BBs, removing superfluous branches:

```
BBL0:    CMOVE(3, R0)
         CMOVE(10, R1)
BBL1:    CMPLT(R1, R0, R2)
         BT(R2, BBL2)
BBL3:    SUB(R1, R0, R1)
         BR(BBL4)
BBL2:    SUB(R0, R1, R0)
BBL4:    CMPEQ(R1, R0, R2)
         BF(R2, BBL1)
end:
```

Summary: Modern Compilers

Frontend (analysis)
Produces IR if correct program
Produces meaningful errors

Backend (synthesis)
Produces optimized program