

# Target Code Generation

## 1 Overview

We finally arrive at the last step for a working compiler: Target Code Generation. Actually, we've already done code generation in phase 3, in which we designed a conceptual runtime environment and translated source code into three-address code following that convention. The three-address code for the SPL language is a kind of mid-level intermediate representation (IR), however, something more low-level remains unhandled. With the three-address code in hand, your compiler will handle these low-level stuff and finally generate executable code, which can run on a MIPS32 machine.

The low-level details for machine code lie on three aspects:

- The three-address code and machine code are not one-to-one correspondent. A single TAC instruction can be translated into multiple machine code instructions, and vice versa. So we have to design our **instruction selection** strategy.
- We assume unlimited number of local variables in three-address code, although it is not the case in real world. For example, there are only 8 general-purpose registers on x86, and 32 registers on MIPS32 (the runtime memory for a program also has a limited capacity) to support computation. We need to carefully design **register allocation** algorithm so that only the active variables reside in the registers.
- We use **ARG** and **CALL** TAC instructions to invoke functions with arguments. However, there are no special instructions for this task in most hardware. We should maintain the call stack using only jump and data-movement primitives.

In this phase, we are going to deal with the above problems. We choose MIPS32 as our target language. MIPS is a Reduced Instruction Set Computer (RISC) Instruction Set Architecture (ISA), and its 32-bit version is called MIPS32. MIPS32 is an ideal target language for our compiler, for two reasons:

1. It is a reduced instruction set, with compact instruction encoding and clear semantics.
2. It provides 32 registers, each with a clear usage convention, so to benefit the design of register allocation/assignment.

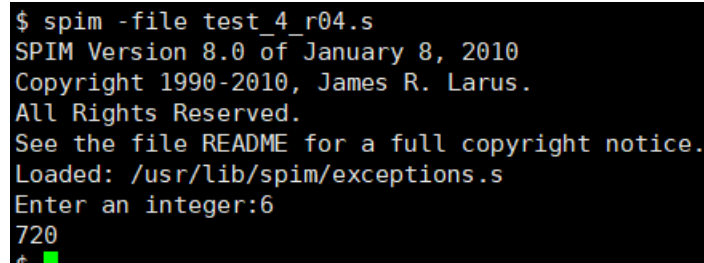
To ease your implementation, we provide you starter code so that you could start quickly. The starter code defines several function stubs so that you could insert your code and check the result immediately. A loading function is provided to directly read and translate a textual IR code file; you could also adapt your compiler front-end to the target code generator. Both approaches are acceptable in our project, but the latter is preferable for it gives you an end-to-end compiler, and, moreover, is more exciting :-)

## 2 Lab Environment

We will evaluate your program on an Ubuntu 18.04 (64-bit) environment. Flex and Bison are still necessary if you implemented target code generation based on previous phases. Again, we give you the following dependencies' versions:

- GCC version 7.4.0
- GNU Make version 4.1
- GNU Flex version 2.6.4
- GNU Bison version 3.0.4
- Python version 3.6.8
- urwid (Python module) 2.0.1
- Spim version 8.0

The generated target machine code can be run directly in a simulator (in fact, even an actual MIPS32 machine). In the lab virtual machine, we've installed the SPIM<sup>1</sup> simulator. SPIM is a self-contained simulator that runs MIPS32 programs. It reads and executes assembly language programs written for MIPS32 processors, rather than the binary executable.



```
$ spim -file test_4_r04.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Enter an integer:6
720
$
```

To run your generated MIPS code (assume the code file path is `test/test.a.s`), simply type the following command in the terminal:

```
spim -file test/test.a.s
```

It is suggested to end the generated code file with the `“.s/.asm”` extension, though SPIM will not check this extension. For more command line options and other functionalities about SPIM, you could refer to its official document.

SPIM also provides a GUI version called QtSPIM. You could follow the instructions on <http://pages.cs.wisc.edu/~larus/spim.html#qtspim> to install QtSPIM in your device. The GUI version provides more user-friendly runtime inspection and debugging, though we will evaluate your code generator on the command-line version of SPIM.

---

<sup>1</sup><http://pages.cs.wisc.edu/~larus/spim.html>

## 3 Instruction Selection

### 3.1 Writing MIPS32 Assembly

#### 3.1.1 Basics

SPIM accepts a textual assembly file and simulates its execution. Typically, an assembly program file ends with the `.s/.asm` extension. An assembly program contains several code (or text) segments and data segments, which correspond to `.text` and `.data` directives. SPIM also supports single line comment, which starts with `#` and ends with a newline character. Programmers can declare constants or global variables in data segments, with the following format:

```
name: storage_type value(s)
```

Here, **name** is a label that locates the memory address to the declared variable, **storage\_type** is the data type of the variable, and **value** is its initial value. Table 1 lists several commonly used data types and their declarations.

