CM30171

Rob Willison

December 9, 2016

Contents

1 Overview

This report is an explanation of the design decisions undertaken while writing a –C compiler and interpreter. The project was written in C and involves three different sections of code, an interpreter, TAC compiler and MIPS compiler.

The interpreter for –C handles functions, closures, if - else statements and while loops. It uses an environment comprised of frames to store variable values in different scopes. Each variable can either be a integer or closure.
In order for functions to work the AST is walked to fetch all the function definitions there are in the code. For each definition the token and AST node are stored in the environment, then the AST node is retrieved and jumped to when the function is called. An extension to this is closures which are stored like functions however a pointer to the enviroment frame in which it was defined is also stored.

The TAC Compiler supports the same features as the interpreter. It translates the –C code into TAC blocks and performs block level optimistation to reduce the number of instructions generated. In order to perform this optimisation the next use information is calculated for each of the blocks. Just like in the GCC compiler the basic blocks have two links, one to the previous block and one to the next [1]. These two link the blocks in a way as to preserve the instruction flow in the source code. It was decided not to implement a control

flow graph [2], this was because this information is not required for the basic block optimisation or the techniques used to create MIPS code.
As –C allows for functions to be defined within functions the compiler must support nested function definitions. In the TAC these nested functions are flattened so there are no nested function definitions, this was done to simplify and unify the way all functions apear in the generated mips code. For this to work the TAC translator renames all functions to numbers using a simple count variable, this allows functions in diffrent scopes to still have the same name.
The environment for the TAC compiler consists of a linked list of frames each frame containing a binding of token to the TAC register it is saved in, this is one of the simplest ways to implement an environment [3]. This environment structure is very similar to that of the interpreter and MIPS translator.

Once the TAC has been generated the MIPS translation phase can take place. This takes the TAC Blocks and converts them to MIPS code. The TAC blocks are translated one at a time, with each TAC instruction being converted to a number of MIPS instructions. Every TAC register is stored in the activation frame and when it is needed it is loaded into a register then stored back to the frame afterwards. This is the simplest solution and was not improved due to running out of time. The disadvantages of this are that there are far more instructions than required and therefore the program will run slower than the ideal. The advantages of this are that it is very simple to implement as it requires no knowlage of the program structure or data flow, so a control flow graph would not be needed. Also there is no limit on the number of variables that can be used which would be the case if TAC registers where directly mapped to MIPS ones. It was planned that an graph coloring algorithm would be implemented such as the one proposed for gcc [4] that uses block level and global level register allocation phases, however there was not enough time to finish this. In order to store variables in MIPS an activation frame is created for each function that is called. This activation frame has space for all the temporary and local variables as well as any arguments passed to the function. There is another environment structure for the MIPS translation phase, it is very similar to the other two. The environment is made up of a series of frames each holding the bindings between a TAC register and a index into the current activation frame.

2 Running the Compiler and interpreter

compile the source code by running "make all' then to run the interpreter run "./mycc -i path/to/sourcefile' or to run the compiler run "./mycc -c path/-to/sourcefile' this outputs a file in the Output folder whcih contains the mips code.

3 Code Interpretation

In order to interpret –C code the abstract syntax tree is walked, at each node the left and right children are compiled and the operation relating to the node is applied to them. In order to support both integer and closure return types all the functions which handle node interpretation return a union structure which either contains the integer value or a pointer to the closure. The interpreter supports variable assignment, closures, if-else and while statements I will explain each now. All the code for the interpreter is in the interpreter.c source code file.

3.1 Variable Assignment

In order to store a value an environment structure was implemented this consists of a list of frames, one for each scope in the program, and each frame contains a list of bindings of token to union, where union is a structure either containing a value or a pointer to a closure. The current environment frame is added to each time a variable assignment is made and new frames are added each time a new scope is entered, frames are removed from the list when this scope is left.

3.2 Closures & Functions

In order to store and call closures there is a closure structure this holds a pointer to the environment frame the closure was defined in and also a pointer to the start node of the AST that defines the function. Then when a closure is applied the variable is pulled out of the environment and the AST interpreted with the environment it was defined with.

The interpretation of the functions AST continues until a return node is found, at this point the hasReturned field is set in the union. When the hasReturned field is set no other lines of code in the current function are executed and the function returns the value in the union.

3.3 Control Structure

There are two types of control structure in –C the if-else and while loop. In order to interpret an if - else statement the condition is interpreted and depending on if the value is non-zero or not either the right or left child are interpreted.
For the while loop firstly the conditional is interpreted then the body, this is repeated while the conditional is non-zero.

