Tiger to RISC V Compiler

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Sourabh Aggarwal
(111601025)

under the guidance of

Dr. Piyush P. Kurur

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Contents

Chapter 1

Introduction

During 6th Semester, I wrote a compiler to compile Tiger to MIPS. Now is my attempt to build upon this compiler, to improve its functionality, efficiency and fix various issues/bugs. Also now instead of MIPS, I'll be compiling to RISC V.

1.1 Issues to resolve

- Currently the number of arguments that can be given to a function is exactly the number of available argument registers. This is to be fixed to support any number of function arguments.
- Register allocation currently works only on saved registers and temporaries, this can be augmented to use free argument registers as well. (Subjective move)
- Register allocation algorithm has to be implemented with better heuristics.
- Basic blocks has to optimized with optimizations like constant propagation, constant folding, etc.
- Error messages has to improved in semantic phase.
- To implement additional arithmetic operations like left/right shift.
- String comparison has to be made as simple as ”str1 > str2”, etc., instead of calling the string comparison functions to determine it.
- To implement ability to include pre-written code (header) files.
- To implement garbage collection.
- Add support for compile time (initial) arguments.
- And much more is possible, its like in a product.

1.2 Goal for this semester

For this semester, my main focus is on translating everything from MIPS to RISC V (along with removing redundancy and refactoring of code) and if there is time then I'll work on the issues mentioned above. Translating to RISC V is taking time as I am doing three things simultaneously, viz., Recollecting what I did previously, simultaneously refactoring and converting those files to RISC V and writing a good documentation for it so that any new comer can easily grasp it with minimum prerequisites.

1.3 Organization of The Report

The code written for this compiler is enormous and this report gives documentation on each of these files in an understandable order. But before proceeding it is important to read about Tiger language from this book (read Tiger Reference Manual in Appendix).

Chapter 2

Introduction to RISC V and runtime.s

2.1 Choosing a Simulator

In case of MIPS, it was straight forward to compile and run the assembly code but in case of RISC V, I had to struggle for some time as I couldn't find an appropriate documentation for it. After many iterations, I found the most suitable simulator, namely, Rars.

Please see the corresponding description of system calls here. It as well have a nice companion documentation which I expect one to read before proceeding further with the report.

There are other ways to compile and run RISC V code:

- Venus, Github repo. Note that for system calls, their argument register is different, see this.
- `spike` (sort of an official simulator), installed when using `riscv-gnu-toolchain`. Note it was as well required to install pk. System calls are different than RARS, basically it follows linux system calls. Can see these system calls here1 and here2. And linux system calls here, note that system calls of interest can be concisely seen here.

  So to compile and run the program, do: (Don't know if this is the intended way but after a lot of trial and error, I found this)

  ```
  riscv64-unknown-elf-as -o filename.o filename.s
  riscv64-unknown-elf-ld -o filename filename.o
  spike pk filename
  ```

2.2 Basic Examples

Few things to keep in mind:

- Note that dont use `$reg`, instead simply use `reg`.

- Always have two sections, one for data and another for text.
- During `ecall` all registers besides the output are guaranteed not to change.
- Put return value in `a0`.
- When we want our register value to be saved across system call, we can simply put it in stack instead of using registers s_i as anyway we would have to save their value first in stack so no change in overhead.
- `a7` is used to tell which system call.
- Save `ra` register if executing `jal` inside a function.

Hello World

```
.data # Tell the assembler we are defining data not code

msg: # Label this position in memory so it can be referred to in our code
  .string "hello world" # Copy the string "hello world" into memory

.text # Tell the assembler that we are writing code (text) now

start:
  li a7, 4 # li means to Load Immediate and we want to load the value 4
  into register a7
  la a0, msg  # la is similar to li, but works for loading addresses
  ecall
  li a7, 10  # Exit call
  ecall
```

To get the same code working using spike.

```
.globl _start # We must need to give _start, .globl helps to see it
outside this file
.data
str:
  .string "Hello World!\n"

.text
_start:

  li a0, 1
  la a1, str
  li a2, 13  # length of the string as required for linux system call. We
  can write a function which will determine the length of the string by
  checking for terminating null character.
  li a7, 64
  ecall

  li a0, 0   # The exit code we will be returning is 0
  li a7, 93  # Again we need to indicate what system call we are making
  and this time we are calling exit(93)
  ecall
```

