

Tutorial Code Assembly Generation

For the code-generation project, we expect your compiler to produce simple assembly code. We shall expose you to a subset of the x86-64 platform.

Example

Consider the following C-TDS program:

```
class Program {  
  
    int inc(int x) {  
        return x + 1;  
    }  
  
    int get_int() extern;  
  
    void printInt(int x) extern;  
  
    void main() {  
        int y;  
  
        y = inc(get_int());  
        if (y == 1) {  
            printInt(1);  
        } else {  
            printInt(0);  
        }  
    }  
}
```

For the code generation phase of the compiler project, you are encouraged to output simple and inefficient (but correct!) assembly code. This assembly code can assign every variable and temporary to a location on the current stack frame. Every expression value will be loaded from the stack, manipulated using the registers %r10 and %r11, and then the result will be written back to the stack, ready to be used in another expression. Compiling the above C-TDS code using this simple scheme might look like this:

```
inc:  
    enter    $(8*2), $0  
    mov     %rdi, -8(%rbp)  
  
    mov     -8(%rbp), %r10  
    add     $1, %r10  
    mov     %r10, -16(%rbp)
```

```
    mov     -16(%rbp), %rax
    leave
    ret

    .globl main
main:
    enter   $(8 * 6), $0

    call    get_int
    mov     %rax, -8(%rbp)

    mov     -8(%rbp), %rdi
    call    inc
    mov     %rax, -16(%rbp)
    mov     -16(%rbp), %r10
    mov     %r10, -24(%rbp)

    mov     -24(%rbp), %r10
    mov     $1, %r11
    cmp     %r10, %r11
    mov     $0, %r11
    mov     $1, %r10
    cmovle  %r10, %r11
    mov     %r11, -32(%rbp)

    mov     -32(%rbp), %r10
    mov     $1, %r11
    cmp     %r10, %r11
    je      .one

    mov     $1, %rax
    call    printInt
    mov     %rax, -40(%rbp)
    jmp     .one_done

.one:
    mov     $0, %rax
    call    printInt
    mov     %rax, -48(%rbp)

.one_done:
    mov     $0, %rax
    leave
    ret
```

We shall dissect this assembly listing carefully and relate it to the C-TDS code. Note that this is not the only possible assembly of the program; it only serves as an illustration of some techniques you can use in this project phase.

```
inc:
    enter $(8 * 2), $0
    mov %rdi, -8(%rbp)
    ...
    leave
    ret
```

- This is the standard boilerplate code for a function definition. The first line creates a *label* which names the entry point of the function. The following `enter` instruction sets up the [stack frame](#). After the function is done with its actual work, the `leave` instruction restores the stack frame for the caller, and `ret` passes control back to the caller.
- Notice that one of the operands to `enter` is a static arithmetic expression. Such expressions are evaluated by the assembler and converted into constants in the final output.
- `Enter` first saves the callers frame (base) pointer (`%rbp`) unto the stack. Then it sets the frame pointer to the stack pointer (`%rsp`) to set the current frame pointer. `Enter` next allocates *N* bytes (where *N* is the left operand) of stack space to be used for locals and temporaries of the stack frame. It does this by subtracting *N* from `%rsp` (remember that the stack grows downward toward address 0). This space is allocated after the caller's frame (base) pointer is saved to the stack.
- The `mov` instruction moves the 1st argument (passed in `%rdi`) to its place on the stack. The argument occupies the first stack position (stack entries are 8 bytes) after the base pointer (`%rbp`). `0(%rbp)` stores the previous frame's base pointer.

```
mov    -8(%rbp), %r10
add    $1, %r10
mov    %r10, -16(%rbp)
```

- The purpose of `inc(int)` is to add 1 to its argument, and return the result. The first `mov` instruction fetches the argument from the stack and places it in the temporary register `%r10`. The next instruction increments the value in `%r10` by the literal or *immediate* value 1. Note that immediate values are always prefixed by a '\$'.
- The second `mov` instruction stores the value of the addition back onto the stack at the second position of the frame (after the saved `%rbp`).

```
mov    -16(%rbp), %rax
```

- According to the [calling convention](#), a function must place its return value in the `%rax` register, so `inc` has succeeded in returning $x + 3$ by moving the value of the $x + 3$ expression into `%rax`.

```

    .globl main
main:
    enter    $(8 * 6), $0
    ...