4 Intermediate Code Generation

4.1 TAC Design

The three address code used in the compiler was designed to be abstract enough as too keep any machine dependent decisions out of the TAC stage. Many of the instructions are obvious such as store and mathematic operations, examples below, and they won't be explained in great detail.

```
r2 := 1
r3 := r1 / r2
```

One thing to note about the store instruction is that in the event that the store operand is not defined in the current scope extra information is added, for example while interpreting the return statement in this example x is defined in the scope above.

```
int main()
{
  int x = 4;
  int test()
  {
    return x;
  }

  return test();
}
```

In that scenario the store command in the test function will have an extra piece of information saying that its defined in the scope one level above, so the store command will look like the following.

```
DEFINED IN 1 r2 := x
```

This information is also included when a closure is called, the TAC would be as follows.

```
CALL _1 FROM SCOPE 1
```

This was done to simplify the MIPS translation phase so the compiler knows that the token is stored in the environment frame above the current one, and can fetch it appropriatly.
For Functions a label instruction denotes the start and a end. After the start label the new activation frame instruction appears which tells the compiler to allocate space for a new activation frame with a given number of arguments, locals and tempories. Before the end there is a return instruction which contains the register with the value to return, it can also be empty i.e. no value returned. Below is an example function in TAC.

```
_1:
NEW FRAME 0 arg 0 loc 1 temp
DEFINED IN 1 r2 := x
RETURN r2
FUNCTION END
```

one of the control structures in the language is the if else, the way this is described in TAC is using a sequence of if instructions and labels denoting the various bodies if the if and else parts. There is an example below.

```
if (1 > 4) {
  return 4;
} else if (2 > 1) {
  return 3;
}
```

```
r1 := 1
r2 := 4
r3 := 1 > 4
IF NOT r3 GOTO 1
r4 := 4
RETURN 4
GOTO 2
LABEL 1: r5 := 2
r6 := 1
r7 := 2 > 1
IF NOT r7 GOTO 3
r8 := 3
RETURN 3
LABEL 3: LABEL 2:
```

Finally there is one loop type in the language, the while loop, this is translated into a goto, an if and a label at the top and bottom of the loop body. the while condition is placed after the body. An example in –C and the corresponding TAC are below.

```
int x = 0;
while (x < 5)
{
  x = x + 1;
}
```

```
r1 := 0
x := r1
GOTO 1
LABEL 2: r2 := x
r3 := 1
r4 := r2 + r3
x := r4
LABEL 1: r5 := x
r6 := 5
r7 := r5 < r6
IF r7 GOTO 2
```

4.1.1 List of TAC instuctions

x := y	Store the value of y in x
x := y + z	Store the result of y + z in x
x := y − z	Store the result of y − z in x
x := y ∗ z	Store the result of y ∗ z in x
x := y / z	Store the result of y / z in x
x := y < z	Store the result of y < z in x
x := y > z	Store the result of y > z in x
x := y >= z	Store the result of y <= z in x
x := y <= z	Store the result of y >= z in x
LABEL x:	Create a label with the name x
GOTO x	jump to the label with name x
IF NOT x GOTO y	If x has the value 0 jump to y
CREATE CLOSURE x	define closure with the code label x
x:	Also a label of value x
ALLOCATE PARAMS x	Allocate a space for x parameters
SAVE PARAM x	Save x in the next parameter space
NEW FRAME x arg y loc z temp	Allocate space for the activation frame
RETURN x	Return the value in x
FUNCTION END	A tag for the end of the function

4.2 TAC Generation

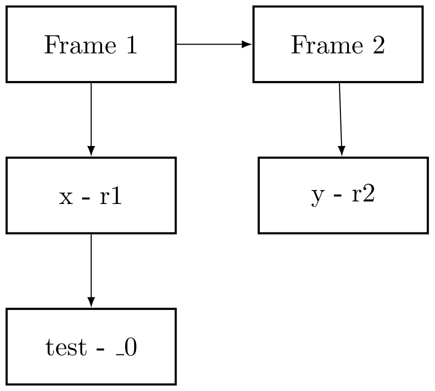
All the TAC complilation is done in the tac_compliler.c file. In order to generate the TAC from source code a tree walk is performed over the parse tree at each node depending on the type a set of TAC instructions are created and added to the current TAC block. If a new function is found a new TAC block is created before that part of the tree is parsed, also if a goto or label TAC instruction are created new blocks are created. When a leaf in the AST is reached a store instruction is created for the value at the leaf.

4.2.1 Enviroment Structure

In order to keep track of the location of any local variables in the code an environment is created to store the association between token and register location. The environment is made up of frames each frame corresponds to a scope in the program and has a linked list of all locals in that scope. The frames also contains a linked list of the functions defined, these are a pair of token and label assigned to the function. For example the environment created while walking the following piece of code is shown below.

```
int main()
{
  int x = 4;
  int test()
  {
    int y = 5;
    return x * y;
  }

  return test();
}
```

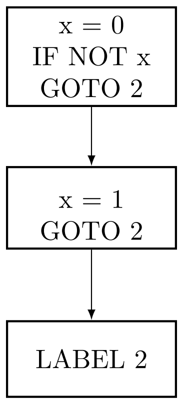


As you can see the environment holds the association between the tokens of the variables and functions with the TAC register location and the function label respectively.

4.2.2 TAC Blocking

The diffrent blocks of TAC code, between labels and jumps, are separated to allow easier optimisation and MIPS translation. As the AST is walked if a instruction which requires a jump (If, While, Function Call) is found a new block is created and set as the current block after the jump. Similarly if a label is created a new block is created and the label is assigned to that new block.

In order to compile the code in the same order as the TAC each block has a link to the block which was created from the code directly after it. A Simple example is given below.



This was done so that the MIPS compilation step can easily follow the sequence of TAC instructions.