Saving callee save registers

```
.data

  bef: .string "Before modification, value is: "
  dur: .string "\nInside function, value is: "
  aft: .string "\nAfter function call, value is: "

.text

main:
  addi s0, zero, 1
  # Print bef
  la a0, bef
  li a7, 4
  ecall
  # Print int
  li a7, 1
  mv a0, s0
  ecall
  jal increment
  # Print aft
  la a0, aft
  li a7, 4
  ecall
  # Print int
  li a7, 1
  mv a0, s0
  ecall
  # Exit
  li a7, 10
  ecall



increment:
  addi sp, sp, -4
  sw s0, 0(sp) # '0' denotes the offset, in case of 0, we can simply omit
  it.
  addi s0, s0, 1
  # Print string
  la a0, dur
  li a7, 4
  ecall
  # Print the incremented integer
  mv a0, s0
  li a7, 1
  ecall
  lw s0, 0(sp)
  addi sp, sp, 4
  jr ra
```

2.3 runtime.s

There are some standard functions which our Tiger program can use. They are written in `runtime.s`. I could have as well used file provided by author but as I want to augment it further and implement things my way, I decided to write runtime myself. Each of the functions of runtime is explained below along with code for easy understanding.

```
.data

__exitMessage: .string "Exited with code: "
__newLine: .string "\n"

.text

# Many of the below written functions assume that the given input is
correct.

# Given the exit code (in a0 ofc), terminate with that exit code.
exit:
    mv t0, a0
    # Print exit message
    la a0, __exitMessage
    li a7, 4
    ecall
    # Print code
    mv a0, t0
    li a7, 1
    ecall
    # Print new line
    la a0, __newLine
    li a7, 4
    ecall
    # Exit
    li a7, 10
    ecall

# Not of non zero integer is 0 whereas not of 0 is 1.
not:
    beqz a0, retOne
    li a0, 0
    jr ra
    retOne:
        li a0, 1
        jr ra

# Given the string s in a0, return the number of characters in it.
# This is aswell needed for string concatenation.
size:
    mv t0, a0
    mv a0, zero
    sizeLoop:
        lb t1, (t0)
        beqz t1, sizeExit
        addi a0, a0, 1
        addi t0, t0, 1
        j sizeLoop
    sizeExit:
        jr ra

# Copy the string completely (i.e. including zero / null character) whose
address is at a1, to the address starting at a0, returning the address of
the last character of copied string.
stringCopy:
    stringCopyLoop:
        lb t0, (a1)
        sb t0, (a0)
        beqz t0, stringCopyExit
        addi a0, a0, 1
        addi a1, a1, 1
        j stringCopyLoop
    stringCopyExit:
        jr ra

# Concatenate str1 present in a0 with str2 present in a1.
```

```
concat:
    addi sp, sp, -12
    sw a0, (sp)
    sw a1, 4(sp)
    sw ra, 8(sp)
    jal size
    li t0, 1  # Will contain len(str1) + len(str2) + 1. '+1' for null
    character.
    add t0, t0, a0
    addi sp, sp, -4
    sw t0, (sp)
    lw a0, 8(sp)  # offset is changed
    jal size
    lw t0, (sp)
    add t0, t0, a0
    addi sp, sp, 4
    mv a0, t0
    li a7, 9
    ecall
    addi sp, sp, -4
    sw a0, (sp)
    lw a1, 4(sp)
    jal stringCopy
    lw a1, 8(sp)
    jal stringCopy
    lw a0, (sp)
    lw ra, 12(sp)
    addi sp, sp, 16
    jr ra

# "function substring (s: string, first : int, n : int) : string"
Substring of string s, starting with character first, n characters long.
# Hoping that given input is valid.
substring:
    # Allocate space
    mv a3, a0  # saving a0
    mv a0, a2
    addi a0, a0, 1  # for null character
    li a7, 9
    ecall
    # making a3 point to the desired substring
    add a3, a3, a1
    mv t0, a0  # we need to return this
    substringLoop:
        lb t1, (a3)
        sb t1, (a0)
        beqz a2, substringExit
        addi a0, a0, 1
        addi a3, a3, 1
        addi a2, a2, -1
        j substringLoop
    substringExit:
        mv a0, t0
        jr ra

# str1 > str2 ?
stringGreat:
    stringGreatLoop:
        lb a2 (a0)
        lb a3 (a1)
        bgt a2, a3  stringGreatA
        blt a2, a3  stringGreatB
        # If we have reached this point that means both are equal and if
        one of them is zero that means other is aswell 0, so in case
        strings are equal, I must return 0.
        beqz a2, stringGreatB
        addi a0, a0, 1
        addi a1, a1, 1
        j stringGreatLoop
    stringGreatA:
        li a0, 1
        jr ra
    stringGreatB:
        li a0, 0
        jr ra

# str1 < str2 ?
stringLess:
    stringLessLoop:
        lb a2 (a0)
        lb a3 (a1)
        blt a2, a3  stringLessA
        bgt a2, a3  stringLessB
        # If we have reached this point that means both are equal and if
        one of them is zero that means other is aswell 0, so in case
        strings are equal, I must return 0.
        beqz a2, stringLessB
        addi a0, a0, 1
        addi a1, a1, 1
        j stringLessLoop
    stringLessA:
        li a0, 1
        jr ra
    stringLessB:
        li a0, 0
        jr ra

# str1 == str2 ?
stringEqual:
    addi sp, sp, -12
    sw a0, (sp)
    sw a1, 4(sp)
    sw ra, 8(sp)
    jal stringGreat
    bnez a0, stringEqualExit
    lw a0, (sp)
    lw a1, 4(sp)
    jal stringLess
    bnez a0, stringEqualExit
    li a0, 1
    lw ra, 8(sp)
    addi sp, sp, 12
    jr ra
    stringEqualExit:
        li a0, 0
        lw ra, 8(sp)
        addi sp, sp, 12
        jr ra


# Single-character string from ASCII value given in a0; halt program if
a0 out of range.
chr:
    # Handling the error part
    addi t0, zero, 127
    bgt a0, t0, chrError
    bltz a0, chrError
    # Allocating
    mv t0, a0
    li a0, 2
    li a7, 9
    ecall
    # Putting the character
    sb t0 (a0)
    sb zero 1(a0)
    jr ra
    chrError:
        addi a0, zero, -1
        j exit

# Given a string in a0, return ASCII value of the first character of it,
return -1 if the string is empty.
ord:
    lb t0, (a0)
    beqz t0, ordEmpty
    mv a0, t0
    jr ra
    ordEmpty:
        li a0, -1
        jr ra

# Read a character from standard input and return it as a string; return
empty string on end of file.
getchar:
    # Allocate space
    li a0, 2
    li a7, 9
    ecall
    sb zero, 1(a0)  # Null character
    # Read the character
    mv t0, a0
    li a7, 12
    ecall
    sb a0, (t0)  # Store the character
    mv a0, t0
    jr ra

# Absolete as of now
flush:
    jr ra

# Print the string whose address is in a0
print:
    li a7, 4
    ecall
    jr ra
```

```
# printInt:
#     # Examples in book do complex computation to print an integer, here
I am putting an inbuilt function
#     # Print the integer in a0
#     li a7, 1
#     ecall
#     jr ra

# a0 contains the number of bytes we need to allocate. So, multiply it by
4 and allocate that much space from heap (system call 9). Return value is
in a0 which tells the address to the allocated block (lower address
value) and remember that in going downwards address decreases. Rest of
the code is easy to follow. Note that a1 contains the value to which we
need to initialize our array.
initArray:
  li t0, 4
  mul a0, a0, t0
    mv t1, a0
  li a7, 9
  ecall
  mv t0, a0
  add t1, t1, t0
  initArrayLoop:
        sw a1, (t0)
        addi t0, t0, 4
        beq t0, t1, initArrayExit
        j initArrayLoop
    initArrayExit:
        jr ra

# Very similar to initArray
# We just need to allocate memory, no need to initialize it with 0.
allocRecord:
    li t0, 4
    mul a0, a0, t0
    li a7, 9
    ecall
    jr ra
```

