NTNU – Trondheim
Norwegian University of
Science and Technology

Problem set 5

Purpose of the assembly finger-exercise

- This is meant to give you a feel for handwritten assembly code
- We also happen to have a splendid assembly generator handy, in the form of a C compiler
  - You can generate a solution and start from there
- You can even fix a generated output up to look less generated without understanding what it does
  - It is none of my concern what you do in order to get to grips with the assembler
  - “Skip” the nuts and bolts at your own peril, it will just make it harder to comprehend what your code generator generates in the second part
  - Don’t say I didn’t warn you ;)

Code generation

- The remaining work is to take the structure we have built (tree, name tables) and translate it into assembly
- This could be one massive problem set, but we break it in 2 because:
  - There is the learning curve of figuring out the (idiosyncratic) assembler
  - Some constructs are simple to translate, others are harder
  - If you figure out the simple ones first, there is a straightforward way to continue (PS6 will be "complete the missing constructs")
  - If you can't figure out the simple ones completely, you'll get somewhere to pick it up from along the way without having lost much steam
- ...and finally...
  - Student life is busy, to procrastinate is human
  - Working out your first complete singing and dancing code generator from scratch in 48 hours is a kind of death march I would not wish upon anyone

Part I

- It's high time to make our programs do something
- We can produce some simple effects using only
  - Strings (to see that stuff is happening)
  - Global variables ('cause they're simple to address)
  - Receiving function calls (so that the program can start)
  - Parameters (as a source of values unknown at compile time)
  - Expressions (to calculate stuff in terms of)
    - Numbers
    - Identifiers
    - Arithmetic
  - Some statements:
    - Assignment
    - RETURN
    - PRINT

The example directory

- This time, vsl_programs just contains one stupid program which uses only the above constructs
  - ...so as not to create the expectation that previous example programs should work, they use features we aren't implementing
  - You still have those from before, if you would like to implement more than what is yet asked for

From the bottom up: dissecting “Hello, world”

- Here's a VSL implementation:

```
FUNC hello()
BEGIN
    PRINT "Hello, world!"
    RETURN 0
END
```

NTNU – Trondheim
Norwegian University of
Science and Technology

Here is the salient part of the assembly:

(won't run on its own yet)

```
.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
```

NTNU – Trondheim
Norwegian University of
Science and Technology

Here is the salient part of the assembly:
(won't run on its own yet)

```
.section .rodata                        ← Read-only data section (could have been just 'data', now marked as immutable)
strout: .string "%s "                   ← String to use when printing strings
STR0: .string "Hello, world!"           ← The string from the source program, numbered '0' because it is the only one

.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
```

Here is the salient part of the assembly:

(won't run on its own yet)

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

.globl main                 ← Suggests that we'll define a 'main' function, you won't have to write that yourself
.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
```

Here is the salient part of the assembly:

(won't run on its own yet)

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

.globl main
.section .text                    ← Text section, for the assembly instructions
_hello:                           ← Function name prefixed with “_”, so that it
        pushq   %rbp                 could have been 'printf' w/o colliding with syslibs
        movq    %rsp, %rbp
        movq    $STR0, %rsi
        movq    $strout, %rdi
        call    printf
        movq    $'\n', %rdi
        call    putchar
        movq    $0, %rax
        leave
        ret
```

Here is the salient part of the assembly:

(won't run on its own yet)

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

.globl main
.section .text
_hello:
        pushq   %rbp              ← Set up stack frame (because we're generating a function)
        movq    %rsp, %rbp
        movq    $STR0, %rsi
        movq    $strout, %rdi
        call    printf
        movq    $'\n', %rdi
        call    putchar
        movq    $0, %rax
        leave
        ret
```

Here is the salient part of the assembly:

(won't run on its own yet)

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

.globl main
.section .text
_hello:
        pushq   %rbp
        movq    %rsp, %rbp                From PRINT statement:
        movq    $STR0, %rsi           ← place the address of the output data (arg.2)
        movq    $strout, %rdi         ← place the address of the string-outputter constant (arg. 1)
        call    printf                ← Leave it to 'printf' to put stuff on the screen
        movq    $'\n', %rdi           ← Last print item printed, prepare new line as output
        call    putchar               ← Output the new line
        movq    $0, %rax
        leave
        ret
```