4.2.3 Inner functions

In order to implement inner functions, functions defined inside another, the code must be flattened as you cannot have recursive function definitions is this TAC implementation. To do this when a function definition is reached inside another the current TAC block is stored and another is created for the inner function, once the inner function's AST has finished being walked the previous block is restored as the current block and the AST tree walk continues. This gives the result of placing any inner functions after the function it was defined in.

4.3 Next Use Info

As the TAC instructions are broken down into blocks next use information can be worked out to allow optimisation. The next use info for a block is calculated by using a recursive function which looks at each instruction in turn from bottom to top. A list of NEXT_USE_INFO structures is used, one structure for each variable used in the block. Each NEXT_USE_INFO structure contains a list of the uses of that variable and the associated liveness at that point. This is all done in the nextUseInfo.c file.

4.4 TAC Optimisation

After the TAC generation phase has been completed the sequence of blocks is handed over to the optimisation phase. This is in the tac_optimiser.c file. This applies constant folding, copy propagation, dead code elimination, common sub expression elimination and algebraic transformations algorithms to the TAC block repeatedly till no change can be made. Each TAC block is processed from bottom to top for each optimisation technique.

4.4.1 Copy Propagation

Copy propagation looks for a store instruction, once on is found it checks the rest of the code below the store until it finds a write to the store destination. For every read of the destination before a write the use is replaced with the stores operand. To demonstrate this copy propergation was applied to the same example as above.

```
                 r1 := 4            r1 := 4
                 x := r1            x := 4
int x = 4;       r2 := 6            r2 := 6
int y = 6;       y := r2            y := 6
                 r3 := x            r3 := 4
return x + y;    r4 := y            r4 := 6
                 r5 := r3 + r4      r5 := 4 + 6
                 RETURN r5          RETURN r5
```

4.4.2 Constant Folding

The constant folding function looks for arithmetic operations and replaces them with the result where possible. This is done by looking for a operation where both operands are tokens with type constant. Below is an example using both constant folding and copy propergation.

```
                 r1 := 4            r1 := 4
                 x := 4             x := 4
int x = 4;       r2 := 6            r2 := 6
int y = 6;       y := 6             y := 6
                 r3 := 4            r3 := 4
return x + y;    r4 := 6            r4 := 6
                 r5 := 4 + 6        r5 := 10
                 RETURN r5          RETURN 10
```

4.4.3 Dead Code Elimination

Dead code elimination works by analysing each instruction using the next use info and removes them if not needed. Firstly this is only done for registers, user defined variables are live at the end of the block so are not removed. A instruction is removed if the next use is not live or there isn't another next use. Using the same example as above and applying dead code elimination gives the following.

```
                    r1 := 4
                    x := 4
                    r2 := 6
int x = 4;          y := 6
int y = 6;          r3 := 4            RETURN 10
                    r4 := 6
return x + y;       r5 := 10
                    RETURN 10
```

The function which completed the dead code elimination step does not recalculate the number of tempories in the code. So when the activation record is allocated it will be for the original number of registers, which can waste a lot of memory. This would be easy to recompute but I ran out of time.

4.4.4 Common Sub Expression Elimination

If two expression have the same operands and operation the later one can be replaced with a store of the destination of the first. These are found by looking for pairs of arithmatic operations, when one is found if the operation and operands are the same the second is replaced with a store. An exaple is given below usign this method in conjunction with copy propergation.

```
                    r1 := 4             r1 := 4
                    r2 := 6             r2 := 6
                    r3 := r1 * r2       r3 := 4 * 6
                    c := r3             c := r3
int c = 4 * 6;      r4 := 4             r4 := 4
int d = 4 * 6;      r5 := 6             r5 := 6
                    r6 := r4 * r5       r6 := r3
return c;           d := r6             d := r3
                    r7 := c             r7 := r3
                    RETURN r7           RETURN r3
```

4.4.5 Algebraic Transformations

The final optimisation technique is to transform some algebraic expressions which will always give 1 or 0 to those values. In order to do this the optimiser looks for those specific expressions and simply replaces them with a store of either 1 or 0 depending on the expression. An example is given below using copy propagation aswell.

```
                    CALL _1               CALL _1
                    r2 := result          r2 := result
                    x := result           x := result
int x = test();     r3 := 0               r3 := 0
int a = 0 + x;      r4 := result          r4 := result
                    r5 := 0 + result      r5 := result
return a;           a := r5               a := result
                    r6 := r5              r6 := result
                    RETURN r5             RETURN result
```

5 Machine Code Generation

After the TAC is generated and optimised the code is translated to MIPS assembler. This is done block by block in the order the TAC was generated. Before the user code is compiled a global MIPS block is added which allocates memory for global scope functions and sets up the environment used for variable lookup. A decision was made to store all variables in memory and ignore the problem of register assignment, this makes the compiler simpler however reduces its performance. The files associated with this section of the compiler are compiler.c and MIPS.c.
In order for the users code to be run the main function has to be found as it was renamed by the TAC to _0. A mips main function is added at the head of the mips this contains a dynamic allocation of memory for the global scope functions, it then fills these spaces with the functions defined globally. After this it calls the function _0, the users main function.