Chapter 3

Phase I : Constructing Abstract Syntax Tree

In this phase, files of interest are:-

- parse.sml
- errormsg.sml
- tiger.lex
- table.sml, table.sig
- symbol.sml
- tiger.grm, absyn.sml

Each of these is explained below.

3.1 parse.sml

We know the compiler follows "Lexical Analysis → Parsing → . . . ". `parse.sml` is the one which does these two and ties the various files associated with it and finally returns the desired abstract syntax tree.

3.2 errormsg.sml

Besides errors encountered in lexical analysis phase, they'll be print using `errormsg.sml` which will print these errors in an elaborate manner.

I'll be explaining the signature of this file here, which is sufficient to understand the working of this file.

```
(* In our lexer, each line will occur at a particular "pos" which is
incremented for each character ("pos" is nothing but the number of
characters read till now). For each line, we maintain its "pos". Our goal
is that given "pos", we have to determine the line number and column
number, this can now be easily done, just determine that line with
maximum "pos" which is less than the given "pos", the difference now
gives the column number. Reason of doing this in such an odd way is
because in our grammar, lexer and in abstract syntax file we define "pos"
to be int and not as a tuple (int, int) where first parameter could
denote line number and other one as column number. *)

signature ERRORMSG =
sig
    val anyErrors : bool ref  (* has their been an occurance of any error?
    *)
    val fileName : string ref (* Updated in parse.sml, used when printing
    errors *)
    val lineNum : int ref (* Number of lines in the read file *)
    val linePos : int list ref (* as defined in the top most comment, it
    is the list containing value of "pos" at each line *)
    val error : int -> string -> unit (* it should take "pos" and error
    message to print, from "pos" it will determine the line number and
    the column number, and will print abstractly
    "filename:lineNo.ColNo:Message". *)
    exception Error
    val impossible : string -> 'a   (* raises Error, for a behavior we
    didn't expect *)
    val reset : unit -> unit  (* reset the parameters, so that we can move
    on to read new file *)
end
```