Here is the salient part of the assembly:

(won't run on its own yet)

```
.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             From RETURN statement:
        movq    $0, %rax            ← set up 0 (from the program) as the return value
        leave                       ← Remove the stack frame
        ret                         ← Return to where the call came from
```

Things we didn't cover there

- Where is the 'main' that will launch the program?
- What about global variables, they need mutable memory?
- What about arguments?
- What about expressions?
- What about assignments?
- What about expressions in PRINT statements?

NTNU – Trondheim
Norwegian University of Science and Technology

Main

- Remember the calling convention, from the lecture on our instruction set:
  - First 6 args go in registers %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Further args go on stack
  - Stack needs 16-byte alignment
- All our arguments are 64-bit integers (quadwords)
- 'main' isn't called like that from the shell:
  - Its 1st argument is a count of command line args in text
  - Its 2nd argument is a pointer to a list of char-pointers

A generic 'main' for VSL programs

- At run time, we need to do this:
  - 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

    (Semantics-by-waving-of-hands: we could have defined a magic function name for starting points, but this will do as well)
  - Take the return value from that and return it to the calling shell
  - Return to the shell

NTNU – Trondheim
Norwegian University of
Science and Technology

This is supplied

- One call method is enough to deal with
- There's a generate_main function provided, it'll spew out a piece of assembly that does this conversion based on a pointer to the symbol_t that is the first defined function in the source program
  (to have the name and parameter count)
- It expects global names to be prefixed with _ in the generated assembly
- It fails with an error message if the shell provides an argument count that doesn't match that of the starting function in the source program
- *There is also a hard-coded main which doesn't do anything*
  - This is so that the assembler won't choke on the output of the unmodified framework
  - You should replace that logic with a main generated from a symbol you supply it with

NTNU – Trondheim
Norwegian University of
Science and Technology

For this purpose

- There is also a started 'generate_stringtable' function which defines some constants, it prints:

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

- 'errout' is only needed by main
- The other two are handy for printing numbers and strings when translating PRINT statements
- Read-only data section is still missing all the strings from the source program, dump them here with numbered labels like *STR0:*, *STR1:*, ...

NTNU – Trondheim
Norwegian University of
Science and Technology

Mutable memory for global variables

- This is up to you
- What I do:
  - 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):

    ```
    _x: .zero 8
    ```
- In this way, references to global var. 'x' in the program can be translated as access to the address '_x'

What about arguments?

- At least the 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
- The contents of those registers are necessarily clobbered when generating function calls
- Side-stepping the potential for optimizing functions that don't call further functions (and a small adventure in register allocation methods), copies of the arguments can well be placed on stack as the first thing a function does
  - That way, they're found at address %rbp + 8*argument_index

16-byte stack alignment

- Accessing arguments (and ultimately, locals) from the bottom up, *i.e.* relative to the base pointer %rbp, what's at the top is not so important
- Each arg (and local) consumes 8 bytes, so if you push an odd number of them, the stack is misaligned

A simple solution

- The only external functions we have to interacting with are printing routines
  - printf, puts, however you choose to do it
- These react with peculiar segfaults when passed a mis-aligned stack pointer
- At a tiny instruction overhead, you can insert code which aligns the stack (and reverts it afterwards) just around the calls to external functions

A trickier solution

- Aligning at run-time inserts a lot of instructions that often don’t do anything
- It’s also possible to implement a counting scheme to track when you have odd/even numbers of items on stack, and arrange the alignment statically
- This makes it very easy to overlook some corner case combination of function-call-in-an-expression-in-a-print-statement or such, but it produces fewer instructions
- You’re not *required* to bend your mind this way, but you can take a shot at it if you want to

