Assignment/Problem statement 5 Recitation

Generating Assembly Code

TDT4205 Compiler Construction

NTNU

2019

Some Exercise Guidelines

- Please submit a **pdf** file, even if it is a small solution and not a text file (huge inconvenience when marking several solutions).
- There will be a skeleton folder inside which all the required files for the exercises will be available (all exercises have this format). For the theory part, the pdf file with the solution should be zipped along with this skeleton folder (containing your solution of course)
- **Please Please Please** after every submission, check if your submission exists on Blackboard. You will also get a confirmation email of your submission. So submit 1 hour before the actual deadline

Exercise 5? What to do

- The previous exercises look at ensuring the required info is available to perform the generation of assembly code. So the syntax tree, symbol table, etc (such as stringlist) have this data.
- This part of the exercise looks at the generation of assembly code
  - Note that there are two parts to this exercise since it is very large.
  - First part deals with simpler parts so better to finish off that to compile simple programs

Part I

- Part I should tackle the following problems
  - Global String Table
  - Global variables
  - Functions
  - Function parameters
  - Arithmetic expressions
  - Arithmetic statements
  - Assignment statements
  - print statements
  - return statements

Part II

- Part II should tackle the following problems
  - Local Variables
  - Function calls
  - Conditionals (if and relations)
  - While loops
  - Continue (Null statement)

Let's dissect a Hello World

```
def hello()
begin
        print "hello,world"
        return 0
end
```

The important parts of the assembly code

```
.section .rodata
strout: .string "%s"
STR0: .string "Hello, world!"

.globl main
.section .text
_hello:
pushq %rbp
movq %rsp, %rbp
movq $STR0, %rsi
movq $strout,%rdi
call printf
movq $'\n',%rdi
call putchar
movq $0,%rax
leave
ret
```

The important parts of the assembly code

```
.section .rodata                    <-- Read only data section
strout: .string "%s"                <-- String to use for printing strings
STRO: .string "Hello, world!"       <-- String from source program numbered '0'

.globl main                         <-- Suggests the main function
.section .text                      <-- Text section for the assemby instructions
_hello:                             <-- Function name prefixed to prevent collision
pushq %rbp                              with syslib printf
movq %rsp, %rbp
movq $STR0, %rsi
movq $strout,%rdi
call printf
movq $'\n/',%rdi
call putchar
movq $0,%rax
leave
ret
```

The important parts of the assembly code

```
.section .rodata
strout: .string "%s"
STRO: .string "Hello, world!"

.globl main
.section .text
_hello:
pushq %rbp                    <-- Stack Frame setup for the function
movq %rsp, %rbp
movq $STR0, %rsi
movq $strout,%rdi
call printf
movq $'\n/',%rdi
call putchar
movq $0,%rax
leave
ret
```

The important parts of the assembly code

```
.section .rodata
strout: .string "%s"
STRO: .string "Hello, world!"

.globl main
.section .text
_hello:
pushq %rbp
movq %rsp, %rbp                    From the "print" statement
movq $STR0, %rsi            <--- Place address of output data
movq $strout,%rdi           <--- place the address of string output constant
call printf                 <--- Leave it to "printf" to put stuff on screen
movq $'\n/',%rdi            <--- Last item printed prepare new item
call putchar                <--- Output new line
movq $0,%rax
leave
ret
```

The important parts of the assembly code

```
.section .rodata
strout: .string "%s"
STRO: .string "Hello, world!"

.globl main
.section .text
_hello:
pushq %rbp
movq %rsp, %rbp
movq $STR0, %rsi
movq $strout,%rdi
call printf
movq $'\n/',%rdi
call putchar
movq $0,%rax                    From the "return" statement
leave                     <-- set up 0 (from the program) as the return value
ret                       <-- remove the stack frame
                          <-- return to where the call came from
```

Things not covered in the above slide set

- The main function
- Global variables? they need mutable memory
- Arguments?
- Expressions
- Assignments
- Expressions in print statements

Main Function

- Remember calling convention from lecture slide on instruction set
  - First 6 args go into registers %rdi %rsi %rdx %rcx %r8 %r9
  - Further args in the stack
  - Stack will need 16 byte alignment
- All args are 64-bit integers
- Main is called differently from the shell
  - 1st argument is the command line args in text
  - 2nd argument is a pointer to a list of char-pointers

A generic 'main' for VSL programs

- At runtime this has to be done:
  - Find the count of arguments
  - If there are some translate them from text to numbers
  - Put them in the right places for an ordinary call
  - Call the 1st function defined in the VSL source program
  - Take the return value from that and return it to the calling shell
  - Return to shell

Generate Main is supplied

- So a generate_main will be supplied. This will simply generate assembly to point to the symbol_t that is the first defined function in the program.
- It expects the global names to be prefixed with the _ in the generated assembly
- It will fail if the shell doesn't provide an argument count that doesn't match that of the starting function in the source program.
- A hard coded main to prevent the assembler from giving errors is also available. Replace that part such that you start off with the symbol_t you supply.