3.3 tiger.lex

Simple lexer for our language, just that for some of the errors like non terminated string, etc. are easily detected in this phase and thus are printed now with putting an end to the compilation phase. Usually one detects many errors and print them all instead of just printing one error and stopping. But I feel that errors in lexical phase, if found, are critical and should be handled first most.

Guidelines

Few important rules to keep in mind:

- Longest match: The longest initial substring of the input that can match any regular expression is taken as the next token.

- Rule priority: For a particular longest initial substring, the first regular expression that can match determines its token type. This means that the order of writing down the regular-expression rules has significance.

- An individual character stands for itself, except for the reserved characters

```
? * + | ( ) ^ / ; . = < > [ { " \  $
```

- A backslash followed by one of the reserved characters stands for that character.

- Inside the brackets, only the symbols

```
\ - ^
```

  are reserved. An initial up-arrow ^ stands for the complement of the characters listed, e.g. [^abc] stands any character except a, b, or c.

- To include ^ literally in a bracketed set, put it anywhere but first; to include - literally in a set, put it first or last.

- The dot . character stands for any character except newline, i.e. the same as

```
[^\n]
```

- The following special escape sequences are available, inside or outside of square brackets:

```
\b backspace
\n newline
\t horizontal tab
\ddd where ddd is a 3 digit decimal escape
```

- Any regular expression may be enclosed in parentheses ( ) for syntactic (but, as usual, not semantic) effect

- A sequence of characters will stand for itself (reserved characters will be taken literally) if it is enclosed in double quotes " ".

- A postfix repetition range {a, b} where a and b are small integers stands for any number of repetitions between a and b of the preceding expression. The notation {a} stands for exactly a repetitions. Ex: [0-9]{3} Any three-digit decimal number.

- The rules should match all possible input. If some input occurs that does not match any rule, the lexer created by ML-Lex will raise an exception LexError.

3.4 table.sml, table.sig

For our immediate steps, we would need a way to map our "keys" to a particular "value", for this we create a functor, "IntMapTable" which takes the type of "key" and a function to get an integer corresponding to that "key". Then this functor uses "IntBinaryMap", a hash map to represent our map. This "IntBinaryMap" is polymorphic, so we can use it for any value corresponding to our int (which we obtained from our key). This is the purpose served by "table.sig" and "table.sml".

3.5 symbol.sml

Each of the variable, function declaration etc. will be stored as "symbols" (a pair of (string, int)). Where each symbol will have a unique integer. We can then map this integer to anything using our "IntBinaryMap". Purpose of keeping this symbols is that we would like to see whether this variable is declared before or not, whether this type exists or not, etc.

3.6 tiger.grm, absyn.sml

`tiger.grm` is the parser for our language, it will read the tokens from lexer and build AST using `absyn.sml`. Grammar directly follows the rules given in the book.

Guidelines

- Format of ML-YACC: user declarations %% parser declarations %% grammar rules.
- By default, ML-YACC resolves shift-reduce conflicts by shifting, and reduce-reduce conflicts by using the rule that appears earlier in the grammar. If then (else) shift-reduce conflict is thus not harmful.
- Consider for example:

  ```
  E -> E * E.   (+)
  E -> E. + E   (any)
  ```

  The precedence declarations (%left, etc.) give priorities to the tokens; the priority of a rule is given by the last token occurring on the right-hand side of that rule. Thus the choice here is between a rule with priority "*" and a token with priority "+"; the rule has higher priority, so the conflict is resolved in favor of reducing.
- When the rule and token have equal priority, then a %left precedence favors reducing, %right favors shifting, and %nonassoc yields an error action.
- Instead of using the default "rule has precedence of its last token," we can assign a specific precedence to a rule using the %prec directive. This is commonly used to solve the "unary minus" problem. In most programming languages a unary minus

binds tighter than any binary operator, so —6 * 8 is parsed as (—6) * 8, not —(6 * 8).

- Precedence declaration should be written in the order of increasing precedence.

For more description, please see the comments in the file `tiger.grm`.

Chapter 4

Phase II : Constructing Intermediate Representation Tree

Now that we have constructed our Abstract Syntax Tree, we will now have to convert it to an Intermediate Representation to solve $m \times n$ problem.