Table 1: Commonly used `storage_type` in MIPS32

<code>storage_type</code>	<code>description</code>
<code>.ascii str</code>	store string <code>str</code> in memory, without null-terminate
<code>.asciiz str</code>	store string <code>str</code> in memory, with null-terminate
<code>.byte b1,b2,...,bn</code>	store <code>n</code> 8-bit values in successive bytes of memory
<code>.half h1,h2,...,hn</code>	store <code>n</code> 16-bit quantities in successive memory halfwords
<code>.word w1,w2,...,wn</code>	store <code>n</code> 32-bit quantities in successive memory words
<code>.space n</code>	allocate <code>n</code> bytes of space in the data segment

#### 3.1.2 Registers

MIPS32 has 32 registers, numbered 0~31. Each register has an abbreviation that reflects the register's intended use. Table 2 shows all MIPS32 registers as well as their conventional usages as in `gcc`. Most MIPS32 registers are identical at hardware level, except that `$0` always contains the hardwired value 0. This design implies that the usage conventions are not enforced by the hardware, but should be followed by the software. You can generate working MIPS code that disobeys these conventions. However, since most programmers, compilers, assemblers do follow them, violating the conventions limits the portability of your generated code. What's worse, the simulator may fail to execute them.

Register `$0` always contains the hardwired value 0, which cannot be changed. Registers `$at`, `$k0`, and `$k1` are reserved for the assembler and operating system. They should not be used by user programs or compilers. If your compiler tries to use them, SPIM will raise a syntax error when loading the assembly program. Registers `$v0` and `$v1` are used to return values from functions. Registers `$a0` - `$a3` are used to pass the first four arguments to routines (remaining arguments are passed on the stack).

Table 2: General-purpose registers in MIPS32

register	number	description
\$zero	0	constant 0
\$at	1	(assembler temporary) reserved for assembler
\$v0, \$v1	2-3	(values) expression evaluation or function return
\$a0, \$a1, \$a2, \$a3	4-7	(arguments) function arguments
\$t0, \$t1, \$t2, \$t3, \$t4, \$t5, \$t6, \$t7	8-15	(temporaries) values not preserved across procedure calls, the caller is responsible for saving values
\$s0, \$s1, \$s2, \$s3, \$s4, \$s5, \$s6, \$s7	16-23	(saved values) values preserved across procedure calls, the callee is responsible for saving and restoring values
\$t8, \$t9	24-25	(temporaries) values not preserved across procedure calls, the caller is responsible for saving the values
\$k0, \$k1	26-27	reserved for OS kernel
\$gp	28	(global pointer) to the middle of a 64K static data segment
\$sp	29	(stack pointer) to the top of the stack
\$fp/\$s8	30	frame (activation record) pointer by gcc, or \$s8 by native MIPS32
\$ra	31	return address

Registers \$t0 - \$t9 are *caller-saved registers* that are used to hold temporary values that need not be preserved across calls. The values residing in those registers should be saved into the main memory by the caller before making a procedure call. Registers \$s0 - \$s7 are *callee-saved registers* that hold long-lived values that should be preserved across calls. For the values in callee-saved registers, it is the callee's job to save them into the main memory before really executing callee's code, and load them back to the registers before the callee returns. We will discuss more about this convention later in Section 5.

Register \$gp is a global pointer that points to the middle of a 64K block of memory in the static data segment, typically, it stores a constant address 0x10008000. MIPS32 instructions are all encoded into 32-bit length. Those load and store instructions cannot directly access a 32-bit address since all immediate values are at most 16-bit. In other words, we cannot access address out of 16-bit range in a single instruction. However, with \$gp register, we can use signed 16-bit off set fields to access the first 64 KB of the static data segment. For example, to load the word at address 0x10010020 into \$v0, we can use:

```
lw $v0, 0x8020($gp)
```

Register \$sp is the stack pointer, which points to the top of the stack area, while register \$fp (30) is the frame pointer. The jal instruction writes to register \$ra the return address from a procedure call, which can be used by jr after a procedure is completed and returns.

In short, you may use \$zero as constant zero, assign \$t0 - \$t9, \$s0 - \$s7 for arbitrary use. \$at, \$k0, \$k1 should never be allocated, while others should be carefully maintained during procedure calls.

