CS415 Homework 1

Sample solution

Spring 2018

Problem 1. ILOC programming - Factorial

Register-register model

There are several possible solutions, depending on how much knowledge you assume the compiler has about the code.

In class, we talked about local register allocation. Here, assigning values to registers are done for a single basic block without the knowledge of any surrounding basic blocks, or how the control flows from one basic block to the next. Particularly, local register allocation does not know that a basic block is the body of a loop.

Solution (A): Compiler knows that the basic block is part of the loop. It also may know that variable "a" is not used after the loop, so there is no need to write it back to memory. This allows the compiler to keep the value of "a" entirely in a register. The same holds for "result", until it needs to be printed out which requires the value to be assigned to memory (outputAI).

```
loadI 1024 => r0   // base address
loadI 5 => r1      // a := 5
loadI 1 => r2      // result := 1
loadI 1 => r3      // loop termination condition

L0: nop
cmp_GT r1, r3 => r4
cbr r4 => L1, L2

L1: nop
mult r2, r1 => r2  // result := result * a;
                   // reassigns r2, but this is needed since ‘‘result’’ lives in r2
subI r1, 1 => r1   // a := a - 1;
br L0

L2: nop
storeAI r2 => r0, 4 // must write back ‘‘result’’ into memory location at @result = 4
outputAI  r0, 4     // print out "result"
```

Solution (B): Compiler knows nothing about the context of the basic block. This means that no values can be kept in registers across the end of a basic block. This means that modified variables ("a" and "result") have to be reloaded at the beginning of the basic block, and stored back at the end since they have been modified.

```
loadI 1024 => r0   // base address
loadI 5 => r1
storeAI r1 => r0, 0  // offset of ‘‘a’’ is 0, i.e., @a = 0
loadI 1 => r2
storeAI r2 => r0, 4  // offset of ‘‘result’’ is 4, i.e., @result = 4
loadI 1 => r3      // loop termination condition

L0: nop
loadAI r0, 0 => r1 // r1 contains value of ‘‘a’’
cmp_GT r1, r3 => r4
cbr r4 => L1, L2

L1: nop
loadAI r0, 4 => r2 // r2 contains value of ‘‘result’’
mult r2, r1 => r5  // result := result * a;
subI r1, 1 => r6   // a := a - 1;
storeAI r5 => r0, 4 // write back to memory ‘‘result’’
storeAI r6 => r0, 0 // write back to memory ‘‘a’’
br L0

L2: nop
outputAI r0, 4 // offset of ‘‘result’’ is 4, i.e., @result = 4
```

Memory-memory model

```
loadI 1024 => r0

loadI 5 => r1        // a := 5
storeAI r1 => r0, 0

loadI 1 => r2        // result := 1
storeAI r2 => r0, 4

loadI 1 => r3        // loop termination condition
storeAI r3 => r0, 8

L0: nop
loadAI r0, 0 => r4
loadAI r0, 8 => r5
cmp_GT r4, r5 => r6
```

```
cbr r6 => L1, L2

L1: nop
loadAI r0, 4 => r7   // result := result * a;
loadAI r0, 0 => r8
mult r7, r8 => r9
storeAI r9 => r0, 4

loadAI r0, 0 => r10  // a := a - 1;
subI r10, 1 => r11
storeAI r11 => r0, 0
br L0

L2: nop
outputAI r0, 4 // offset of ‘‘result’’ is 4, i.e., @result = 4
```

Problem 2. Feasible Registers for ILOC

The minimum number of registers needed to execute any ILOC instruction when all operands are in memory is **three.** The base register **r0** does not count since it is assumed to be a special purpose register that cannot be allocated.

The `storeAO` command reads the values from three registers, which all need to be available at that time.

```
ILOC store operations:
store r1 => r2       // r1 => MEMORY(r2)
storeAO r1 => r2,r3  // r1 => MEMORY(r2+r3)
storeAI r1 => r2,c3  // r1 => MEMORY(r2+c3)
```

Other commands can be satisfied with two registers, since they involve reading from one/two registers and writing to one. Worst case: Values for both operands have to be loaded from memory, this can be done as follows for operation a := a + b:

```
loadAI r0,0 => r1      // load first value @a = 0
loadAI r0,4 => r2      // load second value @b = 4
add r1,r2 => r1        // add values and reuse register
storeAI r1 => r0, 0   // store result in @a = 0
```

Problem 3. Top-down register allocation

3.1 register-register ILOC code

```
loadI 1024 => r0
loadI 1 => r1         // a := 1
loadI 2 => r2         // b := 2
subI r2, 4 => r3      // c := b - 4
add r1, r2 => r4      // d := a + b
addI r4, 1 => r5      // e := d + 1
mult r3, r5 => r6     // c * e
sub r5, r6 => r7      // f := e - (c * e)
add r4, r5 => r8      // d + e
add r8, r7 => r9      // g := (d + e) + f
add r9, r1 => r10     // h := g + a
storeAI r10 => r0, 4 // printing requires value to be in memory
outputAI r0, 4       // print @h = 4 , h is only value in memory
```