Generating Stringtable

- Also a generate_stringtable function will be provided which prints the following:

```
.section .rodata
intout: .string "%ld"
strout: .string "%s"
errout: .string "Wrong number of arguments"
```

- errout is only needed by the main
- intout and strout are handy for printing numbers and strings when translating "print" statements
- Read-only data section is still missing from the source. Dump them here with numbered labels like STR0:, STR1:,...

Mutable Memory for Global Variables

For global variables you need mutable memory. What can be done for this is as follows:

- Emit a ".section .data" (mutable)
- Put labels under it for the global vars, such as "_x:" for variable "x"
- Place a 64-bit zero value at that address, for the program to change at run time (the 'zero' directive takes a byte count) i.e. : _x: .zero 8
- In this way reference to global variable 'x' is translated as an access to '_x'

Arguments

- First few of these reside in registers For convenient reference the call convention order is placed in a static string array 'record[6]' which contains strings with the register names in order
- For function calls these registers will change values
- The copies of the arguments can well be placed on stack as the first thing a function does so that they've found an address %rbp + 8*argument_index

Stack Alignment

- Accessing arguments relative to the base pointer %rbp from bottom up.
- Every arg and local consumes 8 bytes. pushing an odd number create stack misalignment
- Pad it with 8 bytes (prevents crashing of generated system calls ex: printf)

Expressions

- Treat the process as a stack machine when generating code
- Let %rax have the results. Numbers translate into setting them in %rax. Variables translate to copying their contents
- Operations translated recursively
  - Recursively generate subexpression 1 (put result in %rax)
  - Push result
  - Step 1 for subexpression 2
  - Combine result with top of stack element to obtain result of operation
  - Remove the temporary Result of subex 1 from stack
  - Result is in %rax and stack is restored
- Be aware of the multiply and divide instructions

Assignments

Going by this scheme, assignment is only about generating the code for the RHS expression and moving the result in %rax to the location of the assignment destination

Printing I

- It is a list to contain strings numbers identifiers and expressions. Broken down as
  - Generate code to print 1st element
  - Same as above for the second and so forth
  - ...
  - Code to print new line
- the effect of print statement is a concatenation of these

Printing II

Which is to iterate over the list of print items as follows:

- Strings: setup and call printf with strout and the string
- Numbers: setup and call printf with intout and number
- Identifiers: setup and call printf with intout and the contents of the identified address
- Expressions: Generate the expression, setup and call printf with intout and the contents of %rax

Next things to tackle:

- Local Variables
- Function calls
- Conditionals
- Loops
- Continue

Local Variables

- Local variables are not accessed in the same mechanism as the global variables. They go on the run-time stack.
- Their sequence number can be used to find offset from the base pointer
- Begin a function by creating space for them on the stack
  - They were counted in the process of generating the symbol table
  - (Note the 16 byte alignment)
- Otherwise these can go into expressions in the same manner as global variables

Function calls

- They appear in expressions
- Generating them involves following the discussed calling convention i.e:
  - Put first 6 arguments in designated registers
  - additional arguments go into the stack
  - Call the function
  - Restore stack with result in %rax

Conditionals (IF and relations)

- Relation is generate in same manner as arithmetic expressions
- Recursively generate code to evaluate the left expression leaving result in %rax
- Put result on the stack
- Generate code to evaluate the right expression
- Get previous result from the stack
- Compare and jump as needed

The process of jumping

- The expression: "if (a=b) then A else B" can be turned to

```
evaluate a
evaluate b
compare
jump-not-equal ELSE
A
jump ENDIF
ELSE:
B
ENDIF:
(rest of the program)
```

- This needs a numbering scheme and since the conditional statements can be nested, one would need a stack to push and pop the counter values from the numbering scheme. This will track the nesting.

Loops

These are also treated like conditionals; while(condition)expression turns to:

```
WHILELOOP:
evaluate condition
jump-false ENDWHILE
expression
jump WHILELOOP:
ENDWHILE:
(Remainder of the program)
```

Treat the loops in the same way as IFs for the nesting i.e. have a numbering scheme. Use a separate stack for them.

Continue

- Continue-Statements skip directly to condition evaluation of the while loop
- If one considers a shared counting scheme for WHILEs and IFs then the enclosing construct could be an IF
- With separate stacks, the index of the enclosing while loop is on the stack top of the while_stack

The End

- That's it for the Compiler Construction Course