CS323 Project Phase 4 Target Code Generation

Introduction

In order to generate the final target machine code, the three-address code generated in the third stage was translated into MIPS32 target machine code by means including register allocation, procedures management, and branch condition conversion.

Algorithms

Register Allocation

For register selection and allocation, we use the register allocation algorithm described in the documentation, i.e., when a variable needs to be allocated a register, we traverse all the registers, and when we find an empty register, we directly allocate the register to the variable. We also reserve some registers for storing temporary variables, i.e t0, t1, etc. In this project, we assume that all variables occupy a word length like an integer, so we use a continuous array to store them.

To maintain the mapping between variables and registers, we maintain the Node node, which can be used as a linked list. The `offset` is the offset of the corresponding variable in the MIPS array. The `key` is the name of the node. The `Reg` is the register the variable in. We can use the `next` element to find the next Node, which acts like a linked list.

```
struct Node
{
  int offset;
  char *key;
  struct Node *next;
  Register reg;
};
```

Around the register selection, we have two related functions, `findOffset` and `findRegister`, where the `findOffset` function loads the relevant variable from memory, and if the variable is already rvalue, we will use `findRegister` directly to find the corresponding node and then extract the registers from it.

```
int findOffset(struct Node *head, char *key)
{
    // create a link
    struct Node *temp = head;
    while (temp->next != NULL)
    {
        temp = temp->next;
        if (strcmp(temp->key, key) == 0)
        {
            return temp->offset;
        }
    }
    return -1;
}
```

```
struct Node *findRegister(struct Node *head, char *key)
{
    struct Node *temp = head;
    while (temp->next != NULL)
    {
        temp = temp->next;
        if (strcmp(temp->key, key) == 0)
        {
            return temp;
        }
    }
    return NULL;
}
```

Procedure Management

The more difficult part of this project is handling the intermediate code related to function calls. The goal of this task is to handle which variables local to the function need to be saved so that they can be restored after the function call is complete.

In order to manage the process better, the first thing we need to do is to calculate the number of instructions of the formal and real parameters. In this part we use the attribute of the linked list, by iterating through the parameters of the process, we continuously calculate the attribute value of the current node `next` until it is empty so that by calculating the number of parameters we can know exactly the offset of the address of the function to be shifted into.

```
tac *emit_param(tac *param)
{
    /* COMPLETE emit function */
    tac *pointer = param;
    char *varName;
    int paramCount = 0;
    while (_tac_kind(pointer) == PARAM) // 不断遍历 next 直到为空
    {
        paramCount++; // 用于参数计数
        varName = pointer->code.param.p->char_val;
        if (findRegister(a0_a3_list, varName) == NULL)
        {// 如果没有对应的寄存器，就在对应的位置插入该变量

            insertParam(a0_a3_list, varName, paramCount + 3);
        }
        pointer = pointer->next;
    }
    return pointer;
}
```

For the recursion function in `test_4_01.spl`, the calling sequence of Hanoi is

```
addi $sp, $sp, -20 #### fixed
  sw $ra, 0($sp)
  sw $a0, 4($sp) #### fixed
```

```
sw $a1, 8($sp) #### fixed
sw $a2, 12($sp) #### fixed
sw $a3, 16($sp) #### fixed
lw $a0, array + 84
lw $a1, array + 88
lw $a2, array + 92
lw $a3, array + 96
jal hanoi
lw $ra, 0($sp)
lw $a0, 4($sp)
lw $a1, 8($sp)
lw $a2, 12($sp)
lw $a3, 16($sp)
addi $sp, $sp, 20
move $s0, $v0
sw $s0, array + 100
```

By doing this iterative process, we can handle variables other than function arguments and return addresses before the calling sequence, and only handle function arguments and function return addresses during the calling sequence, reducing the workload of the calling sequence and increasing the efficiency of the calling sequence.

Condition Translation

For the machine code to be generated, the important task is not only the parsing of the procedure but also the handling of conditional branches. In TAC, we have defined six kinds of conditional statements, respectively `IFLT,IFLE,IFGT,IFGE,IFNE,IFEQ`, the important point in the processing of conditional branches is the comparison of immediate data, and without optimization, this process would require an instruction dedicated to loading such immediate data into memory, which would cause some loss in the efficiency of conditional statements. Our group has made an optimization for this case, i.e., we have done so by reserving two registers dedicated to the case of immediate data in conditional branches. This approach takes advantage of the fact that registers only need to perform a bit of transfer function and avoids the overhead of register selection.

For `emit_iflt()` example

```
tac *emit_iflt(tac *iflt)
{
    /* COMPLETE emit function */
    Register x, y;
    if (_tac_quadruple(iflt).c1->kind == OP_CONSTANT)
    {
        x = t5; // only assign t5
        _mips_iprintf("li %s, %d", _reg_name(x),
                      _tac_quadruple(iflt).c1->int_val);
    }
    else
    {
        x = get_register(_tac_quadruple(iflt).c1);
    }
    if (_tac_quadruple(iflt).c2->kind == OP_CONSTANT)
    {
        y = t6; // only assign t6
        _mips_iprintf("li %s, %d", _reg_name(y),
                      _tac_quadruple(iflt).c2->int_val);
    }
    else
    {
        y = get_register(_tac_quadruple(iflt).c2);
    }
    _mips_iprintf("blt %s, %s, label%d", _reg_name(x), _reg_name(y),
_tac_quadruple(iflt).labelno->int_val);
    restore_reg();
    return iflt->next;
}
```

By this means, we can optimize the conditional translation procedure.

Acknowledgment

I would like to thank Prof. Liu and the teaching assistants for their hard work this semester, and for your careful review of each assignment and project and your prompt responses to questions. We feel that the quality of these projects is very high, which can enhance our understanding of compilation principles, and we will continue to work hard to improve our coding skills.