3.2 Live ranges

```
loadI 1024 => r0
loadI 1 => r1      // r1
loadI 2 => r2      // r1 r2
subI r2, 4 => r3   // r1 r2 r3
add r1, r2 => r4   // r1    r3 r4
*addI r4, 1 => r5  // r1    r3 r4 r5
*mult r3, r5 => r6 // r1       r4 r5 r6
*sub r5, r6 => r7  // r1       r4 r5    r7
add r4, r5 => r8   // r1                r7 r8
add r8, r7 => r9   // r1                      r9
add r9, r1 => r10  //                            r10
storeAI r10 => r0, 4
outputAI r0, 4
```

MAX_LIVE is 4.(instructions with *)

3.3(a) top-down by number of accesses(EAC)

Register	Number of accesses
r1	3
r2	3
r3	2
r4	3
r5	4
r6-r10	2

Since we only have MAX_LIVE-1 = 4-1 = 3 registers to allocate, we choose virtual registers r5, r1, and r2. One possible resulting ILOC code is listed below:

```
// Feasible registers: r1(f1), r2(f2)
// Available physical registers for allocation: r3, r4, r5
// Mapping of virtual registers (live ranges) to physical registers / memory:
//    r1 -> r4,
//    r2 -> r5
//    r3 -> memory: @r3 = -4
//    r4 -> memory: @r4 = -8
//    r5 -> r3
//    r6 -> memory: @r6 = -12
//    r7 -> memory: @r7 = -16
//    r8 -> memory: @r8 = -20
//    r9 -> memory: @r9 = -24
//    r10 -> memory: @r10 = -28

loadI 1024 => r0
loadI 1 => r4
loadI 2 => r5
subI r5, 4 => r1       // "subI r2, 4 => r3"
storeAI r1 => r0, -4   // @r3 store
add r4, r5 => r1       // "add r1, r2 => r4"
storeAI r1 => r0, -8   // @r4 store
loadAI r0, -8 => r1    // @r4 load
addI r1, 1 => r3       // "addI r4, 1 => r5"
loadAI r0, -4 => r1    // @r3 load
mult r1, r3 => r1      // "mult r3, r5 => r6"
storeAI r1 => r0, -12  // @r6 store
loadAI r0, -12 => r1   // @r6 load
sub r3, r1 => r1       // "sub r5, r6 => r7"
storeAI r1 => r0, -16  // @r7 store
loadAI r0, -8 => r1    // @r4 load
add r1, r3 => r1       // "add r4, r5 => r8"
storeAI r1 => r0, -20  // @r8 store
loadAI r0, -20 => r1   // @r8 load
loadAI r0, -16 => r2   // @r7 load
add r1, r2 => r1       // "add r8, r7 => r9" - needs two feasible registers
storeAI r1 => r0, -24  // @r9 store
loadAI r0, -24 => r1   // @r9 load
add r1, r4 => r1       // "add r9, r1 => r10"
storeAI r1 => r0, -28  // @r10 store
loadAI r0, -28 => r1   // @r10 read
storeAI r1 => r0, 4    // "storeAI r10 => r0, 4"
outputAI r0, 4         //
```

3.3(b)top-down by length of live range (class)

By number of accesses, using length of the live range as a tie breaker (spill the longer live range).

```
// Feasible registers: r1(f1), r2(f2)
// Available physical registers: r3, r4, r5
// Mapping of virtual registers (live ranges) to physical registers / memory:
//
//   MAX_VAL = 4  -- process instructions in textual order:
//     - instruction: "addI r4, 1 => r5"   --> spill r3
//     - instruction: "mult r3, r5 => r6"  --> spill r6
//     - instruction: "sub r5, r6 => r7"   --> spill r7
// NOTE: spilling r4 would have been a better choice, but our algorithm looks only  at single instruct
//    r1 -> r3
//    r2 -> r4
//    r3 -> memory: @r3 = -4
//    r4 -> r5
//    r5 -> r4
//    r6 -> memory: @r6 = -8
//    r7 -> memory: @r7 = -12
//    r8 -> r5
//    r9 -> r5
//    r10 -> r5

loadI 1024 => r0
loadI 1 => r3
loadI 2 => r4
subI r4, 4 => r1       // "subI r2, 4 => r3"
storeAI r1 => r0, -4   // @r3 store
add r3, r4 => r5       // "add r1, r2 => r4"
addI r5, 1 => r4       // "addI r4, 1 => r5"
loadAI r0, -4 => r1    // @r3 load
mult r1, r4 => r1      // "mult r3, r5 => r6"
storeAI r1 => r0, -8   // @r6 store
loadAI r0, -8 => r1    // @r6 load
sub r4, r1 => r1       // "sub r5, r6 => r7"
storeAI r1 => r0, -12  // @r7 store
add r5, r4 => r5       // "add r4, r5 => r8"
loadAI r0, -12 => r1   // @r7 load
add r5, r1 => r5       // "add r8, r7 => r9"
add r5, r3 => r5       // "add r9, r1 => r10"
storeAI r5 => r0, 4    // "storeAI r10 => r0, 4"
outputAI r0, 4         //
```

NOTE: We used a simple code generation strategy: spilled registers are loaded just before needed, and stored right after definition. In some cases, a spilled register is stored to memory using a feasible register, immediately followed by a load back to the same physical register. We will see later how local optimizations, called "peephole optimizations" can catch such inefficiencies.