In this phase, files of interest are:-

- types.sml
- env.sml, env.sig
- temp.sml, temp.sig
- findescape.sml
- tree.sml
- risc.sml
- translate.sml
- semant.sml

Each of these is explained below.

4.1 types.sml

Our next step is to do type checking of our input AST. For this purpose we need to define auxiliary types. Thus, we have "types.sml" representing the same. Thus now we can easily map a "symbol" to its corresponding type.

4.2 env.sml, env.sig, temp.sml, temp.sig

Now for type checking, we need some environment under which we have to check compatibility of types as for instance, environment can be different for different functions as within let block of each function there can be new declarations etc. By default we have some standard types and functions already present in our environment. Now each of these functions will correspond to some label in our MIPS code, for which we have "temp.sml". Note that "temp.sml" is also used to represent our infinite register in Intermediate Representation.

4.3 findescape.sml

It would be important for us to determine which variable cannot go to a register, i.e., which variable escapes. Idea of finding escapes is straight forward: Just see if this thing was defined in outer scope, note that scope extends only when we are calling a function.

4.4 tree.sml, tree.sig

It is a datatype representing our Intermediate Tree.

```
  datatype stm = SEQ of stm * stm (* The statement s2 followed by s2. *)
  | LABEL of label (* Define the constant value of name n to be the
  current machine code address. This is like a label definition in
  assembly language. *)
  | JUMP of exp * label list (* Transfer control (jump) to address exp.
  The destination exp may be a literal label, as in NAME(lab), or it may
  be an address calculated by any other kind of expression. For example,
  a C-language switch (i) statement may be implemented by doing
  arithmetic on i. The list of labels labs specifies all the possible
  locations that the expression exp can evaluate to; this is necessary
  for dataflow analysis later. The common case of jumping to a known
  label "l" is written as jump(name l, [l]). *)
  | CJUMP of relop * exp * exp * label * label (* CJUMP(o, e1, e2, t, f):
  Evaluate e1, e2 in that order, yielding values a, b. Then compare a, b
  using the relational operator o. If the result is true, jump to t
  otherwise jump to f. The relational operators *)
  | MOVE of exp * exp
  (*
     MOVE(TEMP t, e): Evaluate e and move it into temporary t.
     MOVE(MEM(e1), e2) Evaluate e1 yielding address a. Then evaluate e2
and store the result into wordSize bytes of memory starting at a.
  *)
  | EXP of exp (* Evaluate exp and discard the result. *)

and exp = BINOP of binop * exp * exp (* The application of binary
operators to operands exp1, exp2. *)
```

```
  | MEM of exp (* The contents of wordSize bytes of memory starting at
  address exp (where wordSize is defined in the Frame module). Note that
  when MEM is used as the left child of a move, it means "store," but
  anywhere else it means "fetch." *)
  | TEMP of Temp.temp (* Temporary t. A temporary in the abstract machine
  is similar to a register in a real machine. However, the abstract
  machine has an infinite number of temporaries. *)
  | ESEQ of stm * exp (* The statement s is evaluated for side effects,
  then e is evaluated for result *)
  | NAME of label (* The value NAME(n) may be the target of jumps, calls,
  etc. *)
  | CONST of int (* The integer constant int. *)
  | CALL of exp * exp list (* A procedure call: the application of
  function exp1 to argument list exp2 list. The subexpression exp1 is
  evaluated before the arguments which are evaluated left to right. *)

and binop = PLUS | MINUS | MUL | DIV | AND | OR | LSHIFT | RSHIFT |
ARSHIFT | XOR

and relop = EQ | NE | LT | GT | LE | GE | ULT | ULE | UGT | UGE
```

