CS 4240: Compilers and Interpreters
Phase 3: Code generation

Total Points: 125 + Bonus

Due Date: November 30th (Sunday), 2014

This phase of the project consists of two parts plus bonus parts. You will be allocating registers for the IR you produced in phase 2, and you will be selecting the final MIPS instructions. For bonus points, we have two parts: you can do a complete intra-procedural register allocator and/or you can implement function calls. You will be able to run your output on a MIPS simulator by the end of this phase.

Part 1: Register Allocation

Your stream of IR instructions from phase 2 relied on an unlimited number of temporaries to store intermediate results. In this part, the goal is to allocate these temporaries in registers and achieve an instruction stream that is faithful to MIPS' register set. Wikipedia's page on MIPS includes details on all of the registers you will need for this part:
 http://en.wikipedia.org/wiki/MIPS_architecture

The general idea of register allocation is straightforward. At any given point, the processor can only hold a finite number of values in its registers, and the values being manipulated need to be in those registers. If there are not enough registers, we have ***register spill***, and extra store and load instructions are needed. The way to handle register spill is to use memory. Because a ***live variable*** will be used in the near future, it must not only be stored in memory but also loaded back into the register file before its use. All variables are allocated space during compile-time in the .data memory segment (see part 2). When a variable must be brought into a register, its memory address in this segment is known. Likewise, when it must be stored, it will be stored at this known address.

There is a large body of knowledge surrounding register allocation, and it is a critical compiler phase for good performance for obvious reasons. For the purposes of this project, we are interested in correctness but not high levels of performance. You will implement three different register allocation schemes.

Naive

The most naive allocation scheme is one in which there is no analysis. Before each instruction, its operands are loaded into registers; the instruction then executes; and finally the result is stored back into that variable's home location in memory. Thus, for each instruction in the IR stream, you will generate and insert the necessary load(s) before that instruction, and you will generate and insert the necessary store(s) after that instruction. This scheme is the slowest, but it will produce correct, working code.

Note: Below, everyone first must do the CFG construction and Intra-block allocation. Then you have to choose between: (a) EBB allocation or (b) the bonus part on whole CFG register allocation. You must choose between (a) and (b) though. If you do part (b) above, you will get extra 20 bonus points.

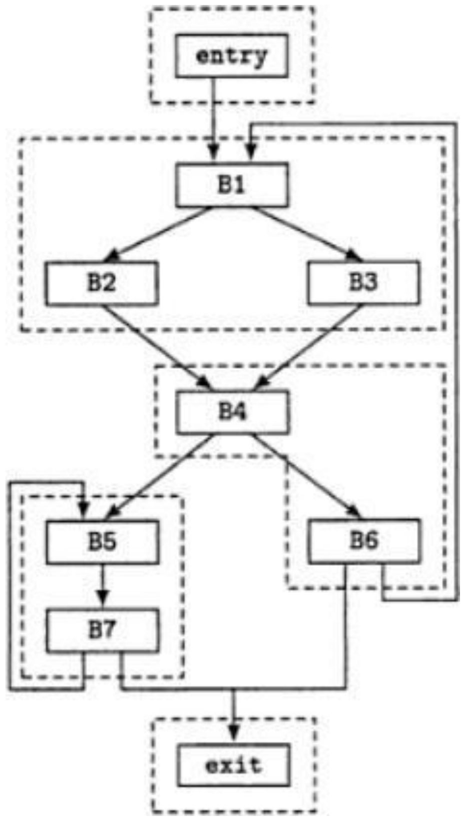
CFG Construction and Intra-block Allocation

An improvement on the naive scheme is to locate the ***basic blocks*** in the stream, construct a ***control flow graph*** (CFG) of those blocks, and carry out ***graph coloring*** at the intra-block level (i.e. only within each basic block). We have provided the resources for how to detect basic blocks, build the control flow graph, and perform graph coloring on each basic block. Notice that at the start of each block, you will need to load a set of variables that you expect to use, and similarly, at the end of each block, you will need to store all values from your registers to memory. Simple heuristic you can use for the allocation could be this: Detect the total number of live variables intra-block and check if those are below 32, if they are, allocate those in registers else pick the best 32 that have highest spill costs and allocate them. You can calculate the spill cost by the number of loads/stores that will be saved if the value were to be allocated in the register.