### 3.1.3 Data movement

MIPS32 separates load and store primitives. Under this architecture, operands of computational instructions must be loaded into the registers in advance. The SPIM supports several addressing modes for loading and storing, as shown in Table 3.

Table 3: Addressing modes supported by SPIM

format	address computation
(register)	content of register
imm	immediate
imm(register)	immediate + content of register
label	address of label
label±imm	address of label ± immediate
label±imm(register)	address of label ± (immediate + content of register)

### 3.1.4 A case of procedure

In the starter code, the instructions for the SPL built-in `read` function is:

```
read:
    li $v0, 4
    la $a0, _prmp
    syscall
    li $v0, 5
    syscall
    jr $ra
```

The first line here defines a label `read`, which corresponds to the function identifier. Note that assembly code doesn't distinguish a label name or a function name, they are all labels. Then the `li` instruction loads the immediate value 4 to the register `$v0`. The `la` instruction loads an address to `$a0`. Here the address is a self-defined identifier `_prmp` declared in the data segment, it is a string of prompt message. The `syscall` instruction triggers a software interrupt to invoke a particular OS service. This service number is already loaded into the `$v0`. In this case, the `syscall` service is 4 (`print_string`). The next two lines perform a `read_int` service. Finally, this procedure is returned by unconditional jump `jr`, to the location stored in the return address register `$ra`.

In the code above, we assume that the return address of `read` function is already stored in the `$ra` register, so the generated code can safely return to the call-site after invoking `read`. The `syscall read_int` will store its return value to `$v0`, so the generated code should obtain this value from the corresponding register. These runtime behaviors are not parts of the MIPS32 specification, but the convention for most real-world compilers. Though we don't require you to follow these conventions, it is strongly recommended to do so because violating them may cause bugs.

### 3.2 Translating Three-Address Code

Instruction Selection is actually a problem of pattern matching. To translate intermediate representation into machine code, we need to find particular patterns, and then convert them into the corresponding instructions. This process is similar to the process of intermediate-code generation, though it is machine-dependent.

We provide a linear IR definition in the starter code, hence the translation is rather straightforward: you can simply convert the TAC by one-to-one mapping. Table 4 shows a mapping scheme for data manipulation and jump instructions. The function `reg(·)` represents the register assigned for a variable, and we will talk more about it later. This scheme is not unique. You could also design your own scheme for better code optimization.

Table 4: An example mapping between TAC and MIPS32

three-address-code	MIPS32 instruction
<code>x := #k</code>	<code>li reg(x), k</code>
<code>x := y</code>	<code>move reg(x), reg(y)</code>
<code>x := y + #k</code>	<code>addi reg(x), reg(y), k</code>
<code>x := y + z</code>	<code>add reg(x), reg(y), reg(z)</code>
<code>x := y - #k</code>	<code>addi reg(x), reg(y), -k</code>
<code>x := y - z</code>	<code>sub reg(x), reg(y), reg(z)</code>
<code>x := y * z</code>	<code>mul reg(x), reg(y), reg(z)</code>
<code>x := y / z</code>	<code>div reg(y), reg(z)</code> <code>mflo reg(x)</code>
<code>x := *y</code>	<code>lw reg(x), 0(reg(y))</code>
<code>*x := y</code>	<code>sw reg(y), 0(reg(x))</code>
<code>GOTO x</code>	<code>j x</code>
<code>x := CALL f</code>	<code>jal f</code> <code>move reg(x), \$v0</code>
<code>RETURN x</code>	<code>move \$v0, reg(x)</code> <code>jr \$ra</code>
<code>IF x &lt; y GOTO z</code>	<code>blt reg(x), reg(y), z</code>
<code>IF x &lt;= y GOTO z</code>	<code>ble reg(x), reg(y), z</code>
<code>IF x &gt; y GOTO z</code>	<code>bgt reg(x), reg(y), z</code>
<code>IF x &gt;= y GOTO z</code>	<code>bge reg(x), reg(y), z</code>
<code>IF x != y GOTO z</code>	<code>bne reg(x), reg(y), z</code>
<code>IF x == y GOTO z</code>	<code>beq reg(x), reg(y), z</code>

It is worth mentioning that, multiplication, division and branching instructions do not take non-zero constants as operands, hence immediates involved in these instructions should be loaded into the registers first.

This translation scheme may produce inefficient code. Take the translation of integer array accessing expression `arr[3]` as an example. Suppose the base pointer `arr` is already

stored in register `$t1` and you want to save the value of `arr[3]` into `$t2`, for one-to-one mapping translation, you may obtain the following code:

```
addi $t3, $t1, 12
lw $t2, 0($t3)
```

By utilizing the addressing mode, however, they can be merged into a single instruction:

```
lw $t2, 12($t1)
```

The example above shows how we can do local optimizations with sliding window. Typically, we will look ahead several TAC instructions to see whether they can be merged into a single target instruction. Such a method is also called *peephole optimization*.