5.0.1 Functions

When a function is compiled the first thing that happens is the number of locals, temporaries and arguments are counted and a space big enough for all these is allocated. This activation frame also contains space for the return address, previous frame and enclosing frame. Once this is created and the first 3 spaces are filled with the previous frame, the return address and a enclosing scope if needed. The arguments are all loaded into the next spaces in the frame. Any arguments are loaded from the memory pointed by $a0. The function code is then compiled. The final section of the code is to return to the calling code, any result is moved to $v0 the previous frame is restored into the $fp and the return address jumped to. Below is a simple function example in mips with annotations.

```
                       function1:        (1)
                       move $t2 $a0      (2)
                       li $a0 40         (3)
                       li $v0 9          (4)
                       syscall           (6)
                       move $t0 $fp      (5)
                       move $fp $v0      (6)
                       sw $t0 0($fp)     (7)
                       sw $ra 4($fp)     (8)
                       sw $a1 8($fp)     (9)
int times(int n, int d)lw $t1 0($t2)     (10)
{                      sw $t1 12($fp)
  return n * d;        lw $t1 4($t2)
}                      sw $t1 16($fp)
                       lw $t1 12($fp)    (11)
                       lw $t2 16($fp)
                       mult $t1 $t2
                       mflo $t0
                       sw $t0 20($fp)
                       lw $v0 20($fp)    (12)
                       lw $t0 4($fp)     (13)
                       lw $fp 0($fp)     (14)
                       jr $t0            (15)
```

1. Function label
2. Move argument pointer into tempory register
3. set the number of bytes required to allocate
4. set the syscode for memory allocation
5. Move the old frame pointer into a temporary register
6. Save the new frame pointer
7. Put the previous frame pointer in the frame
8. Put the return adderess in the frame
9. The enclosing scope is passed in $a1 so store this in the frame
10. for both arguments load them from the arg pointer and store in the frame
11. Do the function body
12. store the return value in $v0
13. load the return address

14. restore the previous frame pointer
15. jump to the return address

5.0.2 Enviroment

In order for the location of tokens and TAC registers to be found they are stored in an environment. The environment is made up of several frames each of which store information about a different scope in the program. Each frame holds the association between either TAC registers or tokens and there memory locations.

5.0.3 Closures

In order to have closures we need to define a pair of program code and environment. The structure for a closure in MIPS is using a 8 byte object the first word is the address of the function code and the second is the address where the frame the closure was defined in is stored. When a closure is called the enclosing frame is placed in $a1 and the program code is jumped to. Below is an example of the definition of a closure.

```
li $a0 8               Allocate 2 Words
li $v0 9
syscall
la $t1 function0       Load the address of the code
sw $t1 0($v0)          Save the address in the first place
sw $fp 4($v0)          Save the current frame pointer in the other
```

Then when the closure is called the following code is generated.

```
lw $t0 16($fp)         The closure object is loaded
lw $t1 0($t0)          The code address is loaded
lw $a1 4($t0)          The enclosing frame is loaded into $a0
jal $t1                The code is jumped to
```

6 Testing

Both the compiler and the interpretter had tests writtern for them, these are run by a python script "test.py" and were run after each change was made to check nothing had been broken. The interpreter test simply runs the code with an example and checks the output has the correct value. In order to automatically test the compiler I had to write a custom error handler script for the spim interpreter which calls the main function then prints the returned value out, with this the spim command line tool can be used to check the values returned by the mips code are correct.

6.1 Interpretation

The interpreter was run with various test cases, the results from some of these cases are shown below along with the result returned from the interpreter.

6.1.1 Math Test - test_math.c

```
int main()
{
    return 8 * 2 - 2;
}
```

RESULT - 14

6.1.2 Simple Test - test_simple.c

```
int main()
{
    int y;
    int x = 4;
    y = 4;
    return x + y;
}
```

RESULT - 8

6.1.3 If Else Test - test_if_else.c

```
int main()
{
    if (1 > 4) {
      return 4;
    } else if (2 > 1) {
      return 3;
    }

    return 8;
}
```

RESULT - 3

6.1.4 While Test - test_while.c

```
int main()
{
    int x = 0;
    while (x < 5)
    {
      x = x + 1;
    }

    return x;
}
```

RESULT - 5

6.1.5 Function Test - test_function.c

```
int test()
{
  return 4;
}

int main()
{
  return test();
}
```

RESULT - 4

6.1.6 Function With Arguments Test - test_function_args.c

```
int times(int n, int d)
{
  return n * d;
}

int main()
{
  return times(3, 2);
}
```

RESULT - 6

6.1.7 Inner Function Test - test_innerfunc.c

```
int times2(int n) {
  int times(int n, int m) {
    return n * m;
  }
  return times(n, 2);
}

int main()
{
  return times2(3);
}
```

RESULT - 6

6.1.8 Twice Test - test_twice.c

```
function twice(function f) {
  int g(int x) { return f(f(x)); }
  return g;
}

void main()
{
  int addten(int n) {return n + 10;}
  return twice(addten)(2);
}
```

RESULT - 22

6.1.9 Cplus Test - test_cplus.c

```
function cplus(int a) {
  int cplusa(int b) { return a+b; }
  return cplusa;
}

int main()
{
  return cplus(5)(2);
}
```

RESULT - 7

6.1.10 Factorial Test - test_fact.c

```
int fact(int n) {
    int inner_fact(int n, int a) {
      if (n==0) return a;
        return inner_fact(n-1,a*n);
      }
    return inner_fact(n,1);
}
```

```
int main()
{
  return fact(4);
}
```

RESULT - 24

6.2 Compiler

In order to test both the TAC and MIPS compilation stages various –C programs were run though the compiler and the result they leave in the return register was checked with the expected value. In order to make this simpler a new exception handler was writtern for the spim interpreter which prints the value left in $v0 after the user code is run. The compilation is done by using "./mycc -c FileName" and then running it in spim using "spim -exception_file testExceptionHandler.s -file Output/test.asm". Given this here are a few example test cases there TAC and MIPS code and result value.

