MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring 2007

Recitation 5/9
Register Machines

Expression Types

`(const C)` A constant value. It acts somewhat like quote. To get the number one, you would use `(const 1)`.

`(reg R)` Retrieve the value of a register R. To get the value of the register `arg0`, you would use `(reg arg0)`.

`(label L)` Retrieve the offset of the given label L. To get the value of the label `loop-top`, you would use `(label loop-top)`.

`(op O)` Perform operation O on some values. Following the `(op O)`, you should list the input arguments to the operation, which may be consts, regs, or labels. An expression may only contain 1 op. In order to compute the result of adding 1 to the register arg0, you would use
`(op +) (reg arg0) (const 1)`.

Instruction Types

`(assign reg` *expr*`)`

Sets register reg to be the result of expression *expr*. The assigned register doesn't need a tag because it is always a register being assigned. For example, to increment the result register:
`(assign result (op +) (reg result) (const 1))`

`(goto` *expr*`)`

Sets the `pc` to be the result of *expr*, which is usually a label or a register. Effectively continues the execution at another point in the code. To jump to the label `loop-top`:
`(goto (label loop-top))`

`(test` *expr*`)`

This is equivalent to assigning the `cr`. The `cr` register is used to determine whether to take a `branch`. For example, to set the `cr` based on whether the register `x` is less than 10:
`(test (op <) (reg x) (const 10))`

`(branch` *expr*`)`

If the value in the `cr` is true, acts like a `goto`. Otherwise it does nothing. To conditionally jump to the label `loop-done`:
`(branch (label loop-done))`

Writing Code

Write `double`: code to compute $2x$, given x in `arg0`, and leave the output in `result`.

```
double
 (assign result (op *) (reg arg0) (const 2))
 (goto (reg continue))
```

1. Write `func`: code to compute $x^2 + y$, given x in `arg0`, y in `arg1`, and leave the output in `result`.

2. Write `abs`: code to compute $|x|$, give x in `arg0`, leave the output in `result`. `abs` is *not* an available primitive.

3. Write `infinite-loop`: code that never halts.

4. Determine what the following code does, then write the scheme code that does the same thing.

```
foo
 (test (op <) (reg arg0) (reg arg1))
 (branch (label foo-done))
 (assign arg0 (op -) (reg arg0) (reg arg1))
 (goto (label foo))
foo-done
 (assign result (op =) (reg arg0) (const 0))
 (goto (reg continue))
```

Contracts

Input Register(s) whose value is read and used before it is written.

Output Register(s) designated as output.

Modifies Register(s) whose value after the code block *could* differ from their original value.

1. What is the contract for the following code:

```
expt
 (assign result (const 1))
expt-loop
 (test (op <=) (reg arg1) (const 0))
 (branch (reg continue))
 (assign result (op *) (reg result) (reg arg0))
 (assign arg1 (op -) (reg arg1) (const 1))
 (goto (label expt-loop))
```

   Input:
   Output:
   Modifies:

2. What is the contract for the following code:

```
foo
 (assign y (reg x))
 (assign x (op cons) (reg x) (reg y))
 (test (op null?) (reg x))
 (branch (label yack))
 (assign val (const 2))
 (assign x (reg y))
 (goto (reg continue))
yack
 (assign foo (const 7))
 (assign val (op car) (reg x))
 (goto (reg continue))
```

   Input:
   Output:
   Modifies:

Save and Restore

`(save` *reg)*

Place the value in register *reg* on top of the stack. To place the value in the register `result` on the stack:
`(save result)`

`(restore` *reg)*

Take the top value off the stack and put it in register *reg*. To remove the top element of the stack and place it in the register `result`:
`(restore result)`

Procedure Call

1. `save` things you care about
2. `assign` values to the inputs, including `continue` to an appropriate label
3. `goto` the procedure's label
4. return label
5. `restore` things you cared about, in reverse order

Problems

3. Implement `aexpb`, which computes ae^b. You should call `expt` in your solution.