## 4 Register Allocation

Most MIPS32 instructions' operands come from registers, except load/store. Any variable involved in computation/comparison should be firstly loaded into a register. Since there are only a limited number of registers, we need *register allocation* algorithm to decide which values to keep in registers and which registers they will reside in.

The most trivial, and also the most expensive, approach is to store all variables in the memory, and only load those values involved in the next instruction. Since all MIPS32 instructions take at most three addresses, we can simply assign them to `$t0-$t2`. After executing this instruction, we store back those values. Obviously, it produces functionally correct codes, though they might be extremely slow. It is totally fine if you choose to implement this naive algorithm. If you do so, you can skip the advanced algorithms introduced in the remaining subsections.

In the following, we will introduce two algorithms concerning register allocation. It is worth noticing that, finding an optimal allocation is NP-hard in real-world cases, hence modern compilers can only find approximate solutions to this problem.

### 4.1 Local Register Allocation

The major challenge for register allocation is that there is only a limited number of registers. Generally, the number of variables are more than the number of registers, hence the variables should be swapped in-and-out among these registers. Frequently loading and storing variables brings overheads for memory access, hence we need a mechanism to properly assign registers to variables.

*Local register allocation* is such an algorithm, in which the registers are only allocated inside *basic blocks*<sup>2</sup> through heuristics. When the control flow exits a basic block, all allocated registers' values should be stored into the memory. As the program enters a basic block, all

---

<sup>2</sup>The definition of basic block can be found in textbook 8.4

registers are labeled as idle. Then we scan the code in a basic block. If there is a variable that needs to be loaded into a register, do the following:

- if there is an idle register, assign the variable to it
- if no registers are idle, store (or *spilling*) the content of a register to the memory. Obviously, it is good to select the register, of which the content is not going to be accessed in the near future or inside the basic block

The algorithm above adopts a heuristic that attempts to minimize the cost of spilling variables. We try to reduce the overhead of memory access, by greedily choosing the last-accessed variable. Assume that a TAC instruction is with the generic form:

```
z := x op y
```

where *x* and *y* are operands, *op* is the operators and *z* stores the result. The local register allocation does the following routine for each TAC instruction:

```
rx = Ensure(x)
ry = Ensure(y)
rz = Allocate(z)
Emit([rz := rx op ry])
if (x is not needed after the current operation)
    Free(rx)
if (y is not needed after the current operation)
    Free(ry)
```

The function `Free` labels a register as idle, and `Emit` outputs a MIPS32 instruction. The other two auxiliary functions are:

```
Ensure(x):
    if (x is already in register r)
        result = r
    else
        result = Allocate(x)
        Emit([lw result, x])
    return result

Allocate(x):
    if (exists idle register r)
        result = r
    else
        result = register whose value's next-use is the farthest
        spill result
    return result
```

The `getReg` described in textbook Section 8.6.3 is similar to our algorithm. The `getReg` function in textbook introduces register descriptor and address descriptor to eliminate data movement among registers, hoping to minimize the number of load/store instructions. It is more efficient but also more complicated. You can choose to implement the `getReg` function instead of the local register allocation algorithm described here, or even design and implement your own strategy.



## 4.2 Global Register Allocation

The term “local” in the local register allocation stands for the fact that we only focus on the variables inside a basic block. However, if we try to allocate registers across multiple basic blocks, the local strategy is no longer valid. The primary challenge to this problem is: we don’t know the actual control flow of the program, so we don’t know whether a value  $x$  in the register should be spilled into the memory, since the control flow may jump to a basic block that involves  $x$ , or another which does not.

Allocating registers across basic blocks has the advantage of reducing the number of load/store operations, hence improving code efficiency. A typical example is that, since programs spend most of their time in loops, keeping the most active value (such as the index  $i$ ) in a fixed register will save a lot of load/store instructions at block boundaries.

In the local register allocation, we store all *live variables* to the memory at the end of each basic block. In the global register allocation, however, we should determine the liveness of each variable across basic blocks, such technique is called *liveness analysis*. Then, we decide which variables reside in registers, by reducing the register allocation problem to *graph coloring* on a graph of variable interference.

