# CMPE 152: Compiler Design

## November 21 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017

Instructor: Ron Mak

www.cs.sjsu.edu/~mak

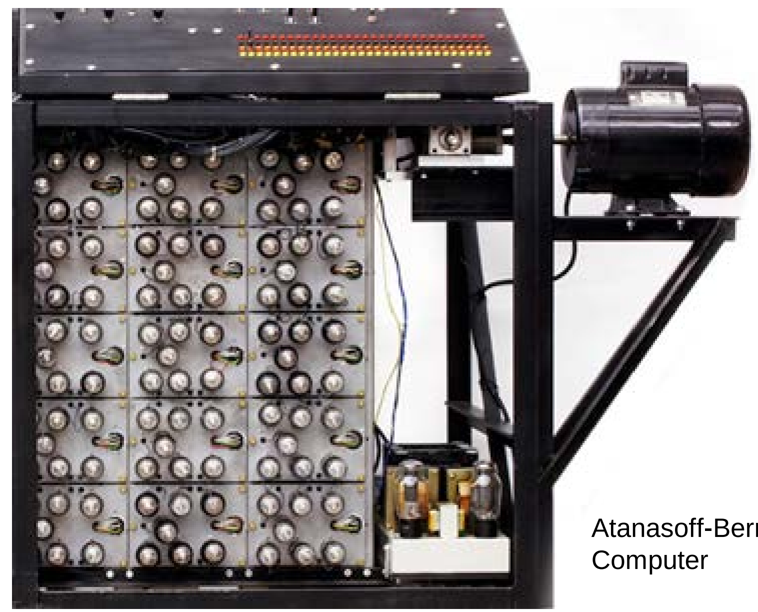
# Unofficial Field Trip

- **Computer History Museum in Mt. View**
  - http://www.computerhistory.org/
  - Provide your own transportation to the museum.
- **Saturday, December 9, 11:30 – closing time**
  - Special free admission.
  - Do a self-guided tour of the Revolution exhibit.
  - See a life-size working model of Charles Babbage's Difference Engine in operation, a hand-cranked mechanical computer designed in the early 1800s.
  - Experience a fully restored IBM 1401 mainframe computer from the early 1960s in operation.

# Unofficial Field Trip, *cont'd*

- See the extensive Revolution exhibit!
  - Walk through a timeline of the
    First 2000 Years of Computing History.
  - Historic computer systems, data processing equipment, and other artifacts.
  - Small theater presentations.

Hollerith Census Machine



Atanasoff-Berry Computer

# Unofficial Field Trip, *cont'd*

- IBM 1401 computer, fully restored and operational.
  - A small transistor-based mainframe computer.
  - Extremely popular with small businesses in the late 1950s through the mid 1960s
    - Maximum of 16K bytes of memory.
    - 800 card/minute card reader (wire brushes).
    - 600 line/minute line printer (impact).
    - 6 magnetic tape drives, no disk drives.

# Unofficial Field Trip, *cont'd*

- Information on the IBM 1401:
  - General info: http://en.wikipedia.org/wiki/IBM_1401
  - My summer seminar: http://www.cs.sjsu.edu/~mak/1401/
  - Restoration: http://ed-thelen.org/1401Project/1401RestorationPage.html

# Instruction Selection

- What sequence of target machine instructions should the code generator emit?
- The symbol table and parse tree are the primary sources of information for the code generator.

# Instruction Selection, *cont'd*

- Retargetable compilers can generate code for multiple target machines.
  - The symbol table and parse tree are source language independent.
- Use code templates that are <u>customized</u> for each target machine.

# Instruction Selection: JVM Examples

- Load and store instructions
  - Emit **`ldc`** ***x*** or **`iconst_`*****n*** or **`bipush`** ***n***
  - Emit **`iload`** ***n*** or **`iload_`*****n***
  - Emit **`istore`** ***n*** or **`istore_`*****n***

- Pascal **CASE** statement
  - Emit **`lookupswitch`** if the test values are sparse.
  - Emit **`tableswitch`** if the test values are densely packed.

# Instruction Selection: JVM Examples, *cont'd*

- Pascal assignment `i := i + 1`
  (assume `i` is local variable #0)

```
iload_0
iconst_1
iadd
istore_0
```

or

```
iinc 0 1
```

# Register Allocation

- Unlike the JVM, many real machines can have <u>hardware registers</u> that are faster than main memory.
  - General-purpose registers
  - Floating-point registers
  - Address registers

- A smart code generator emits code that:
  - Loads values into registers as much as possible.
  - Keeps values in registers as long as possible.
    - But no longer than necessary!

# Register Allocation, *cont'd*

- The code generator assigns registers on a <u>per-routine</u> basis.

- Procedure or function call:
  - Emit code to save the caller's register contents.
  - The procedure or function gets a "fresh" set of registers.

- Return:
  - Emit code to restore the caller's register contents.
  - Better: Save and restore only the registers that a routine uses.

# Register Allocation Challenges

- <u>Limited number</u> of registers.
- May need to spill a register value into memory.
  - Store a register's value into memory
    in order to free up the register.
  - Later reload the value back from memory
    into the register.
- Pointer variables
  - Cannot keep a variable's value in a register
    if there is a pointer to the variable's memory location.

# Data Flow Analysis

- Determine which variables are live.
- A variable ***v*** is live at statement ***p1*** in a program if:
  - There is an execution path from statement ***p1*** to a statement ***p2*** that uses ***v***, and
  - Along this path, the value of ***v*** does not change.
- Only live variables should be kept in registers.