```

- The `.globl main` directive makes the symbol `main` accessible to modules other than this one. This is important, because the C run-time library, which we link against, expects to find a `main` procedure to call at program startup.
- The `enter` instruction allocates space for 6 quadwords on the stack: one for a local variable and 5 for temporaries.

```

    call     get_int
    mov      %rax, -8(%rbp)

```

- We call the `get_int` function, which reads an integer from standard input and returns it. The function takes no arguments.
- The integer is returned in `%rax`, and we store the value of the method call expression onto the stack.

```

    mov      -8(%rbp), %rdi
    call     inc
    mov      %rax, -16(%rbp)
    mov      -16(%rbp), %r10
    mov      %r10, -24(%rbp)

```

- Now we are ready to call `inc`. We start by loading the temporary that stored the return value of `get_int` into `%rdi`. According to the calling convention defined in the Linux ABI ([see below](#)), `%rdi` is used to pass the first argument. Then we call `inc`.
- Once `inc` returns, we store the return value, stored in `%rax`, onto the stack at location `-16(%rbp)`.
- Next, we perform the assignment to `y` of the return value of `inc` by loading the temporary into `%r10` and storing `%r10` into the stack location designated for `y`, `-24(%rbp)`.

```

    mov      -24(%rbp), %r10
    mov      $1, %r11
    cmp      %r10, %r11
    mov      $0, %r11
    mov      $1, %r10
    cmov     %r10, %r11
    mov      %r11, -32(%rbp)

```

- This sequence demonstrates how a comparison operation might be implemented using only two

registers and temporary storage. We begin by loading the values to compare, i.e., `y` and the literal 15, into registers. This is necessary because the comparison instructions only work on register operands.

- Then, we perform the actual comparison using the `cmp` instruction. The result of the comparison is to change the internal flags register.
- Our aim is to store a boolean value—1 or 0—in a temporary variable as the result of this operation. To set this up, we place the two possible values, 1 and 0, in registers `%r10` and `%r11`.
- Then we use the `cmove` instruction (read c-mov-e, or conditional move if equal) to decide whether our output value should be 0 or 1, based on the flags set by our previous comparison. The instruction puts the result in `%r11`.
- Finally, we store the boolean value from `%r11` to a temporary variable at `-32(%rbp)`.

```

    mov    -32(%rbp), %r10
    mov    $1, %r11
    cmp    %r10, %r11
    je     .one
    ...
    jmp    .one_done
.one:
    ...
.one_done:

```

- This is the standard linearized structure of a conditional statement. We compare a boolean variable to 1, and perform a `je` (jump if equal) instruction which jumps to its target block if the comparison succeeded. If the comparison failed, `je` acts as a no-op.
- We mark the end of the target block with a label, and jump to it at the end of the fall-through block. Conventionally, such *local labels*, which do not define functions, are named starting with a period.

```

    mov    $1, %rax
    call   printInt
    mov    %rax, -40(%rbp)

```

- The block of instructions performs the false (else) block of the if statement.
- After the call, the final move instruction stores the return value of `printInt` onto the stack. Note that the return value is never referenced.

```

    mov    $0, %rax
    leave
    ret

```

- At the end of the procedure, we set %rax to 0 to indicate that the program has terminated successfully (used even though the main method is declared to be of type void).

Reference

This handout only mentions a small subset of the rich possibilities provided by the x86-64 instruction set and architecture. For a more complete (but still readable) introduction, consult [The AMD64 Architecture Programmer's Manual, Volume 1: Application Programming](#).

Registers

In the assembly syntax accepted by gcc, register names are always prefixed with %. All of these registers are 64 bits wide.

The register file is as follows:

Register	Purpose	Saved across calls
%rax	temp register; return value	No
%rbx	callee-saved	Yes
%rcx	used to pass 4th argument to functions	No
%rdx	used to pass 3rd argument to functions	No
%rsp	stack pointer	Yes
%rbp	callee-saved; base pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10-r11	temporary	No
%r12-r15	callee-saved registers	Yes

For the code generation phase of the project you will not be performing register allocation. You should use %r10 and %r11 for temporary values that you load from the stack.

Instruction Set

Each mnemonic opcode presented here represents a family of instructions. Within each family, there are variants which take different argument types (registers, immediate values, or memory addresses) and/or argument sizes (byte, word, double-word, or quad-word). The former can be distinguished from the prefixes of the arguments, and the latter by an optional one-letter suffix on the mnemonic.