### 4.2.1 Liveness analysis

We firstly define what is a live variable. We say that variable  $x$  is *live* at particular program position, if and only if:

1. if instruction  $i$  uses the value of  $x$ , then  $x$  is live before executing  $i$
2. if  $x$  is live after executing instruction  $i$ , where  $i$  does not define  $x$ , then  $x$  is also live before executing  $i$
3. if  $x$  is live after executing instruction  $i$ , and instruction  $j$  could be reached by jumping from  $i$ , then  $x$  is live before executing  $j$
4. if instruction  $i$  defines  $x$  but does not use it, then  $x$  is not live (or *dead*) before executing  $i$

Rule 1 defines the origin of a live variable, Rules 2 and 3 define how the liveness propagates, and Rule 4 defines when the liveness is over.

We then define the *successor* of instruction  $i$  as:

1. if  $i$  unconditionally jumps to instruction  $j$ , then  $\text{succ}[i] = \{j\}$
2. if  $i$  jumps to  $j$  with a condition, then  $\text{succ}[i] = \{j, i+1\}$
3. if  $i$  is a return statement, then  $\text{succ}[i] = \emptyset$
4. for other cases,  $\text{succ}[i] = \{i+1\}$

Then we define the set  $def[i]$  as the variables defined at instruction  $i$ ,  $use[i]$  as the set of variables that are used/read at  $i$ ,  $in[i]$  as the variables that are live before executing  $i$ , and  $out[i]$  as those live variables after executing  $i$ .

Following these definitions, we can formalize the liveness analysis problem as the data-flow equations:

$$\begin{cases} in[i] = use[i] \cup (out[i] - def[i]) \\ out[i] = \cup_{j \in succ[i]} in[j] \end{cases} \quad (1)$$

The equation system can be solved iteratively: at the beginning,  $in[i] = \emptyset$ . Then, for each instruction  $i$ , we update  $in[i]$  and  $out[i]$  according to (1), until all sets converge to fixed sizes. Actually, according to the lattice theory, the evaluation order of  $in[i]$  and  $out[i]$  does not affect the convergence. However, evaluating them in a reverse order (i.e., from the end of the code to the beginning) may converge faster.

#### Implementation Hint

To perform set operations effectively, a good practice is to apply **bit vector**. Assume that there are 9 elements in the universal set, then we can use 9 bits to represent such a set and all its subsets. A 1 at the  $i$ -th position indicates that the  $i$ -th element belongs to this set, while 0 indicates the opposite. Under such a representation, the intersection of two sets can be performed by boolean-and operation, the union of two sets can be performed by boolean-or operation, and the complement of a set can be performed by boolean-not operation.

#### 4.2.2 Allocation by graph coloring

Based on the result of liveness analysis, we are able to do global register allocation. Obviously, we should assign two live variables to distinct registers, otherwise, lots of load/store operations will be introduced to access these live variables. However, there are two exceptions:

- for assignment  $x := y$ , both  $x$  and  $y$  can share the same register, even both of them are live after assignment, because they are equivalent
- for binary operation  $x := y + z$ , if  $x$  is no longer live after this instruction, while  $y$  is still live, they should not share the same register, otherwise, assigning a new value to  $x$ 's register will also overwrite  $y$ 's value

According to the analysis above, we define the *interference* between variables  $x$  and  $y$  as:

1. there exists an instruction  $i$ , where  $x \in out[i]$  and  $y \in out[i]$
2. there exists an instruction  $i \neq [x := y]$ , where  $x \in def[i]$  and  $y \in out[i]$

Here the set  $def[i]$  and  $out[i]$  are collected through liveness analysis as described in Section 4.2.1. The most important thing at this stage is: if  $x$  and  $y$  interfere with each other, we shall assign them to distinct registers.

Let a variable in the program as a vertex, and the interference relation between two variables as an edge, then we are able to draw an *interference graph* for a program. Assume that we will assign each variable a register, while there are  $k$  registers ( $k$  colors) available. What's more, two adjacent vertices cannot share the same register, or equivalently they cannot have the same color. Then the register allocation is reduced to the well-known *graph coloring* problem.

For a fixed  $k$ , deciding whether a graph can be  $k$ -colorized is NP-Complete. To solve this problem in polynomial time, we typically adopt heuristics. A simple heuristic for  $k$ -coloring is call the *Kempe algorithm*, which does the following:

1. If there is a vertex in the interference graph that has a degree of less than  $k$ , remove it from the graph and then push it into a stack.
2. Repeat step 1, until no vertex can be removed. If there are less than  $k$  vertices left, assign them with different colors, then pop the vertices from the stack, and assign each one with a color distinct from its neighbors. Otherwise, go to step 3.
3. Remove and push a vertex, and mark it as “spilled”, go to step 1.
4. If a popped vertex is marked as “spilled”, check the colors of its neighbors. If its neighbors are colored by less than  $k$  colors, then simply assign it by the last color, otherwise, leave this node **uncolored**, which means it should be spilled into the memory.

