NTNU – Trondheim
Norwegian University of
Science and Technology

Recitation lecture: problem set 5

Correction about keys

- You were told earlier to use symbol name opt. with some mangling as keys in symbol table, newest skeleton uses sequence number as key in local table
  - Other solutions are not *wrong*, just more convoluted
  - Hopefully, this better explains the motivation for adding sequence numbers! (sorry..)

PS 5: Code generation (pt 1)

- Code generation without control structures
  - Functions
  - Print statements
  - Arithmetic expressions
  - Assignment statements
  - Global string table
  - Global and local variables
- If, while are implemented in PS 6
- New .vsl files for ps 5 should generate an executable program.

PS 5: Code generation (pt 1)

- Tasks can still be be done on M1 Macbooks, but you won’t be able to assemble and run your generated code
  - Use remote machine: https://i.ntnu.no/wiki/-/wiki/English/SSH
  - QEMU emulator VM
  - Rosetta2
- If you haven’t used them yet, two valuable tools for debugging C programs (and your generated asm)
  - GDB for stepping and breakpoints
  - Valgrind for memory checks and traces

x86-64 (x64) assembly

- 16r egisters: rax, rbx, rcx, rdx, rdi, rsi, rbp (base pointer), rsp (stack pointer), r8-15
- General syntax: **op** src, dest
  - Arithmetic operations resemble a stack machine: source operand applied to value in destination
- Comments: GAS (GNU assembler) accepts # line comments and **/* */** block comments.
  - nasm uses semicolon, GCC accepts double slashes if invoked with the preprocessor (*.S or *.sx)
- Helpful x64 cheatsheet, advice you to keep it available
  - I heavily rely on this myself, so I can’t answer of the top of my head, this is probably where I’ll review first anyway

x64 assembly

- Addressing modes
  - Register: **%R**
  - Immediate value: **$N** (suffix **0x** for hex, **0b** for for binary)
  - Memory: **(%R)** (%R hold memory address: mem[reg[R]])
    - Displacement: **D(%R)** (mem[reg[R]+D]
    - General form: **D(Rb, Ri, S)** (mem[reg[Rb] + S * reg[Ri])

NTNU – Trondheim
Norwegian University of
Science and Technology

x64 assembly

- **Caution:** Many online examples use the "nasm" assembler. This one uses Intel syntax, which is incompatible with Unix (AT&T) syntax, used by gcc and clang
  - Notably: Order of parameters reversed

x64 assembly

- Sections
  - text: Contains our program code
  - bss: Block Starting Symbol, contains statically allocated, uninitialized data (global variables). Saves object file space as opposed to data section
  - data: Pre-initialized data section
  - rodata: Read-only data, where we will put our strings

Code gen: string data

- We have our global string_list
- Use format strings (printf) as usual
- Create a *data* section declaring all strings
- Also define strout, intout and errout
- Directives **.asciz** and **.string** are synonymous

```
func hello() begin
    print “Hello”, “World!”
end
```

$\longrightarrow$

```
.data
.strout: .asciz “%s”
.intout: .asciz “%ld”
.errout: .asciz “Wrong number of arguments
.STR0: .asciz “Hello”
.STR1: .asciz “World!”
...
```

NTNU – Trondheim
Norwegian University of
Science and Technology

Code gen: global variables

- Like with strings we declare them in a separate section: **.bss**
- All values are 64 bit integers, entire section can be 8 bytes aligned: **.align 8**
- Variables are unitiliatized, only their name need to be declared: **.my_global_var0:**

Code gen: printing

- Special case of calling a std library function, printf

```
movq $.strout, %rdi
movq $.STR0, %rsi
call printf
movq $.STR1, %rsi
call printf
movq $'\n', %rdi
call putchar        // just add that newline
```

```
func hello() begin
    print "Hello", "World!"
end
```

NTNU – Trondheim
Norwegian University of Science and Technology

Code gen: expressions

- For simplicity, we will treat the processor as a stack machine, pushing all intermediate results to the stack
- Again, traverse the AST, writing out correct instructions for each node
- Generally for expressions:
  - Generate lhs of expression, push to stack
  - Generate rhs of expression
  - Pop from stack to an unused register (e.g. r10)
  - Perform operation (e.g. **add %r10, %rax**)

Code gen: calling functions

- Preparing parameters
  - First 6 parameters go in registers: **%rdi, %rsi, %rdx, %rcx, %r8, %r9**
  - Subsequent parameters are pushed onto the stack

```
// call foo(1, 15)
movq   $1, %rdi
movq   $15, %rsi
call     foo            // Push return address and jump to label foo
```

  - .
- Caller saved registers
  - **%rax, %rcx, %rdx, %rdi, %rsi, %rsp** and **%r8-r11** must be pushed to the stack if their value are needed after the call (safe side: always save)
  - Simplification: most are used for arguments, the rest aren't used.
- Name mangling (for real this time)
  - Avoid collisions with internal names. **generate_main** generates a **main** function, but what if you called your function **main** as well?
  - **main** becomes e.g. **_main**, **_vsl_main** or **_lots_of_mangling_why_not_main**

NTNU – Trondheim
Norwegian University of
Science and Technology

Code gen: entering function

- Push %rbp, caller’s BP, move SP to BP
  ```
  pushq %rbp
  movq %rsp, %rbp
  ```
- Push arguments to stack
  - Stack needs to be padded for 16-bytes alignment. Push a zero if we have an odd number of arguments

Code gen: exiting function

- Clear the stack, restore caller SP
  - We saved caller’s SP to %rbp, now return it
- Restore callee saved registers
  - If they were used (probably not
- Return value saved in **rax**

Code gen summary

- Declare strings
- Declare global variables
- Declare functions
- Generate function bodies handling all node types except IF, WHILE, NULL, RELATION

Running our program

- gcc -no-pie prog.S