6.2.1 Math Test - test_math.c

```
int main()
{
    return 8 * 2 - 2;
}
```

```
DEFINE CLOSURE _0
_0:
NEW FRAME 0 arg 0 loc 5 temp
RETURN 14
FUNCTION END
```

```
main:
li $a0 16
li $v0 9
syscall
move $fp $v0
sw $ra 4($fp)
li $a0 8
li $v0 9
syscall
la $t1 function0
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
move $a1 $fp
jal function0
lw $t0 4($fp)
```

```
jr $t0
function0:
move $t2 $a0
li $a0 32
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $v0 14
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

RESULT - 14

6.2.2 Simple Test - test_simple.c

```
int main()
{
    int y;
    int x = 4;
    y = 4;
    return x + y;
}
```

```
DEFINE CLOSURE _0
_0:
NEW FRAME 0 arg 3 loc 4 temp
RETURN 8
FUNCTION END
```

```
main:
li $a0 16
li $v0 9
syscall
move $fp $v0
sw $ra 4($fp)
li $a0 8
li $v0 9
syscall
la $t1 function0
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
```

```
move $a1 $fp
jal function0
lw $t0 4($fp)
jr $t0
function0:
move $t2 $a0
li $a0 40
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $v0 8
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

RESULT - 8

6.2.3 If Else Test - test_if_else.c

```
int main()
{
    if (1 > 4) {
      return 4;
    } else if (2 > 1) {
      return 3;
    }

    return 8;
}
```

```
DEFINE CLOSURE _0
_0:
NEW FRAME 0 arg 0 loc 9 temp
r3 := 0
IF NOT r3 GOTO 1
RETURN 4
GOTO 2
LABEL 1: r7 := 1
IF NOT r7 GOTO 3
RETURN 3
LABEL 3: LABEL 2: RETURN 8
FUNCTION END
```

```
main:
li $a0 16
li $v0 9
syscall
move $fp $v0
sw $ra 4($fp)
li $a0 8
li $v0 9
syscall
la $t1 function0
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
move $a1 $fp
jal function0
lw $t0 4($fp)
jr $t0
function0:
move $t2 $a0
li $a0 48
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $t0 0
sw $t0 12($fp)
lw $t2 12($fp)
beq $t2 $zero label1
li $v0 4
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
j label2
label1:
li $t0 1
sw $t0 16($fp)
lw $t2 16($fp)
beq $t2 $zero label3
li $v0 3
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
label3:
```

```
label2:
li $v0 8
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

RESULT - 3

6.2.4 While Test - test_while.c

```
int main()
{
     int x = 0;
     while (x < 5)
     {
       x = x + 1;
     }

     return x;
}
```

```
DEFINE CLOSURE _0
_0:
NEW FRAME 0 arg 2 loc 8 temp
x := 0
GOTO 1
LABEL 2: r4 := x + 1
x := r4
LABEL 1: r7 := x < 5
IF r7 GOTO 2
RETURN x
FUNCTION END
```

```
main:
li $a0 16
li $v0 9
syscall
move $fp $v0
sw $ra 4($fp)
li $a0 8
li $v0 9
syscall
la $t1 function0
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
move $a1 $fp
```

```
jal function0
lw $t0 4($fp)
jr $t0
function0:
move $t2 $a0
li $a0 52
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $t0 0
sw $t0 12($fp)
j label1
label2:
lw $t1 12($fp)
li $t2 1
add $t0 $t1 $t2
sw $t0 16($fp)
lw $t0 16($fp)
sw $t0 12($fp)
label1:
lw $t1 12($fp)
li $t2 5
sltu $t0 $t1 $t2
sw $t0 20($fp)
lw $t2 20($fp)
bne $t2 $zero label2
lw $v0 12($fp)
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

RESULT - 5

6.2.5 Function Test - test_function.c

```
int test()
{
  return 4;
}

int main()
{
  return test();
}

DEFINE CLOSURE _1
DEFINE CLOSURE _0
_1:
NEW FRAME 0 arg 0 loc 1 temp
RETURN 4
FUNCTION END
_0:
NEW FRAME 0 arg 0 loc 2 temp
ALLOCATE PARAMS 0
CALL _1 FROM SCOPE 1
RETURN result
FUNCTION END

main:
li $a0 20
li $v0 9
syscall
move $fp $v0
sw $ra 4($fp)
li $a0 8
li $v0 9
syscall
la $t1 function1
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
li $a0 8
li $v0 9
syscall
la $t1 function0
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 16($fp)
move $a1 $fp
jal function0
lw $t0 4($fp)
jr $t0
function1:
move $t2 $a0
li $a0 16
li $v0 9
syscall
move $t0 $fp
move $fp $v0
```

```
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $v0 4
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
function0:
move $t2 $a0
li $a0 20
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $a0 0
li $v0 9
syscall
move $a0 $v0
move $t7 $fp
lw $t7 8($t7)
lw $t0 12($t7)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $v0 $v0
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

RESULT - 4

6.2.6 Function With Arguments Test - test_function_args.c