risc.sml

Here we will define an interface for RISC V, which finds some use in this phase, but is mainly there for code generation phase. Its applications and purpose will become clear by seeing the below written code. Before proceeding please read and understand about **static links** given at page 132-134. Basic gist is that in languages that allow nested function declarations, the inner functions may use variables declared in outer functions. One way to accomplish this is that whenever a function *f* is called, it can be passed a pointer to the frame of the function statically enclosing *f*; this pointer is the static link.

```
structure RiscFrame: FRAME = struct

structure T = Temp
structure Tr = Tree


(* in register or in frame? i.e. whether are we putting it in register or
in stack? *)
datatype access = InFrame of int | InReg of T.temp

(* numLocals: How many variables got called by allocLocal which were put
in our stack (i.e. not in register but in Frame) *)
(* name: Name of the frame (function), access list: tells for each formal
parameter whether it is in Frame or in reg. *)
```

```
type frame = {name : T.label, formals : access list, numLocals : int ref,
shiftInstr : Tr.stm list}

type register = string  (* so type of our register list will be "string
list", register list is useful to tell available registers *)

(*
  Given a Tiger function definition comprising a level and an
already-translated body expression, the Translate phase should produce a
descriptor for the function containing this necessary information:
    1. frame: The frame descriptor containing machine-specific
information about local variables and parameters.
    2. body: The result returned from procEntryExit1 (defined in detail
below). Call this pair a fragment to be translated to assembly language.
*)
(* A string literal in the Tiger (or C) language is the constant address
of a segment of memory initialized to the proper characters. *)
(* For each string literal "lit", the Translate module makes a new label
"lab", and returns the tree "Tree.name (lab)". It also puts the
assembly-language fragment "Frame.STRING (lab, lit)"" onto a global list
of such fragments to be handed to the code emitter. *)
(* All this would be handled in putting it all together phase (chapter 12
i.e. in main.sml) *)
datatype frag = PROC of {body: Tree.stm, frame: frame}
              | STRING of T.label * string

(* ----------------CPU Registers---------------------- *)

val zero = T.newtemp() (* constant 0 *)
val ra = T.newtemp() (* return address *)
val sp = T.newtemp() (* stack pointer *)

(* arguments *)
val a0 = T.newtemp()
val a1 = T.newtemp()
val a2 = T.newtemp()
val a3 = T.newtemp()
val a4 = T.newtemp()
val a5 = T.newtemp()
val a6 = T.newtemp()
val a7 = T.newtemp()

val rv = a0
```

```
(* temporary *)
val t0 = T.newtemp()
val t1 = T.newtemp()
val t2 = T.newtemp()
val t3 = T.newtemp()
val t4 = T.newtemp()
val t5 = T.newtemp()
val t6 = T.newtemp()

(* saved temporary *)
val s0 = T.newtemp() (* s0 = fp *)
val s1 = T.newtemp()
val s2 = T.newtemp()
val s3 = T.newtemp()
val s4 = T.newtemp()
val s5 = T.newtemp()
val s6 = T.newtemp()
val s7 = T.newtemp()
val s8 = T.newtemp()
val s9 = T.newtemp()
val s10 = T.newtemp()
val s11 = T.newtemp()

val fp = s0

val specialregs = [(zero, "zero"), (sp, "sp"), (ra, "ra")]

val argRegs = [(a0, "a0"), (a1, "a1"), (a2, "a2"), (a3, "a3"), (a4, "a4"),
(a5, "a5"), (a6, "a6"), (a7, "a7")]

val savedRegs = [(s0, "s0"), (s1, "s1"), (s2, "s2"), (s3, "s3"), (s4,
"s4"), (s5, "s5"), (s6, "s6"), (s7, "s7"), (s8, "s8"), (s9, "s9"), (s10,
"s10"), (s11, "s11")]

val temporaries = [(t0, "t0"), (t1, "t1"), (t2, "t2"), (t3, "t3"), (t4,
"t4"), (t5, "t5"), (t6, "t6"), (t7, "t7")]

(* Supporting only traditional a0 - a7 arguments, i.e. maximum of 8
arguments. *)
exception ArgExceed of string

val wordSize = 4

(* registers allocated for arguments in risc *)
```

```
val argRegsCount = List.length argRegs

fun getFirstL (ls) = (map (fn (x, y) => x) ls)
fun getSecondL (ls) = (map (fn (x, y) => y) ls)

(* Getting all the registers available for coloring *)
val registers = getSecondL(savedRegs @ temporaries)

(* tempMap is a table from registers (not all registers but some) to
their name *)
(* basically its use is for new temporaries, so that we can assign
register to them as well *)
val tempMap =
let
  fun addToTable ((t, s), table) = T.Table.enter(table, t, s)
  val toAdd = specialregs @ argRegs @ savedRegs @ temporaries
in
  foldl addToTable T.Table.empty toAdd
end

(* Helper Functions *)
fun incrementNumLocals ({numLocals, ...} : frame) = numLocals :=
!numLocals + 1
fun getOffset ({numLocals, ...} : frame) = !numLocals * (~wordSize)
fun getFOffset (frame' : frame) = ~(getOffset (frame')) + 8
fun name ({name, ...} : frame) = name
fun formals ({formals, ...} : frame) = formals

(* To generate label for string *)
fun genString (lab, s) = Symbol.name lab ^ ": .string \"" ^ s ^ "\"\n"

(* The function Frame.exp is used by Translate to turn a Frame.access
into the Tree expression. The Tree.exp argument to Frame.exp is the
address of the stack frame that the access lives in. Thus, for an access
"a" such as InFrame(k), we have Frame.exp (a) (TEMP(Frame.FP)) =
MEM(BINOP(PLUS, TEMP(Frame.FP), CONST(k))). Why bother to pass the tree
expression temp (Frame.FP) as an argument? The answer is that the address
of the frame is the same as the current frame pointer only when accessing
the variable from its own level. When accessing "a" from an inner-nested
function, the frame address must be calculated using static links, and
the result of this calculation will be the Tree.exp argument to
Frame.exp. If "a" is a register access such as InReg(t932) then the
frame-address  argument to Frame.exp will be discarded, and the result
will be simply TEMP t932. *)
```

```
fun exp frameAccess frameAddress =
  case frameAccess of
      InFrame offset => Tr.MEM(Tr.BINOP(Tr.PLUS, frameAddress, Tr.CONST
      offset))
    | InReg temp => Tr.TEMP(temp)


(* Given the name of the frame and list of variables mentioned in the
