CS212

Java Practicum

Lecture 2
SaM

Announcements

- http://www.cs.cornell.edu/courses/cs212/
- Part1 coming up!
- not on CMS for 212? e-mail any 212 TA
- Yes, you need to read Chapter 1
- Update on GBA
- More details on compiler project...

What is SaM? Why SaM?

- **From last lecture**:
  - computer stores data and instructions in memory
  - ***fetch-and-decode cycle***:
  - JVM is ________ of computers
  - bytecodes are _______________
- ***SaM***:
  - stands for: ______________________
  - see SaM on CS212 for full instruction set
  - gives us legible instruction set
    - your compiler will generate ________
    - BTW, what's a compiler? (last panel...)

Samcode Instructions

- **Low-level instructions**:
  - push and pop values in memory
  - ***mnemonics*** for bit patterns
- **Structure**:
  `opcode`
  `opcode operand`
- **Areas** (watch DIS play w/SaM)

Structure of Samcode File

- ASCII Text! (What's ASCII?)
- Write instructions on new lines
- `//` indicates single-line comments, which are ignored
- Program ends with __________
- Program must leave how many items on Stack?

5

Focus on Stack

- ***Call Stack*** (and other names):
  - function calls function calls ...
  - when last function done, go back, then back, then ...
  - how to picture this structure?
- ***Frame***:
  - each function's portion of Stack
  - variables, data, administrative info
- Cells and addresses
  - **start at 0!**
- Helpful picture?

6

Useful Registers

- ***Frame Based Register*** (***FBR***)
  - administrative information
  - keeps track of current frame (and thus, function)
- ***Stack Pointer*** (***SP***)
  - uses register
  - store location of next free cell in stack
- Helpful picture?

7

Some Instructions

- **ALU**:
  - arithmetic, boolean, comparison
  - generally follows ***below op top***
  - usually pops both values and pushes result
- **Stack Manipulation**:
  - pushing
  - swapping, duplicating
  - storing, retrieving
- **Register**
- **Control**
- Descriptions:
  - see on-line documentation
  - see **Chapter 1**

8

Some Examples

- Notation:
  - Infix: (1 – 2) – 3
  - Postfix: 1 2 – 3 –
- Logical: ~(4 <= 5)
  - Samcode rem: ***below op top***
- Samcode?

9

Program Storage?

- Main memory model:
  - store programs as __________
  - so, instructions have patterns of _______
- Where are they in SaM?
  - Samcode read into an array
  - array stores instruction objects
- Want more? See documentation and source code
  - **SaM→Individual Files→Core→Instructions**
  - See next page for example
- How to load your own instructions?
  - recompile everything (a pain)
  - or...use SaM's ***instruction loader***

10

Example

```
package edu.cornell.cs.sam.core.instructions;
import edu.cornell.cs.sam.core.*;

public class SAM_ADD extends SamInstruction {
    public void exec() throws SystemException {
        int type1 = mem.getType(cpu.get(SP) - 2);
        int type2 = mem.getType(cpu.get(SP) - 1);
        mem.push(higherPrecedence(type1, type2), mem.pop() + mem.pop());
        cpu.inc(PC);
    }
}
```

11

Variable Scope

- Take an aside... is SaM really useful?
- Example:
  - is the following legal?

    ```
    int x(int x) { return x++; }
    int y(int x) { return x(x); }
    ```
  - why? why not?
- Scope of variable:
  - region of code in which variable represents something
  - how does Java indicate?
- Local and global variables:
  - each function has its own local variables
  - global variables shared
- Does SaM help?

12

Variables and Frames

- **A way to picture variables in frames...**
  - variable gets cell
  - Aside: SaM shows type of cell
- **Samcode program**:
  - allocate cell
  - fill cell
  - later retrieve/change contents
  - finally deallocate cell (why?)

13

Allocation and Deallocation

- Pushing:
  - `PUSHIMM`... (see SaM website)
- Allocating:
  - Allocate *v* amount of vars: `ADDSP v`
  - Deallocate *v* amount of vars: `ADDSP -v`
- Example:

```
ADDSP 3
ADDSP -1
ADDSP -1
ADDSP -1
STOP
// error mesg (why?)
```

14

How to access a variable?

- **Addressing of variables**:
  - absolute
  - relative
- ***Absolute***:
  - don't worry about your current frame
  - figure out variable address on stack
  - eg) globals
- ***Relative***:
  - do worry about your current frame
  - figure out variable address with respect to FBR value
  - eg) locals

15

Absolute Address

- **Instructions**:
  - To **store** a value *v* at location *i*:
    - `PUSHIMM v`: Stack[SP] ← *v*; SP++
    - `STOREABS i`: Stack[*i*] ← Stack[SP-1]; SP--
  - To **retrieve** a value *v* from location *k*:
    - `PUSHABS k`; Stack[SP] ← Stack[*k*]; SP++
- **Example:**

```
int rv;
int x;
int y;
x = 10;
y = 20;
rv = x + y;
return rv;
```

```
ADDSP 3
PUSHIMM 10
STOREABS 1
PUSHIMM 20
STOREABS 2
PUSHABS 1
PUSHABS 2
ADD
STOREABS 0
ADDSP -2
STOP
```

16

Relative Address

- Instructions:
  - To store a value ***v*** at location ***i***:
    - **`PUSHIMM`** ***v***: Stack[SP] ← ***v***; SP++
    - **`STOREOFF`** ***i***: Stack[***i***+FBR] ← Stack[SP-1]; SP--
  - To retrieve a value ***v*** from location ***k***:
    - **`PUSHOFF`** ***k***: Stack[SP] ← Stack[***k***+FBR]; SP++
- Picture?

17

Example

```
ADDSP 1      // rv of program
JSR add      // new frame (jump to "add")
STOREOFF 0   // store rv of "add"
STOP         // done

add:         // code for "add" function
LINK         // store old FBR (0) and set new FBR (2)
ADDSP 3      // allocate space for x, y, rv of add
             // rv of add is at relative address 1
PUSHIMM 10   // push value 10
STOREOFF 2   // store 10 in x's cell
PUSHIMM 20   // push value 20
STOREOFF 3   // store 20 in y's cell

PUSHOFF 2    // retrieve x
PUSHOFF 3    // retrieve y
ADD          // x+y
STOREOFF 1   // store x+y as rv of add
ADDSP -2     // deallocate x, y

SWAP         // exchange rv of add for old FBR
UNLINK       // restore old FBR (0)
SWAP         // exchange rv of add for return address
RST          // return to Samcode just after "JSR add"
```

```
public int add()
   int x, y;
   x = 10;
   y = 20;
   return x+y;
}
```

NOTE: We will use a different frame structure later!

18

Human Compiling

- ***Compiling***:
  - translate **`code`** (like Java) to machine **`code`** (like Samcode)
  - compiler (like **`javac`**) does the work for you
- ***Human Compiling*** (Part 1 of CS212):
  - you identify simple expressions and statements
  - you convert them into Samcode
  - you test your Samcode problems in SaM
  - we grade your correctness and style

19