```
int times(int n, int d)
{
  return n * d;
}

int main()
{
  return times(3, 2);
}
```

DEFINE CLOSURE _1

```
DEFINE CLOSURE _0
_1:
NEW FRAME 2 arg 2 loc 3 temp
LOAD PARAM n
LOAD PARAM d
r3 := n * d
RETURN r3
FUNCTION END
_0:
NEW FRAME 0 arg 0 loc 4 temp
r4 := 3
r5 := 2
ALLOCATE PARAMS 2
SAVE PARAM r4
SAVE PARAM r5
CALL _1 FROM SCOPE 1
RETURN result
FUNCTION END

main:
li $a0 20
li $v0 9
syscall
move $fp $v0
sw $ra 4($fp)
li $a0 8
li $v0 9
syscall
la $t1 function1
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
li $a0 8
li $v0 9
syscall
la $t1 function0
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 16($fp)
move $a1 $fp
jal function0
lw $t0 4($fp)
jr $t0
function1:
move $t2 $a0
li $a0 40
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
lw $t1 0($t2)
sw $t1 12($fp)
lw $t1 4($t2)
sw $t1 16($fp)
lw $t1 12($fp)
lw $t2 16($fp)
mult $t1 $t2
mflo $t0
sw $t0 20($fp)
lw $v0 20($fp)
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
function0:
move $t2 $a0
li $a0 28
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $t0 3
sw $t0 12($fp)
li $t0 2
sw $t0 16($fp)
li $a0 8
li $v0 9
syscall
move $a0 $v0
lw $t0 12($fp)
sw $t0 0($v0)
addi $v0 $v0 4
lw $t0 16($fp)
sw $t0 0($v0)
addi $v0 $v0 4
move $t7 $fp
lw $t7 8($t7)
lw $t0 12($t7)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $v0 $v0
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

RESULT - 6

6.2.7 Inner Function Test - test_innerfunc.c

```
int times2(int n) {
  int times(int n, int m) {
    return n * m;
  }
  return times(n, 2);
}

int main()
{
  return times2(3);
}
```

```
DEFINE CLOSURE _1
_1:
NEW FRAME 1 arg 1 loc 4 temp
LOAD PARAM n
DEFINE CLOSURE _2
r4 := n
r5 := 2
ALLOCATE PARAMS 2
SAVE PARAM r4
SAVE PARAM r5
CALL _2 FROM SCOPE 0
RETURN result
FUNCTION END
DEFINE CLOSURE _0
_2:
NEW FRAME 2 arg 2 loc 3 temp
LOAD PARAM n
LOAD PARAM m
r3 := n * m
RETURN r3
FUNCTION END
_0:
```

```
NEW FRAME 0 arg 0 loc 3 temp
r7 := 3
ALLOCATE PARAMS 1
SAVE PARAM r7
CALL _1 FROM SCOPE 1
RETURN result
FUNCTION END

main:
li $a0 16
li $v0 9
syscall
move $fp $v0
sw $ra 4($fp)
li $a0 8
li $v0 9
syscall
la $t1 function1
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
move $a1 $fp
jal function0
lw $t0 4($fp)
jr $t0
function1:
move $t2 $a0
li $a0 36
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
lw $t1 0($t2)
sw $t1 12($fp)
li $a0 8
li $v0 9
syscall
la $t1 function2
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 16($fp)
lw $t0 12($fp)
sw $t0 20($fp)
```

```
li $t0 2
sw $t0 24($fp)
li $a0 8
li $v0 9
syscall
move $a0 $v0
lw $t0 20($fp)
sw $t0 0($v0)
addi $v0 $v0 4
lw $t0 24($fp)
sw $t0 0($v0)
addi $v0 $v0 4
lw $t0 16($fp)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $v0 $v0
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
li $a0 8
li $v0 9
syscall
la $t1 function0
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 28($fp)
function2:
move $t2 $a0
li $a0 40
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
lw $t1 0($t2)
sw $t1 12($fp)
lw $t1 4($t2)
sw $t1 16($fp)
lw $t1 12($fp)
lw $t2 16($fp)
mult $t1 $t2
mflo $t0
sw $t0 20($fp)
```

```
lw $v0 20($fp)
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
function0:
move $t2 $a0
li $a0 24
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $t0 3
sw $t0 12($fp)
li $a0 4
li $v0 9
syscall
move $a0 $v0
lw $t0 12($fp)
sw $t0 0($v0)
addi $v0 $v0 4
move $t7 $fp
lw $t7 8($t7)
lw $t0 12($t7)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $v0 $v0
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

RESULT - 6

6.2.8 Twice Test - test_twice.c