EBB Construction and Intra-EBB Allocation

Extended basic blocks (EBBs) allow us to perform register allocation on a larger section of code (thereby allowing for better register allocation). An extended basic block is a collection of connected basic blocks such that only the first basic block may have more than one predecessor in the control flow graph. For illustration, consider Figure 1. Each dashed box encloses an EBB. A complete algorithm for constructing all EBBs is given in Algorithm 1. (***Source: Advanced Compiler Design Implementation by Muchnick)***

Figure 1: Control flow graph with extended basic blocks enclosed in dashed boxes



Algorithm 1: Construction of all EBBs

```
entry: Node EbbRoots:
set of Node
AllEbbs: set of (Node × set of Node)

procedure main()
begin
    Build_All_Ebbs(entry, Succ, Pred)
end

procedure Build_All_Ebbs(r, Succ,
    Pred) r: in Node
    Succ, Pred: in Node ⟶ set of Node
begin
    x: Node
    s: Node × set of Node
    EbbRoots := {r}
    AllEbbs := ∅

    while EbbRoots ≠ ∅ do
        x := ♦ EbbRoots
        EbbRoots -= {x}
        if ∀s ∈ AllEbbs (s@1 ≠ x) then
            AllEbbs ∪= {<x, Build_Ebb(x, Succ, Pred)>}
        fi
    od
end

procedure Build_Ebb(r, Succ, Pred) returns set of
    Node r: in Node
    Succ, Pred: in Node ⟶ set of Node
begin
    Ebb := ∅: set of Node
    Add_Bbs(r, Ebb, Succ, Pred)
    return Ebb
end
```

```
procedure Add_Bbs(r, Ebb, Succ,
      Pred) r: in Node
      Ebb: inout set of Node
      Succ, Pred: in Node → set of Node
begin
      x: Node
      Ebb ∪=
      {r}
      for each x ∈ Succ(r) do
            if |Pred(x)| = 1 & x ∉ Ebb then
                  Add_Bbs(x, Ebb, Succ,
                  Pred)
            elif x ∉ EbbRoots
                  then EbbRoots
                  ∪= {x}
            fi
      od
end
```

Once all EBBs have been constructed, you will allocate registers within an EBB using the same graph coloring technique from the previous scheme. Similarly, you will load needed values at the start of an EBB and store values at the end of the EBB.

Bonus:
In lieu of EBB oriented register allocation, you can implement the liveness analysis for the complete CFG and build live ranges and webs and then use graph coloring algorithm on the slides to decide the register allocation. If you decide to do this part, after constructing the CFG, you should first set up the data-flow equations (please use the lecture slides on liveness analysis) and then solve them in a backward manner. Then you will build the live ranges, you will then scan backwards and build the webs by connecting the definitions to uses and their transitive closures. You will then build interference graph and solve the same using graph coloring techniques (refer to the stack oriented algorithm in register allocation slides).

Your register allocation implementations should take correct IR code as input and produce modified IR code for use in part 2. We do not need to see the output of this part, but it may be useful to produce an output file for debug purposes.

Part 2: Instruction Selection and Code Generation

You only have to write a single implementation of instruction selection and code generation. Any of the three implementations from part 1 should produce IR code with correct register allocation, and this stream will be used as input to your part 2 solution.

The instruction selection is a matter of converting the IR code you've generated to the appropriate MIPS assembly code. The IR code from phase 2 is actually a relatively close match to the MIPS code you are expected to generate (with some exceptions). Wikipedia's page on the MIPS architecture includes the assembly instruction supported by MIPS. Another challenge is ensuring the .data segment is created correctly. This link may help to understand this:

http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/dataseg.html

You will be running your generated MIPS code on a simulator called SPIM (see the end of this document). For your code to run properly on SPIM, you should generate a "main" label where your program's statement sequences begin.

You must also generate a final instruction, "jr $ra", to return to the caller of your program. Supplied examples will help demonstrate this.

Also, unless you are implementing function calls for bonus points, you may safely ignore instruction selection for function calls, as well.

This part of the project should take the modified code produced in part 1 and output correct assembly code. Your final assembly should run on SPIM.

Part 3 (bonus): Function Calls

If you choose to support function calls for bonus points, then hack on! There are of course many resources on the internet on MIPS calling conventions. This link is a good start:

http://people.cs.pitt.edu/~xujie/cs447/Mips/sub.html

Turn-in

Grading

1. Register allocation code (60 points or 80 points)
   a. Naive (20 points)
   b. CFG and intra-block allocation (25 points)
   c. EBB and intra-EBB allocation (15 points) **OR**
   Bonus Whole function Register Allocation (35 points)
2. Instruction selection and generation code (35 points)
3. Passes tests using generated code executing on simulator (20 points)
4. Report (design internals, how to build, run, etc.) (10 points)

Bonus:

1. Function calls (working MIPS code) (40 points)

SPIM

SPIM is a MIPS simulator you will use to run your compiled, MIPS assembly programs. To download SPIM, use the following link:

 http://spimsimulator.sourceforge.net/

Once SPIM is running, execute the hello-world example: File > Load File

Select <path-to-spim-simulator>/helloworld.s

Click the green arrow to run it

SPIM console should display “Hello World”

Troubleshooting:

If you run into trouble with their binary package, you can download the source with svn: svn

checkout svn://svn.code.sf.net/p/spimsimulator/code/ spimsimulator-code

Then you can build the source with QtCreator. Use this to get the most recent Qt:

https://qt-project.org/downloads

After running Qt-Creator, open the project:

<path-to-spim-simulator>/QtSpim/QtSpim.pro.

Build and run the project. This should launch SPIM, and then you can proceed with the hello-world example.