CS 8803-008: Project Phase 2
Register allocation, Instruction selection and Code generation

Total Points: 100

Due Date: 7/23/2019, 7 am EDT

This phase of the project consists of generating MIPS assembly code from the intermediate representation (IR). You have generated IR in the quad format in your phase I as per the attached specification and example given in Appendices A and B below that is produced for the language Tiger and in this phase you will be generating the assembly from the same which should execute on a MIPS machine. Your generated assembly code should be able to execute on the MIPS simulator that will be given to you.

Part 1: Register Allocation

A typical IR generates a large number of temporaries to hold the intermediate results of processing an expression. The IR also contains instructions for storage allocation of the variables and temporaries and control flow and branch instructions. Refer to the Appendices A and B that spell out the IR format in details. The goal of this part is to convert the IR into MIPS assembly which should execute on a MIPS machine. In order to do so, one should allocate the program variables (including temporaries) in registers and then generate an instruction stream that is faithful to MIPS' assembly format. MIPS assembly format uses certain register naming and usage conventions. Wikipedia's page on MIPS includes details on all of the register conventions you will need for this part (first study those including some examples):

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 used 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. Refer to the class lessons on control flow graph generation, on liveness analysis and on register allocation for all the details wherein we studied register allocation topic in depth. All variables are allocated space during compile-time in the .data memory segment (see part 2 below). 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 will be implementing a naïve register allocation strategy, and a global (that spans whole function or a procedure) Briggs' style graph coloring register allocator. You will implement these two different register allocation schemes starting with the simplest going towards the complex and compare their performances on the generated code.

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. This is our first goal, all you will need here is to do the necessary load/store and instruction generation to get this working.

For those who are doing it in Java, we will be providing a parser that reads IR and holds it in an internal data structure. Using this data structure, you can perform instruction selection and native code generation by inserting loads and stores as described above. We recommend that you build the native code generator in a step by step manner: start with simple assignment statements, then add some control flow statements if…then…else, then loops etc. Finally you can add array statements. You can construct appropriate Tiger programs with respective statements (assignment, if then else, loops etc) as test cases, check the generated IR and then check the generated MIPS code and debug. You can then construct more and more complex Tiger programs and check the correctness of the generated assembly. Design the native code generation in a manner that you can plug in register allocator phase which will replace or annotate the original IR symbols with register names and annotations. You can then just run the instruction selection pass on it to generate MIPS assembly (note that many load./stores in native code generator will be gone in register allocated code).

Global (whole function) Briggs' style graph coloring allocator:

As the last exercise, you will be building the full inter-block (whole function) register allocator. First you will build the control flow graph, then perform the full CFG liveness analysis across the basic blocks and build live ranges and the webs as discussed in the lecture notes. You will then build an interference graph and perform graph coloring using Briggs' optimistic coloring algorithm to allocate the registers. Refer to all the respective lessons and corresponding book reading materials for all the details.

After performing the register allocation, necessary load store instructions are inserted in the IR and the IR itself is decorated with the register names that replace the respective variable (and temporary) names. Next you will perform instruction selection as the next step towards generating working code

Part 2: Instruction Selection and Code Generation

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

The instruction selection is a matter of converting the Tiger IR code you've generated to the appropriate MIPS assembly code. The IR code format given to you 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. Allocate necessary storage via MIPS .data directives and generate necessary instructions for each IR statement. 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 begins. You must also generate a final instruction, "jr $ra", to return to the caller of your program. Supplied examples will help demonstrate this.

- You can first study the MIPS assembly and its SPIM implementation through http://www2.engr.arizona.edu/~ece369/Resources/spim/QtSPIM_examples.pdfand http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm
- You can then devise code generation (translation of an IR instruction into a set of MIPS instruction) – first start with ALU operations and load/stores and then go to branch and control flow finally do the logical operators.
- When you do the above step, work with small Tiger programs and their IR to check – assignment statements, then control flow and then looping etc. Finally do arrays and structures
- Design in a way to get the register annotation reflected on the IR by first doing register allocation and then doing the code generation on the annotated IR by adding necessary load/stores (in case the operands are in memory) or using the assigned registers (if they are held in registers).

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.

Code quality Comparisons

We are going to compare the code quality using the naïve register allocation to that generated with Global Register allocator. The MIPS simulator will allow you to generate dynamic instruction counts which you will compare as one of the measures of the code quality. A second measure will be the static size of the MIPS assembly generated for each of the above. Please document and analyze these comparisons in your report on given benchmarks.

Turn-in

Please turn in the following:

1, Complete source code of your project including make files and full build instructions

2. An executable of the compiler

3. Generated MIPS code using the test input IR provided to you (this code should load and execute on the supplied SPIM simulator

4. A small report that compares code quality (dynamic instruction counts and code sizes) across all the two schemes : naïve, and whole function register allocator.