Expressions

- Again, to avoid tricky register allocation issues, we can systematically treat the processor as a stack machine when generating code
- We've got the accumulator register %rax to contain results
  - Numbers translate into setting them in %rax
  - Variables translate into copying their contents to %rax
- Operations can then be translated recursively:
  - Recursively generate subexpression #1 (puts its result in %rax)
  - Push that result
  - Recursively generate subexpression #2 (obtain its result in %rax)
  - Combine %rax with the top-of-stack element to obtain result of operation
  - Remove the temporary result of subex #1 from stack again
  - Overall result is now computed in %rax, stack is back where it was
- Mind the funny multiply and divide instructions

  (cf. Foil set about x86_64 instruction set)

Assignments

- If this scheme is obeyed, assignment is a matter of
  - Generating code for the r.h.s. expression
  - Moving the result (found in %rax) into the location of the assignment's destination

Printing stuff

- Lists of stuff to print can contain strings, numbers, identifiers and expressions
- As in the hello example, this can be broken into
  - Generate code to print the first element
  - Generate code to print the second element
  - …
  - Generate code to print a new line
- The effect of a print statement is just a concatenation of these

In other words

- Iterate over the list of print items
- For
  - Strings

    setup and call printf with `strout` and the string
  - Numbers

    setup and call printf with `intout` and the number
  - Identifiers

    setup and call printf with `intout` and the contents of the ident. address
  - Expressions

    generate the expression, setup and call printf with `intout` and the contents of %rax

NTNU – Trondheim
Norwegian University of Science and Technology

Why `strout`?

- We *could* just pass strings directly to printf
- printf format codes, however, have no meaning in VSL strings
- If a string can go directly from the source and to the system call, VSL programs could inspect the stack by containing strings with %d, %zu, *etc.* in them
- That's a bit of a far-fetched corner case, but it doesn't cost much to eliminate.

Debugging the output

- Unless you get everything right on the first go, you'll find yourself sifting through your generated assembly code
- 'make' likes to remove intermediate files

  `make program` will remove intermediates if compilation of `program.vsl` fails

  `make program.s` will stop compilation of `program.vsl` after emitting the assembly, so you can read it

NTNU – Trondheim
Norwegian University of
Science and Technology

Debugging the output

- I find it easier to read assembly with indentation of the code in functions
- It can *work* if you just creatively printf stuff all over the place, but input strings grow unreadable
  - Quotemarks '"' that belong in the assembly must be escaped in the compiler source
  - %edi becomes "%%edi" to keep %e from being taken as a format code
  - Spacing is messy, orderly output like

    ```
    _myprog:
        pushq    %rbp
    ```

    may look something like

    ```
    printf ( "_%s:\n\tpushq\t%%rbp\n", (char*) symbol->data );
    ```

    in the compiler source
- This is not necessarily nice to hunt errors in

Debugging the output

- As a confessed macro junkie, I use notations like

  ```
  #define ASM1(op,a) puts ( “\t”#op”\t”#a )
  ```

  so that I can write

  ```
  ASM1(push,%rbp)
  ```

  to get 1-operand instructions like

  ```
  push %rbp
  ```

  in the output,

  ```
  #define ASM2(op,a,b) puts ( "\t"#op"\t"#a", "#b )
  ```

  for turning 2-operand instructions like

  ```
  ASM2(movq,%rax,(%rsp))
  ```

  into

  ```
  movq %rax, (%rsp)
  ```

  and similar, thereby concentrating the unreadability (somewhat) in one place
- These work by turning macro arguments into strings using the preproc. # operation
- There are still some printfs here and there, but many instructions have constant contents
- I find it makes it easier to match the compiler source up with its generated assembly, you decide for yourself

At the end

- This is enough to get a restricted set of programs running
- It's also the point where we in practice generalize from *“programs which produce a result”* into *“programs which produce programs which produce a result”*
- This way to think is a neat trick in itself

Things may fall into place

- We've talked about where this is going on paper
  - If you've seen it from afar, there should be no surprises
- If you haven't written code generators before, it is not uncommon to experience a small epiphany when you get the hang of it
  - That is the TDT4205 easter egg
- If you get on a roll and find it fun, please feel free to go with it
  - PS6 will predictably request implementation of “the rest of the language”, *i.e.* locals, ifs, whiles, continues, function calls...
  - You have all the tools to get a head start on it if you wish