format of whether they escape or not, function returns the new frame *)
(* Above comment is sufficient but can look at page 142 *)
fun newFrame {name : T.label, formals : bool list} : frame =
let
  fun allocFormals(offset, [], allocList) = allocList
    | allocFormals(offset, curFormal::l, allocList) =
      (
      case curFormal of
          true => allocFormals(offset + wordSize, l, allocList @ [InFrame
          offset])
        | false => allocFormals(offset, l, allocList @
        [InReg(T.newtemp())])
      )
  val aformals = allocFormals (wordSize, formals, []) (* first word is
  reserved for something *)
  fun viewShift (acc, reg) = Tr.MOVE(exp acc (Tr.TEMP fp), Tr.TEMP reg)
  (* getting the value in argRegs to their correct positions *)
  val shiftInstr = ListPair.map viewShift (aformals, getFirstL (argRegs))
in
  if (List.length formals <= List.length argRegs) then
    {name = name, formals = aformals, numLocals = ref 0, shiftInstr =
    shiftInstr}
  else
    raise ArgExceed ("No. of arguments exceeded!")
end

(* allocating a variable for our frame *)
fun allocLocal frame' escape = (
  case escape of
      true => (incrementNumLocals frame'; InFrame(getOffset frame'))
    | false => InReg(T.newtemp())
)
```

```
(* Calling runtime-system functions. To call an external function named
initArray with arguments a, b, simply generate a CALL such as
CALL(NAME(Temp.namedlabel ( "initArray" )), [a, b]) This refers to an
external function initArray which is written in a language such as C or
assembly language - it cannot be written in Tiger because Tiger has no
mechanism for manipulating raw memory. But on some operating systems, the
C compiler puts an underscore at the beginning of each label; and the
calling conventions for C functions may differ from those of Tiger
functions; and C functions don't expect to receive a static link, and so
on. All these target-machine-specific details should be encapsulated into
a function provided by the Frame structure. where externalCall takes the
name of the external procedure and the  arguments to be passed. *)
fun externalCall (s, args) = Tr.CALL(Tr.NAME(T.namedlabel s), args)

(* needed as we are going to add new tree instructions *)
fun seq nil = Tr.EXP (Tr.CONST 0)
  | seq [st] = st
  | seq (st :: rest) = Tr.SEQ(st, seq (rest))

(*
  Each Tiger function is translated into a segment of assembly language
with a prologue, a body, and an epilogue.

  The body of a Tiger function is an expression, and the body of the
translation is simply the translation of that expression.

  The prologue, which precedes the body in the assembly-language version
of the function, contains:
    1. Pseudo-instructions, as needed in the particular assembly
language, to
    announce the beginning of a function. (Not relevant in our context)
    2. A label definition for the function name. (Done in procEntryExit3)
    3. An instruction to adjust the stack pointer (to allocate a new
frame). (Done in procEntryExit3)
    4. Instructions to save "escaping" arguments - including the static
link - into the
    frame, and to move nonescaping arguments into fresh temporary
registers. (Done in procEntryExit1)
    5. Store instructions to save any callee-save registers - including
the return
    address register - used within the function. (Done in procEntryExit1)

   (6) The function body.
  The epilogue comes after the body and contains
    7. An instruction to move the return value (result of the function)
to the register reserved for that purpose. (Already added in body by
translate)
    8. Load instructions to restore the callee-save registers. (Done in
procEntryExit1)
    9. An instruction to reset the stack pointer (to deallocate the
frame). (Done in procEntryExit3)
    10. A return instruction (JUMP to the return address). (Done in
procEntryExit3)
    11. pseudo-instructions, as needed, to announce the end of a
function. (Not relevant in our context)

  Some of these items (1, 3, 9 and 11) depend on exact knowledge of the
frame size, which will not be known until after the register allocator
determines how many local variables need to be kept in the frame because
they don't fit in registers. So these instructions should be generated
very late, in a frame function called procEntryExit3.
*)

(* As mentioned in the above comment, it does what is known as "view
shift" *)
fun procEntryExit1(frame' as {shiftInstr, ...} : frame, body) =
let
  val pairs = map (fn reg => (allocLocal frame' false, reg)) ([ra] @
  getFirstL (savedRegs))
  val saves = map (fn (allocLoc, reg) => Tr.MOVE (exp allocLoc (Tr.TEMP
  fp), Tr.TEMP reg)) pairs
  val restores = map (fn (allocLoc, reg) => Tr.MOVE (Tr.TEMP reg, exp
  allocLoc (Tr.TEMP fp))) (List.rev pairs)
in
  seq(shiftInstr @ saves @ [body] @ restores)
end


(* This function appends a "sink" instruction to the function body to
tell the register allocator that certain registers are live at procedure
exit. Having zero live at the end means that it is live throughout, which
will prevent the register allocator from trying to use it for some other
purpose. The same trick works for any other special registers the machine
might have. *)
(* The following snippet was given in book, just modified src. *)
fun procEntryExit2(frame, body) =
        body @
        [Assem.OPER {assem = "",
                     src = getFirstL (specialregs @ savedRegs),
                     dst = [], jump = SOME[]}
        ]

fun procEntryExit3(frame' : frame, body) =
  {prolog = Symbol.name (name frame') ^ ":\n" ^
    (* fp -> 0(sp) *)
    "sw fp 0(sp)\n" ^
    (* sp -> fp *)
    "move fp sp\n" ^
    (* sp = sp -  (allocating space in stack) *)
    "addiu sp sp -" ^ Int.toString (getFOffset (frame')) ^ "\n",
    body = body,
            (* fp -> sp *)
    epilog = "move sp fp\n" ^
    (* lw Rdest, address      Load Word
Load the 32-bit quantity (word) at address into register Rdest. *)
    "lw fp 0(sp)\n" ^
    (* jr Rsource        Jump Register
Unconditionally jump to the instruction whose address is in register
Rsource.*)
    "jr ra\n"}
end
```