For example, a mov instruction which sets the value of the 64-bit %rax register to the immediate value 3 can be written as

```
movq    $3, %rax
```

Immediate operands are always prefixed by \$. Un-prefixed operands are treated as memory addresses, and should be avoided since they are confusing.

For instructions which modify one of their operands, the operand which is modified appears second. This differs from the convention used by Microsoft's and Borland's assemblers, which are commonly used on DOS and Windows.

Opcode	Description
Copying values	
mov src, dest	Copies a value from a register, immediate value or memory address to a register or memory address.
cmove %src, %dest	Copies from register %src to register %dest if the last comparison operation had the corresponding result (cmove: equality, cmovne: inequality, cmovg: greater, cmovl: less, cmovge: greater or equal, cmovle: less or equal).
cmovne %src, %dest	
cmovg %src, %dest	
cmovl %src, %dest	
cmovge %src, %dest	
cmovle %src, %dest	
Stack management	
enter \$x, \$0	Sets up a procedure's stack frame by first pushing the current value of %rbp on to the stack, storing the current value of %rsp in %rbp, and finally decreasing %rsp to make room for x byte-sized local variables.
leave	Removes local variables from the stack frame by restoring the old values of %rsp and %rbp.
push src	Decreases %rsp and places src at the new memory location pointed to by %rsp. Here, src can be a register, immediate value or memory address.
pop dest	Copies the value stored at the location pointed to by %rsp to dest and increases %rsp. Here, dest can be a register or memory location.
Control flow	
call target	Jump unconditionally to target and push return value (current PC + 1) onto stack.
ret	Pop the return address off the stack and jump unconditionally to this address.
jmp target	Jump unconditionally to target, which is specified as a memory location (for example, a label).
je target	Jump to target if the last comparison had the corresponding result (je: equality; jne: inequality).
jne target	
Arithmetic and logic	
add src, dest	Add src to dest.
sub src, dest	Subtract src from dest.
imul src, dest	Multiply dest by src.
idiv divisor	Divide rdx:rax by divisor. Store quotient in rax and store remainder in rdx.
shr reg	Shift reg to the left or right by value in cl (low 8 bits of rcx).

shl reg	
ror src, dest	Rotate dest to the left or right by src bits.
cmp src, dest	Set flags corresponding to whether dest is less than, equal to, or greater than src

Stack Organization

Global and local variables are stored on the stack, a region of memory that is typically addressed by offsets from the registers %rbp and %rsp. Each procedure call results in the creation of a *stack frame* where the procedure can store local variables and temporary intermediate values for that invocation. The stack is organized as follows:

Position	Contents	Frame
8n+16(%rbp)	argument n	Previous
...	...	
16(%rbp)	argument 7	
8(%rbp)	return address	Current
0(%rbp)	previous %rbp value	
-8(%rbp)	locals and temps	
...		
0(%rsp)		

Calling Convention

We will use the standard Linux function calling convention. The calling convention is defined in detail in [System V Application Binary Interface—AMD64 Architecture Processor Supplement](#). We will summarize the calling convention as it applies to C-TDS.

The caller uses registers to pass the first 6 arguments to the callee. Given the arguments in left-to-right order, the order of registers used is: %rdi, %rsi, %rdx, %rcx, %r8, and %r9. Any remaining arguments are passed on the stack in reverse order so that they can be popped off the stack in order.

The callee is responsible for perserving the value of registers %rbp %rbx, and %r12-r15, as these registers are owned by the caller. The remaining registers are owned by the callee.

The callee places its return value in %rax and is responsible for cleaning up its local variables as well as for removing the return address from the stack.

The call, enter, leave and ret instructions make it easy to follow this calling convention.

Since we follow the standard linux ABI, we can call C functions and library functions using our callout structure. For the purposes of the project we are only going to call printInt and get_int. When calling printInt, we must set the value of register %rax to 0 before issuing the call instruction. This is because printInt uses a variable number of arguments and %rax specifies how many SSE registers are used for the arguments. For our purposes the value will always be 0. Since callouts can only return an single integer value, we have provided a function get_int(), which will read a single integer input from the terminal and return its integer value. This function is included in the 6035 static library. We

cannot use scanf because it returns the number of items read.