# Contents

# Project Phase 3 Design Internals Report

Abhijit Singh asingh811@gatech.edu

June 2021

## 1 About this project

This report describes the implementation of "phase 3: Design Internals" of the compiler. This includes allocating registers for variables using various strategies and MIPS back end code generation.

## 2 Code hierarchy

The following define the main categories of classes, this program is split into.

A: Instruction: Objects with this base class map onto individual IR instructions based on the provided IR file and then translate them to mips assembly (upon being provided a mapping to a variable)

There exist various instruction derived classes:

1) GenericInst: an instruction that isnt defined in IR but is later defined by the back end (mostly for load and stores)

2) LabelInst: a label

3) AssignInst: integer assignment

4) AssignArrayToArrayInst : array to array assignment

5) InitialiseArrayConstInst: const to array assignment

6) AddInst, SubInst, MultInst, DivInst, AndInst, OrInst: Add, subtract, multiply and divide instructions respectively

7) GotoInst: A jump instruction

8) BreqInst, BrneqInst, BrgtInst, BrltInst, BrleqInst, BrgeqInst: branch to label conditions

9) ReturnProcedureInst, ReturnFunctionInst: return to caller function from fn/procedure

10) CallProcedureInst, CallFunctionInst: call a function or procedure

11) CallStoreArrayInst, CallLoadArrayInst: load or store at index of array

B: RegisterAllocator: These objects define the strategy for variable to register mapping. They organise the instructions and add additional ones if necessary
The following are types of allocators

1) NaiveAllocator: simply loads and stores vars every instruction. As one can imagine, these are highly inefficient
2) BlockAllocator: creates blocks from provided list of instructions. Maps which block points to which other block
3) BriggsAllocator: NOT FULLY IMPLEMENTED DUE TO TIME CONSTRAINTS

The following data storage class was used to help with code generation
A) Block: Defines and computes a block and a set of blocks as well as holds onto a global set of blocks. Also computes live ranges
B) IntList: list of defined variables in the current function or the static scope 3) Function: defines functions as sets of instructions and a locally defined intlist

The following class organises the core of the application:

FunctionReader: Performs the following duties

1) Reads and writes to and from ".ir" (Intermediate representation and ".s" (MIPS Assembly) files

2) Using the provided input file, generates various "Instruction" objects for each function

3) based on provided program args, generate one of the 3 types of allocators and write their instructions to file

---

# 3 Instruction

The static function parse(): parses provided line of IR code to derive an instruction associated with the string

An instruction inherited class primarily is required to implement the following functions:

The **constructor** (to parse variables that are part of the IR)
The **getInstructionType**
an enum that helps identify which child class this instruction is associated with

**getMIPSInstruction()**
Get mips instruction associated with this derived class

An example implementation for "AssignInstruction::getMIPSInstruction":

```
std::string getMIPSInstruction() override {
   std::ostringstream stringStream;
   if (isInteger(_vars[1])) {
      stringStream << "li " << _varRegMap[_vars[0]] << "," <<
          _vars[1];
   }
   else {
      stringStream << "move " << _varRegMap[_vars[0]] << "," <<
          _varRegMap[_vars[1]];

   }

   return stringStream.str() + "\n";
}
```

Listing 1: AssignInstruction::getMIPSInstruction

Here, based on whether the assignment is an integer or a var, we appropriately generate either a "li" call or a "move" call.

# 4 RegisterAllocator

Using a list of Functions (collections of instructions), this generates register mappings to be used by each of the instructions

## 4.1 NaiveAllocator

Instruction by instruction,
first it maps registers from var to mappedRegister calls

getLoadInstruction(variable, mappedRegister) to load var into mapped register from memory
Then calls
getMIPSInstruction() to get the assembly code Then calls
getStoreInstruction to store the variable in the mapped register to memory

The following is a sample assign array statement

```
int−list:a, b[1]
assign, b, a
```

It generates the following code

```
lw $s7,0($sp)
la $s6,28($sp)
lw $s7,0($s6)
sw $s7,0($sp)
```

## 4.2 BlockAllocator

Using the provided function list, it generates a set of blocks and a CFG

Blocks are led by and terminate just before labels and jump instructions. CFG was built by identifying which following blocks the code falls through into, plus which it branches into

It uses this information to load the most used variables at the start of the block, and store them at the end of the block
If it runs out of registers, it uses the Naive allocator's strategy on the remaining instructions of the block The following is how it loads and stores values in blocks

For the given "block" identified by this class

```
assign, a, 5,
        assign, b, 50,
        add, 100, a, _t0
        add, _t0, b, _t1
        assign, c, _t1,
        call, printi, c
```