The rationale of step 1 is, when such a vertex is dropped, if the resulting graph is  $k$ -colorizable, then the original graph must also be  $k$ -colorizable. For a graph that each vertex has at least  $k$  neighbors, it is still possible to  $k$ -colorize it. But we cannot efficiently compute the answer since it is NP-Complete. Nevertheless, the Kempe's algorithm gives us a nice heuristic solution for the graph coloring problem, especially in the context of register allocation.

After running graph coloring algorithm, all variables in the intermediate code are either assigned with a register, or marked as “spilled”. For a spilled variable, if it is accessed by some computation, it should be loaded into a particular register; after the computation, it will be stored back from the register to memory. To allocate register for it, the simplest way is to pre-define some registers for spilled variables. There are more strategies for better runtime performance, which can be found in the textbook.

## 5 Procedure Call Convention

### 5.1 Stack Layout

When invoking a function, or *procedure*, there are two flows a compiler should consider, namely, the *control flow* and the *data flow*. Basically, control flow transition means to store the program counter at the register `$ra` before calling a procedure, and restore the PC from this register before the procedure returns. This mechanism can be easily maintained by `jal` and `jr` instructions in MIPS32. So your compiler should focus on the data flow transition. Here, we are interested in passing parameters and return values.

Technically, you can store the values anywhere, except some immutable registers (i.e., `$zero`, `$at`), though you are recommended to follow the usage convention (Section 3.1.2). To pass parameters, you should utilize both registers and the stack. For functions with at most four parameters, the parameters' values should be loaded into `$a0`, `$a1`, `$a2`, `$a3`, respectively. The fifth and the following parameters will be stored in the stack. The return value of a function is stored in `$v0`.

The most essential memory segment to realize procedure calls is the **stack**. In runtime, each function will store its *activation record* in the stack, which is also called a *stack frame*. The layout of a stack frame can vary among architectures, or even compiler implementations. Figure 1 shows a typical stack frame layout. In this layout, `$sp` points to the top of stack, while `$fp` points to the bottom of the activation record for the called procedure. If this procedure takes more than four parameters, the following parameters will be stored after the `$fp`. The memory space between `$fp` and `$sp` can store any information needed.

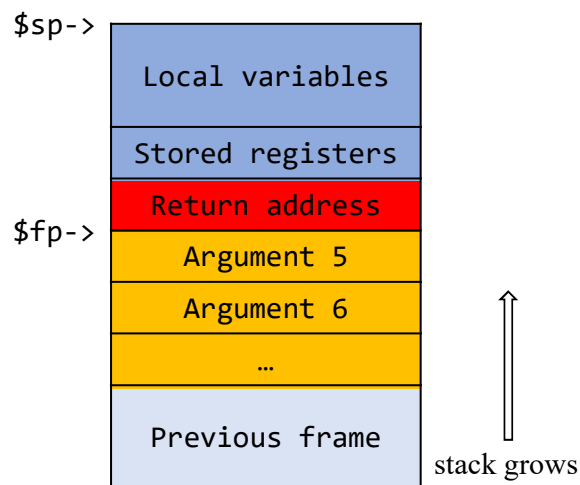


Figure 1: A typical stack frame layout

The **Return address** in the figure is the return address of the **caller** function, while the return address of the callee function is stored in `$ra`. The caller's return address should be restored when the callee exits. This is because when we use the instruction `jal` to call a

procedure, it will overwrite the content of the register `$ra`, hence we need to store the old value of `$ra` to the stack of the called procedure such that it can be restored later.

In register allocation, some variables will be spilled to the memory from registers. Like the `$ra` register, the `$fp` register should also be saved if you adopt the frame pointer implementation (i.e., the stack allocation method discussed in the textbook), though it is not necessary. For local variables, we will store them in the stack. As for global variables, they should be spilled to the writable data segment. Since we don't introduce global variables in project phase 4, you can simply ignore the latter case.

The local variables are stored, typically in their declaring order, in the stack. Also, array and structure variables should be stored in the stack, even though they only consist of single field that fits the register's size.

## 5.2 Calling & Return Sequences

If a procedure/function  $f$  calls another procedure/function  $g$ , then we call  $f$  the *caller* and  $g$  the *callee*. In callee's procedure, some registers will be overwritten. Since their values may be used by the caller after executing the callee, so they should be spilled to the memory before executing callee's code and recovered after execution. Then a question raises: where should these register-saving operations be performed? In the caller or the callee? At the caller's view, it doesn't know which values will be used by the callee; as for the callee, it doesn't know which values are still useful for the caller after the call. So they can only store all registers inside their procedure, which causes inefficiency.