Grading

1. Register allocation code (50 points)
   a. Naive (15 points)
   b. Whole function Register Allocation (35 points)

2. Instruction selection and code generation (25 points)
3. Passes tests using generated code executing on simulator (15 points)
   Report (design internals, how to build, run, code quality comparisons etc.) (10 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

The TA will supply a MIPS simulator that allows generating dynamic instruction counts for code quality comparison

Appendix A : Intermediate Representation

For the purposes of this project, we will be using 4-address code (or "quadruple" 3-address), which has the following form:

```
op, y, z, x
```

This reads as, "Do operation op to the values y and z, and assign the new value to x." A simple example of this can be given with the following: 2 * a + (b - 3)

The IR code for this expression would be following:

```
sub, b, 3, t1

mult, a,2,t2

 add, t1, t2, t3
```

Each function will begin with its signature, which is an IR statement denoting the return type, name of the function, and function parameters. The function signature is followed by "data segment " consisting of the classification of all the variables (including temporaries) inside the function based on the type. In particular, the data segment consists of a list of variables belonging to integer type and float type. For example, the IR header of a function named Foo(), which intakes an integer and a float is as follows:

```
int  Foo ( int first, float  second)

int-list : t1, t2 , t3, a , b

float-list : t4, t3, t6
```

Lastly, your intermediate code will have one exception to the 4-address structure. For instructions for function calls, you will generate an instruction very similar to that in the source. Function calls with no return values will look like the following:

```
            call, func_name, param1, param2, …, paramn
```

And function calls with return values will have a similar structure:

```
            callr, x, func_name, param1, param2, …, paramn
```

An example IR is as follows:

```
void main()
int-list:  A0, B0, C0, D0, E0, var0, var1, var2, var3, var4, var5
float-list:
    assign, E0, 0,
    assign, D0, 0,
    assign, C0, 0,
    assign, B0, 0,
    assign, A0, 0,
    goto, main0
main0:
    add, A0, 0, var0
    assign, A0, var0,
    add, B0, 1, var1
    assign, B0, var1,
    add, C0, 2, var2
    assign, C0, var2,
    add, D0, 3, var3
    assign, D0, var3,
    add, E0, 4, var4
    assign, E0, var4,
    assign, var5, 0,
    breq, B0, 0, if_label0
    assign, var5, 1,
if_label0:
    breq, var5, 0, if_label1
    assign, A0, 0,
    assign, B0, 1,
    assign, C0, 2,
    goto, if_label2, ,
if_label1:
    assign, D0, 3,
    assign, E0, 4,
if_label2:
    call, printi, A0
    call, printi, B0
    call, printi, C0
    call, printi, D0
    call, printi, E0
    return, , ,
```

Note that printi is a library call that has to be implemented by default in MIPS code (https://en.wikipedia.org/wiki/MIPS_instruction_set). SPIM provides some of the system call services as shown in this link (http://students.cs.tamu.edu/tanzir/csce350/reference/syscalls.html). Please use them to implement printi call.

Please download SPIM, either from SPIM site or from https://github.com/ostrichjockey/spim-keepstats.

The github version outputs some of the statistics onto your console. You can also look at it in the gui version of the SPIM site simulator.

Instructions on installation:

1) clone the code base using the following command:

git clone

https://github.com/ostrichjockey/spim-keepstats

2) change to SPIM directory:

cd spim-keepstats/spim

3) build and install spim using the following command:

make

sudo make install

4) test your installation:

spim -keepstats -f ../helloworld.s

[sample output]

```
Hello WorldStats -- #instructions : 13

#reads : 2  #writes 0  #branches 2  #other 9
```

Note:

With -keepstats flag, you are able to get information on the total number of instructions (#instructions), the number of load instructions (#reads), the number of store instructions (#writes), the number of jump instructions (#branches), and the number of the other instructions (#other).

Please report those statistics in your final report.

The appendix below consists of all the instructions supported in the IR.

Appendix B: IR for Tiger Language

Assignment: (op, x, y,_)

Op	Example source	Example IR
assign	a := b	assign, a, b,

Binary operation: (op, y, z, x)

Op	Example source	Example IR
add	a + b	add, a, b, t1
sub	a - b	sub, a, b, t1
mult	a * b	mult, a, b, t1
div	a / b	div, a, b, t1
and	a & b	and, a, b, t1

or	a | b	or, a, b, t1

Goto: (op, label, _, _)

Op	Example source	Example IR
goto	break;	goto, after_loop, ,

Branch: (op, y, z, label)

Op	Example source	Example IR
breq	if(a <> b) then	breq, a, b, after_if_part
brneq	if(a = b) then	brneq, a, b, after_if_part
brlt	if(a >= b) then	brlt, a, b, after_if_part
brgt	if(a <= b) then	brgt, a, b, after_if_part
brgeq	if(a < b) then	brgeq, a, b, after_if_part
brleq	if(a > b) then	brleq, a, b, after_if_part

Return: (op, x, _, _)

Op	Example source	Example IR
return	return a;	return, a, ,

Function call (no return value): (op, func_name, param1, param2, …, paramn)

Op	Example source	Example IR
call	foo(x);	call, foo, x

Function call (with return value): (op, x, func_name, param1, param2, …, paramn)

Op	Example source	Example IR
callr	a := foo(x, y, z);	callr, a, foo, x, y, z

Store into array: (op, array_name, offset, x)

Op	Example source	Example IR
array_store	arr[0] := a	array_store, arr, 0, a

Load from array: (op, x, array_name, offset)

Op	Example source	Example IR
array_load	a := arr[0]	array_load, a, arr, 0

Array Assignment: (op, x, size, value)

Op	Example source	Example IR
assign	var X : ArrayInt := 10; /* ArrayInt is an int array of size 100 */	assign, X, 100, 10