It generates the following code

```
lw $s7,0($sp)
lw $s6,4($sp)
lw $s5,96($sp)
lw $s4,100($sp)
lw $s3,104($sp)
li $s5,5
```

```
li  $s4,50
addi $s7,$s5,100
add $s6,$s4,$s7
move $s3,$s6
sw $s7,0($sp)
sw $s6,4($sp)
sw $s5,96($sp)
sw $s4,100($sp)
sw $s3,104($sp)
sub $sp, $sp, 4
sw $ra,0($sp)
li  $v0, 1
move $a0, $s3
syscall
```

---

As you can see, it loads everything at the start of the block, and just before a link to another block, it stores the vars

The following is a sample CFG generated upon calling the –cfg switch. It contains block numbers, blocks it points to and the associated instructions As you can see a dummy "block0" was generated for the library call **printi**

---

```
----------------------------
----------------------------
 BLOCK NUMBER 0
----------------------------
 POINTS TO BLOCK NUMS: 2, 3, 4, 5, 6, 7, 8, 9, 10,
----------------------------
 INSTRUCTIONS:
#printi
----------------------------
----------------------------


----------------------------
----------------------------
 BLOCK NUMBER 1
----------------------------
 POINTS TO BLOCK NUMS: 2, 0,
----------------------------
 INSTRUCTIONS:
main:

assign, a, 5,
assign, b, 50,
add, 100, a, _t0
```

add, _t0, b, _t1
assign, c, _t1,
call, printi, c
---
---

---
---
BLOCK NUMBER 2
---
POINTS TO BLOCK NUMS: 3, 0,
---
INSTRUCTIONS:
sub, 100, a, _t2
sub, _t2, b, _t3
assign, c, _t3,
call, printi, c
---
---

---
---
BLOCK NUMBER 3
---
POINTS TO BLOCK NUMS: 4, 0,
---
INSTRUCTIONS:
mult, 100, a, _t4
mult, _t4, b, _t5
assign, c, _t5,
call, printi, c
---
---

---
---
BLOCK NUMBER 4
---
POINTS TO BLOCK NUMS: 5, 0,
---
INSTRUCTIONS:
div, 500, a, _t6
div, _t6, b, _t7
assign, c, _t7,
call, printi, c
---
---

---

BLOCK NUMBER 5

---

POINTS TO BLOCK NUMS: 6, 0,

---

INSTRUCTIONS:

```
sub, 100, 50, _t8
add, _t8, 25, _t9
assign, c, _t9,
call, printi, c
```

---

---

---

BLOCK NUMBER 6

---

POINTS TO BLOCK NUMS: 7, 0,

---

INSTRUCTIONS:

```
div, 100, 50, _t10
mult, _t10, 25, _t11
assign, c, _t11,
call, printi, c
```

---

---

---

BLOCK NUMBER 7

---

POINTS TO BLOCK NUMS: 8, 0,

---

INSTRUCTIONS:

```
mult, a, b, _t12
add, _t12, a, _t13
add, _t13, b, _t14
sub, _t14, a, _t15
div, b, a, _t16
sub, _t15, _t16, _t17
assign, c, _t17,
call, printi, c
```

---

---

---
---
BLOCK NUMBER 8
---
POINTS TO BLOCK NUMS: 9, 0,
---
INSTRUCTIONS:
div, 200, 4, _t18
div, 25, 5, _t19
sub, _t18, _t19, _t20
assign, c, _t20,
call, printi, c
---
---

---
---
BLOCK NUMBER 9
---
POINTS TO BLOCK NUMS: 10, 0,
---
INSTRUCTIONS:
mult, b, a, _t21
sub, 100, _t21, _t22
add, _t22, 7, _t23
assign, c, _t23,
call, printi, c
---
---

---
---
BLOCK NUMBER 10
---
POINTS TO BLOCK NUMS:
---
INSTRUCTIONS:
**return**, 0, ,
---
---

---

## 4.3 BriggsAllocator

NOT IMPLEMENTED

# 5 Function

Functions implement the following MIPS convention:

args are stored in $a0..an registers

return values are provided in $v0

A function is called as follows:

1) All active vars are stored on the stack (including the existing $r0 register) by subtracting the stack by sizes of the vars and saving the words

2) args are stored in $a.. registers

3) "jal" instruction is used to jump to another function

4) function reads args, declares vars and then stores return value in $v0

5) "jr" is issued to return to caller

6) Caller reads back vars into registers from stack and return value from $v0 and increments stack $s0

# 6 Limitations and room for improvement

1) BriggsAllocator and the bonus SVN questions were not implemented due to personal circumstances...:(
2) Once again, the class files are header heavy. Could be factored to be cleaner
3) MIPS coding convention was followed fairly well throughout this, however I did have a few hacks:
a) Since $v1 is not used by anything in the scope of this class, we store the stack pointer to the static variables there for convenience
b) Normally in mips, temp registers t1..tn registers are supposed to be saved by the caller, and s1..sn are stored and reloaded by the callee. Instead, we chose to do all loading and storing on the caller's side for the sake of convenience
4) we only support 4 args for a function (due to following mips convention) This could be fixed by usinga stack