A general approach to this problem is to store the registers both in caller and callee. MIPS32 provides both caller-saved registers (`$t0-$t9`) and callee-saved registers (`$s0-$s8`). It is suggested that, caller should save the content of those registers (some of `$t0-$t9`) that will be written during procedure call. On the other hand, callee should save the content of those registers (some of `$s0-$s8`), which is still useful after the call.

### 5.2.1 Caller's sequence

When the caller calls another function, it should store all caller-saved registers' values to the stack, then load arguments to the registers (or push to the stack). When the callee returns, the caller should restore those saved register values. The instruction sequence for this routine is given below:

```
sw live1, offsetlive1($sp)
...
sw livek, offsetlivek($sp)
subu $sp, $sp, max{0, 4*(n-5)}
move $a0, arg1
...
move $a3, arg4
sw arg5, 0($sp)
```

```

...
sw argn, (4*(n-5))($sp)
jal callee
addi $sp, $sp, max{0, 4*(n-5)}
lw live1, offsetlive1($sp)
...
lw livek, offsetlivek($sp)

```

The code above assumes that all arguments are loaded to registers before function call. In practice, you can also compute and pass the arguments one by one. However, if you choose the latter strategy, the offset of each variable should be carefully maintained, since the value of stack pointer `$sp` may change during the process. To solve this problem, you can also choose the frame pointer `$fp` as the base address, rather than using `$sp`.

### 5.2.2 Callee's sequence

The calling sequence is at the top of the callee's body, which is also called the *prologue*; the return sequence, on the other hand, is at the bottom, which we call *epilogue*.

The prologue is responsible for setting up the activation record, storing the return address and the frame pointer. Then, the callee-saved registers are stored, and the arguments are fetched<sup>3</sup>:

```

subu $sp, $sp, fscallee
sw $ra, (fscallee - 4)($sp)
sw $fp, (fscallee - 8)($sp)
addi $fp, $sp, fscallee
sw reg1, offsetreg1($sp)
...
sw regk, offsetregk($sp)
lw p5, (fscallee)($sp)
...
lw pn, (fscallee + 4*(n-5))($sp)

```

In the epilogue, the callee-saved registers are restored, as well as the stack:

```

lw reg1, offsetreg1($sp)
...
lw regk, offsetregk($sp)
lw $ra, (fscallee - 4)($sp)
lw $fp, (fscallee - 8)($sp)
addi $sp, $sp, fscallee
jr $ra

```

---

<sup>3</sup>The notation `fsg` stands for the frame size of function `g`

As we suggest in the caller's sequence, you can also reference the stack by the frame pointer, rather than the stack pointer.

Finally we discuss how the procedure call affects the register allocation. The callee-saved registers `$s0-$s8` preserve their values across procedure calls, hence the caller doesn't need to consider their allocations. What's important is the caller-saved registers `$t0-$t9`, since their values are lost after procedure call.

If you adopt local register allocation, then all values in `$t0-$t9` should be spilled before `CALL` instruction, and they will be restored after procedure call. As for global register allocation, you should avoid assigning `$t0-$t9` to those variables live at `CALL` instruction. If you adopt the naive register allocation, you can ignore these suggestions.

## 6 Project Requirements

### 6.1 Basic Requirements

In the target-code generation stage, the input file is a textual intermediate program, i.e., the `splc` accepts an argument of `.ir` file. Your task here is to design and implement a register allocation algorithm, and translate a set of TAC instructions.

We've prepared you with an archive of starter code. You can compile the code with the `splc` target in `Makefile`, and move the executable to the `bin` directory. For example, for the `Makefile` placed under the project's root directory, we make the `splc` target by:

```
make splc
```

which generates the target code generator executable. Then we run it by:

```
bin/splc test/test_4_r01.ir
```

The generator will print the corresponding MIPS32 code to the standard output. Alternatively, you can also print it to the text file `test/test_4_r01.s`.

Unlike previous phases, this time we have neither optional nor bonus requirements. Your tasks (the required functions) are already declared in the starter code. All you need is to implement them. You are free to modify the provided code, including the structure definitions.