```
function twice(function f) {
  int g(int x) { return f(f(x)); }
  return g;
}

void main()
{
  int addten(int n) {return n + 10;}
  return twice(addten)(2);
}

DEFINE CLOSURE _1
_1:
NEW FRAME 1 arg 1 loc 1 temp
LOAD PARAM f
DEFINE CLOSURE _2
RETURN 2
FUNCTION END
DEFINE CLOSURE _0
_2:
NEW FRAME 1 arg 1 loc 5 temp
LOAD PARAM x
r1 := x
ALLOCATE PARAMS 1
SAVE PARAM r1
CALL _0 FROM SCOPE 1
r2 := result
ALLOCATE PARAMS 1
SAVE PARAM r2
CALL _0 FROM SCOPE 1
RETURN result
FUNCTION END
_0:
NEW FRAME 0 arg 0 loc 6 temp
DEFINE CLOSURE _3
r8 := 3
ALLOCATE PARAMS 1
SAVE PARAM r8
CALL _1 FROM SCOPE 1
r9 := result
r10 := 2
ALLOCATE PARAMS 1
SAVE PARAM r10
CALL r9 FROM SCOPE 0
RETURN result
FUNCTION END
_3:
NEW FRAME 1 arg 1 loc 3 temp
LOAD PARAM n
r7 := n + 10
RETURN r7
FUNCTION END

main:
li $a0 16
li $v0 9
syscall
move $fp $v0
sw $ra 4($fp)
li $a0 8
li $v0 9
syscall
la $t1 function1
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
move $a1 $fp
jal function0
lw $t0 4($fp)
jr $t0
function1:
move $t2 $a0
li $a0 24
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
lw $t1 0($t2)
sw $t1 12($fp)
li $a0 8
li $v0 9
syscall
la $t1 function2
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 16($fp)
lw $v0 16($fp)
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
li $a0 8
li $v0 9
syscall
la $t1 function0
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 20($fp)
function2:
```

```
move $t2 $a0
li $a0 40
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
lw $t1 0($t2)
sw $t1 12($fp)
lw $t0 12($fp)
sw $t0 16($fp)
li $a0 4
li $v0 9
syscall
move $a0 $v0
lw $t0 16($fp)
sw $t0 0($v0)
addi $v0 $v0 4
move $t7 $fp
lw $t7 8($t7)
lw $t0 12($t7)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $t0 $v0
sw $t0 20($fp)
li $a0 4
li $v0 9
syscall
move $a0 $v0
lw $t0 20($fp)
sw $t0 0($v0)
addi $v0 $v0 4
move $t7 $fp
lw $t7 8($t7)
lw $t0 12($t7)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $v0 $v0
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
function0:
```

```
move $t2 $a0
li $a0 36
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $a0 8
li $v0 9
syscall
la $t1 function3
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
lw $t0 12($fp)
sw $t0 16($fp)
li $a0 4
li $v0 9
syscall
move $a0 $v0
lw $t0 16($fp)
sw $t0 0($v0)
addi $v0 $v0 4
move $t7 $fp
lw $t7 8($t7)
lw $t0 12($t7)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $t0 $v0
sw $t0 20($fp)
li $t0 2
sw $t0 24($fp)
li $a0 4
li $v0 9
syscall
move $a0 $v0
lw $t0 24($fp)
sw $t0 0($v0)
addi $v0 $v0 4
lw $t0 20($fp)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
```

```
move $v0 $v0
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
function3:
move $t2 $a0
li $a0 32
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
lw $t1 0($t2)
sw $t1 12($fp)
lw $t1 12($fp)
li $t2 10
add $t0 $t1 $t2
sw $t0 16($fp)
lw $v0 16($fp)
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

RESULT - 22

6.2.9 Cplus Test - test_cplus.c

```
function cplus(int a) {
  int cplusa(int b) { return a+b; }
  return cplusa;
}

int main()
{
  return cplus(5)(2);
}
```

```
DEFINE CLOSURE _1
_1:
NEW FRAME 1 arg 1 loc 1 temp
LOAD PARAM a
DEFINE CLOSURE _2
RETURN 2
FUNCTION END
DEFINE CLOSURE _0
```

```
_2:
NEW FRAME 1 arg 1 loc 3 temp
LOAD PARAM b
DEFINED IN 1 r1 := a
r3 := r1 + b
RETURN r3
FUNCTION END
_0:
NEW FRAME 0 arg 0 loc 6 temp
r5 := 5
ALLOCATE PARAMS 1
SAVE PARAM r5
CALL _1 FROM SCOPE 1
r6 := result
r7 := 2
ALLOCATE PARAMS 1
SAVE PARAM r7
CALL r6 FROM SCOPE 0
RETURN result
FUNCTION END

main:
li $a0 16
li $v0 9
syscall
move $fp $v0
sw $ra 4($fp)
li $a0 8
li $v0 9
syscall
la $t1 function1
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
move $a1 $fp
jal function0
lw $t0 4($fp)
jr $t0
function1:
move $t2 $a0
li $a0 24
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
lw $t1 0($t2)
sw $t1 12($fp)
li $a0 8
li $v0 9
syscall
la $t1 function2
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 16($fp)
lw $v0 16($fp)
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
li $a0 8
li $v0 9
syscall
la $t1 function0
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 20($fp)
function2:
move $t2 $a0
li $a0 32
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
lw $t1 0($t2)
sw $t1 12($fp)
move $t7 $fp
lw $t7 8($t7)
lw $t0 12($t7)
sw $t0 16($fp)
lw $t1 16($fp)
lw $t2 12($fp)
add $t0 $t1 $t2
sw $t0 20($fp)
lw $v0 20($fp)
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