translate.sml

This file is used by `semant.sml` to generate IR tree code.

Its not reasonable to translate expression of our AST to an expression of IR as this is true only for certain kinds of expressions, the ones that compute a value. Expressions that return no value (such as some procedure calls, or while expressions in the Tiger language) are more naturally represented by `Tree.stm`. And expressions with Boolean values, such as $a > b$, might best be represented as a conditional jump - a combination of `Tree.stm` and a pair of destinations represented by `Temp.labels`. Therefore, we will make a datatype exp in the Translate module to model these three kinds of expressions:

```
datatype exp =  Ex of Tr.exp
              | Nx of Tr.stm
              | Cx of Temp.label * Temp.label -> Tr.stm
```

- **Ex** stands for an "expression," represented as a `Tree.exp`.
- **Nx** stands for "no result," represented as a Tree statement.
- **Cx** stands for "conditional," represented as a function from label-pair to statement. If you pass it a true-destination and a false-destination, it will make a statement that

evaluates some conditionals and then jumps to one of the destinations (the statement will never ”fall through”). For example, the Tiger expression

```
a > b | c < d
```

might translate to the conditional:

```
Cx(fn (t,f) => SEQ(CJUMP(GT, a, b, t, z),
SEQ(LABEL z, CJUMP (LT, c, d, t, f))))
```

for some new label `z`.

Sometimes we will have an expression of one kind and we will need to convert it to an equivalent expression of another kind. For example, the Tiger statement

```
flag := (a>b | c<d)
```

. This is achieved by `unEx`. Similarly we have `unNx` and `unCx`.

Similarly each function call defines a level which is encapsulated as

```
datatype level =  Top
                | Lev of {parent: level, frame: F.frame} * unit ref
```

`unit ref` is used to easily check for equality.

All of the functions of `translate.sml` are straight forward so please see and understand them.

4.5 semant.sml

After we receive our AST, we run semant's function `transProg` on it which will do type checking of our AST and simultaneously convert it into our IR. Both this and `translate.sml` are big so I have not put their code here, however please see linked code which is properly documented and thus is easy to understand.

Chapter 5

Conclusion and Future Work

As of now, my progress has been in understanding till Phase II, identifying and fixing few issues and doing suitable modifications to make it RISC V compatible. By the semester end I hope to have fully functional compiler possibly with additional features.

References

[1] Modern Compiler Implementation in ML.

[2] RARS wiki.

[3] User's Guide to ML-Lex and ML-Yacc.