### 6.2 Assumptions

Here we present the assumptions for our test cases. You can build your code generator safely by ignoring their violations. They are:

**Assumption 1** the intermediate code are logically correct, meaning that you can run them on the IR simulator and obtain the correct output

**Assumption 2** there are no structure or array variables, so you don't need to translate the `DEC` instruction

**Assumption 3** all integer constants are in the range  $[-2^{16}, 2^{16}-1]$  so that they can be safely represented by MIPS32 immediates

## 6.3 Required Tasks

### 6.3.1 Register allocation

Your first task is to design and implement a register allocation algorithm. We have discussed two register allocation algorithms in Section 4, in addition to a very naive one. The two algorithms vary in code efficiency and implementation complexity. You can also implement your own strategy to achieve better performance.

We have defined the register descriptor (`struct RegDesc`) and the address descriptor (`struct VarDesc`) in the provided file `mips32.h`. You can add (or delete) fields as you need. You should implement two functions `get_register` and `get_register_w`. The “\_w” suffix indicates the obtained register will be used to load value. The major reason for separating register usage is that, overwriting a register makes the corresponding variable no longer live at that location, hence the old value of the register should be spilled to the memory for data consistency (if needed). Also you should implement the `spill_register` function, which will be called when there are no available registers for new values.

### 6.3.2 TAC translation

Another task you are required to accomplish is to translate some TAC instructions. The starter code already provides some examples. Code listing 1 shows the translation function of LABEL TAC statement. Each emitter function accepts a structure of TAC instance, and returns the next TAC instruction to be translated. This pattern enables local optimization during target code generation.

Listing 1: Translation of LABEL statement

```
1 tac *emit_label(tac *label){
2     assert(_tac_kind(label) == LABEL);
3     _mips_printf("label%d:", _tac_quadruple(label).labelno->int_val);
4     return label->next;
5 }
```

Your major task here is to translate the procedure call sequences, which correspond to four TAC instructions: ARG, CALL, PARAM and RETURN. You can also modify the implemented functions for more optimized code.

## 6.4 Grading Policy

Our test suite contains several intermediate representation programs. You should implement the register allocation and TAC translation so that they will be compiled into working MIPS32 assembly code. Your score depends on which test cases your code generator can pass (80 points), as well as your design and implementation (20 points). To help us assess your design



and implementation, you should write a report (no more than 4 pages) to illustrate your algorithms and how you do optimizations, etc..

There may be multiple members in your team. In that case, we will evaluate the contribution of each member and rate his/her contribution using four levels: A (significant), B (moderate), C (low), and D (very little contribution). If the rating is A, the member will get 100% of the grade obtained by the team as a whole ( $G$ ). If the rating is B, the member will get 85% of  $G$ . If the rating is C, the member will get 60% of  $G$ . If the rating is D, the member will get 30% of  $G$ . We hope everyone gets rated as A or B.

## 7 Submission

**What to submit** You are required to submit your source code and other related files in a .zip archive with file name format: **StudentID-phaseNumber.zip** (e.g., 11849180-phase4.zip). In this project, the zip file tree is:

```
StudentID-phase4/
  bin/
    splc    // generated
  report/
    StudentID-phase4.pdf
  test/
    test_4_r01.ir
    test_4_r01.s
    test_4_r01.spl
    ...    // our provided tests
    test_a.ir
    test_a.s
    test_a.spl
    ...    // your self-written tests
  Makefile
  ...    // source code
```

- **bin** directory contains a single executable file named `splc`, which is generated by an `splc` target in the `Makefile`, make sure it works properly in our environment (Section 2).
- **report** directory contains a pdf file that illustrates your design and implementation. Your report should not exceed **4 pages**, we suggest you to use 11pt font size and single-line spacing for main content.
- **test** directory is released in the GitHub repository, though you may add additional test code for your own. We do not constrain the naming convention on test code, except the IR file name should end with `.ir` extension and generate MIPS program file name should end with `.s`.

- The **Makefile** is provided. You can add any targets, but most importantly, **you have to guarantee the `splc` target compiles and generates a single executable `splc` (the compiler) in the `bin/` directory**. Otherwise, we cannot evaluate your work and you will get 0 for this phase.
- In this phase, you will write code mostly in C/C++, and probably Flex/Bison, you can place them directly under the submitted directory, or under separate folders like `src` and `include`. Again, it's your job to ensure the code can be compiled successfully.

**How to submit** You should upload your zip file on SAKAI before the **11:55 PM, January 10, 2023**. This time we **don't** have a grace period. Late submission will not be evaluated. Please try your best to finish the last mile.

**Contact** For any question regarding this project phase, feel free to contact our teaching team via emails (typically reply within 24 hours):

[CS323] Project Phase 4 (**YourName-StudentID**)

e.g., [CS323] Project Phase 4 (WangSinan-11849180)