```
function0:
move $t2 $a0
li $a0 36
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $t0 5
sw $t0 12($fp)
li $a0 4
li $v0 9
syscall
move $a0 $v0
lw $t0 12($fp)
sw $t0 0($v0)
addi $v0 $v0 4
move $t7 $fp
lw $t7 8($t7)
lw $t0 12($t7)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $t0 $v0
sw $t0 16($fp)
li $t0 2
sw $t0 20($fp)
li $a0 4
li $v0 9
syscall
move $a0 $v0
lw $t0 20($fp)
sw $t0 0($v0)
addi $v0 $v0 4
lw $t0 16($fp)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $v0 $v0
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

RESULT - 7

6.2.10 Factorial Test - test_fact.c

```
int fact(int n) {
    int inner_fact(int n, int a) {
      if (n==0) return a;
        return inner_fact(n-1,a*n);
      }
    return inner_fact(n,1);
}


int main()
{
  return fact(4);
}
```

```
DEFINE CLOSURE _1
_1:
NEW FRAME 1 arg 1 loc 4 temp
LOAD PARAM n
DEFINE CLOSURE _2
r12 := n
r13 := 1
ALLOCATE PARAMS 2
SAVE PARAM r12
SAVE PARAM r13
CALL _2 FROM SCOPE 0
RETURN result
FUNCTION END
DEFINE CLOSURE _0
_2:
NEW FRAME 2 arg 2 loc 12 temp
LOAD PARAM n
LOAD PARAM a
r3 := n == 0
IF NOT r3 GOTO 1
RETURN a
LABEL 1: r7 := n - 1
r10 := a * n
ALLOCATE PARAMS 2
SAVE PARAM r7
SAVE PARAM r10
CALL _2 FROM SCOPE 1
RETURN result
FUNCTION END
_0:
```

```
NEW FRAME 0 arg 0 loc 3 temp
r15 := 4
ALLOCATE PARAMS 1
SAVE PARAM r15
CALL _1 FROM SCOPE 1
RETURN result
FUNCTION END

main:
li $a0 16
li $v0 9
syscall
move $fp $v0
sw $ra 4($fp)
li $a0 8
li $v0 9
syscall
la $t1 function1
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 12($fp)
move $a1 $fp
jal function0
lw $t0 4($fp)
jr $t0
function1:
move $t2 $a0
li $a0 36
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
lw $t1 0($t2)
sw $t1 12($fp)
li $a0 8
li $v0 9
syscall
la $t1 function2
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 16($fp)
lw $t0 12($fp)
sw $t0 20($fp)
```

```
li $t0 1
sw $t0 24($fp)
li $a0 8
li $v0 9
syscall
move $a0 $v0
lw $t0 20($fp)
sw $t0 0($v0)
addi $v0 $v0 4
lw $t0 24($fp)
sw $t0 0($v0)
addi $v0 $v0 4
lw $t0 16($fp)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $v0 $v0
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
li $a0 8
li $v0 9
syscall
la $t1 function0
sw $t1 0($v0)
sw $fp 4($v0)
sw $v0 28($fp)
function2:
move $t2 $a0
li $a0 76
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
lw $t1 0($t2)
sw $t1 12($fp)
lw $t1 4($t2)
sw $t1 16($fp)
lw $t1 12($fp)
li $t2 0
sub $t0 $t1 $t2
sltu $t0 $zero $t0
xori $t0 $t0 1
```

```
sw $t0 20($fp)
lw $t2 20($fp)
beq $t2 $zero label1
lw $v0 16($fp)
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
label1:
lw $t1 12($fp)
li $t2 1
sub $t0 $t1 $t2
sw $t0 24($fp)
lw $t1 16($fp)
lw $t2 12($fp)
mult $t1 $t2
mflo $t0
sw $t0 28($fp)
li $a0 8
li $v0 9
syscall
move $a0 $v0
lw $t0 24($fp)
sw $t0 0($v0)
addi $v0 $v0 4
lw $t0 28($fp)
sw $t0 0($v0)
addi $v0 $v0 4
move $t7 $fp
lw $t7 8($t7)
lw $t0 16($t7)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $v0 $v0
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
function0:
move $t2 $a0
li $a0 24
li $v0 9
syscall
move $t0 $fp
move $fp $v0
sw $t0 0($fp)
sw $ra 4($fp)
sw $a1 8($fp)
li $t0 4
sw $t0 12($fp)
li $a0 4
li $v0 9
syscall
move $a0 $v0
lw $t0 12($fp)
sw $t0 0($v0)
addi $v0 $v0 4
move $t7 $fp
lw $t7 8($t7)
lw $t0 12($t7)
lw $t1 0($t0)
lw $a1 4($t0)
jal $t1
move $v0 $v0
lw $t0 4($fp)
lw $fp 0($fp)
jr $t0
```

RESULT - 24

References

[1] Gnu basic blocks. https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html. Accessed: 06-12-2016.

[2] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[3] Luca Cardelli. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 208–217, New York, NY, USA, 1984. ACM.

[4] Michael Matz. Design and implementation of a graph coloring register allocator for gcc. In *GCC Developers Summit*, pages 151–170, 2003.