# Instruction Scheduling

- Change the order of the instructions that the code generator emits.
- But don't change the program semantics!
- A form of optimization to increase execution speed.

# Instruction Scheduling, *cont'd*

- With most machine architectures, different instructions take to execute.
  - Example: Floating-point instructions take longer than the corresponding integer instructions.
  - Example: Loading from memory and storing to memory each takes longer than adding two numbers in registers.

# Instruction Scheduling Example

- Assume that **`load`** and **`store`** each takes 3 cycles, **`mult`** takes 2 cycles, and **`add`** takes 1 cycle.

- Simple case:
  Sequential execution only.

| Cycle start | Instruction | Operation |
|---|---|---|
| 1 | `load` | w ➔ *r1* |
| 4 | `add` | *r1* + *r1* ➔ *r1* |
| 5 | `load` | x ➔ *r2* |
| 8 | `mult` | *r1* * *r2* ➔ *r1* |
| 10 | `load` | y ➔ *r2* |
| 13 | `mult` | *r1* * *r2* ➔ *r1* |
| 15 | `load` | z ➔ *r2* |
| 18 | `mult` | *r1* * *r2* ➔ *r1* |
| 20 | `store` | *r1* ➔ w |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| load | | | + | load | | | mult | | load | | | mult | | load | | | mult | | store | | |

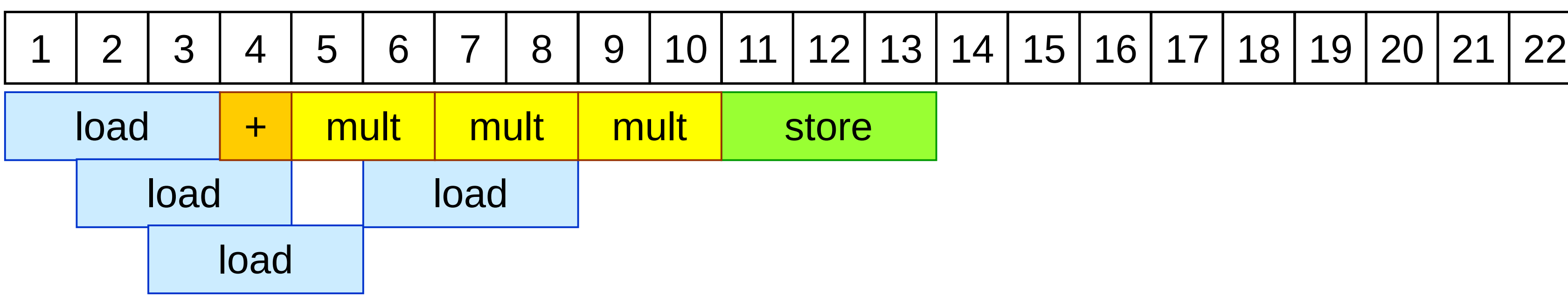
# Instruction Scheduling, *cont'd*

- Assume that **`load`** and **`store`** each takes 3 cycles, **`mult`** takes 2 cycles, and **`add`** takes 1 cycle.

- Assume the machine can overlap instruction execution.
  - instruction-level parallelism

| Cycle start | Instruction | Operation |
|---|---|---|
| 1 | `load` | w → *r1* |
| 2 | `load` | x → *r2* |
| 3 | `load` | y → *r3* |
| 4 | `add` | *r1* + *r1* → *r1* |
| 5 | `mult` | *r1* * *r2* → *r1* |
| 6 | `load` | z → *r2* |
| 7 | `mult` | *r1* * *r3* → *r1* |
| 9 | `mult` | *r1* * *r2* → *r1* |
| 11 | `store` | *r1* → w |

Requires using another register *r3*.



# Introduction to Code Optimization

- Goal: The compiler generates <u>better object code</u>.
- Automatically discover information about the runtime behavior of the source program.
- Use that information to generate better code.

# Introduction to Code Optimization, *cont'd*

- Usually done as one or more passes over the parse tree before the code generator emits the object code.

- The front end parser doesn't worry about optimization.

- A code optimizer in the back end can modify the parse tree so that the code generator will emit better code.

# "Better" Generated Object Code

- Runs faster
  - What people usually mean when they talk about optimization.
- Uses less memory
  - Embedded chips may have limited amounts of memory.
- Consumes less power
  - A CPU chip may be in a device that needs to conserve power.
  - Some operations can require more power than others.

# Code Optimization Challenges: Safety

- The code optimizer <u>must not change the semantics</u> of the source program.

- During execution, the optimized object code must have the same runtime effects as the unoptimized object code.
  - "Same effect": The variables have the same calculated values.

- <u>Bad idea</u>: Compute the wrong values, but faster!

# Code Optimization Challenges: Profitability

- Good optimization is difficult
  to implement correctly.
- It is time-consuming to run an optimizer.
- Optimization can increase compilation time
  by an order of magnitude or more.
- Is it worth it?

# Speed Optimization: Constant Folding

- Suppose we have the constant definition:

  ```
  CONST pi = 3.14;
  ```

  and we have the real expression `2*pi`

- Instead of emitting instructions to load 2, load 3.14, and multiply ...

- Simply emit a single instruction to load the value 6.28

# Speed Optimization: Constant Propagation

- Suppose parse tree analysis determines that a variable ***v*** always has the value ***c*** for a given set of statements.
- When generating code for those statements, instead of emitting an instruction to load the value of ***v*** from memory ...
- Emit an instruction to load the constant ***c***.