X86-64 Architecture Guide

For the code-generation project, we shall expose you to a simplified version of the x86-64 platform.

Example

Consider the following Decaf program:

```
class Program {

    int foo(int x) {
        return x + 3;
    }

    void main() {
        int y;

        y = foo(callout("get_int_035"));
        if (y == 15) {
            callout("printf_035", "Indeed! \'tis 15!\n");
        } else {
            callout("printf_035", "What! %d\n", y);
        }
    }

}
```

One compiled version of this program might look like this:

```
foo:
    enter   $0, $0
    mov     16(%rbp), %rax
    add     $3, %rax
    leave
    ret

    .globl main
main:
    enter   $(8 * 3), $0

    call    get_int_035
    push    %rax

    call    foo
```

```
    add     $8, %rsp
    mov     %rax, -8(%rbp)

    mov     -8(%rbp), %r10
    mov     $15, %r11
    cmp     %r10, %r11
    mov     $0, %r11
    mov     $1, %r10
    cmove   %r10, %r11
    mov     %r11, -16(%rbp)

    mov     -16(%rbp), %r10
    mov     $1, %r11
    cmp     %r10, %r11
    je      .fifteen

    push    -8(%rbp)
    push    $.what
    call    printf_035
    add     $(2 * 8), %rsp
    jmp     .fifteen_done

.fifteen:
    push    $.indeed
    call    printf_035
    add     $(1 * 8), %rsp
.fifteen_done:

    mov     $0, %rax
    leave
    ret

.indeed:
    .string "Indeed, \'tis 15!\n"

.what:
    .string "What! %d\n"
```

We shall dissect this assembly listing carefully and relate it to the Decaf 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.

```
foo:
    enter   $(8 * 0), $0
    ...
```

```
    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.

```
    mov     16(%rbp), %rax
    add     $3, %rax
```

- The purpose of foo is to add 3 to its argument, and return the result. The arguments to a function are stored in its caller's frame, at positive quadword-aligned offsets from `%rbp`. The *k*-th argument is stored at location (8 + 8 *k*)(`%rbp`), so the `mov` instruction moves the value of the first argument into the `%rax` register. The next instruction increments the value in `%rax` by the literal or *immediate* value 3. Note that immediate values are always prefixed by a '$'.

- According to the [calling convention](), a function must place its return value in the `%rax` register, so `foo` has succeeded in returning `x + 3`.

```
    .globl main
main:
    enter   $(8 * 3), $0
    ...
    mov     $0, %rax
    leave
    ret
```

- 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 three quadwords on the stack: one for a local variable and two for arguments passed to functions.

- At the end of the procedure, we set `%rax` to 0 to indicate that the program has terminated successfully.

```
call    get_int_035
push    %rax
```

- We call the `get_int_035` function, which reads an integer from standard input and returns it. The function takes no arguments.
- The integer is returned in `%rax`, and we push it on the stack to be used as an argument to `foo`. Notice that we have optimized somewhat here: another valid approach would have been to store the return value in a local variable, and then load it back from there to push it as an argument.

```
call    foo
add     $8, %rsp
mov     %rax, -8(%rbp)
```

- With the one argument we have already pushed on the stack, we call `foo`.
- Once `foo` returns, we need to clean up the stack by removing the arguments we pushed on to it earlier. Here, we increase `%rsp`; we could also have performed a `pop` instruction.
- Finally, we save the return value stored in `%rax` to a temporary local variable. Local variables are stored at negative offsets from `%rbp`.

```
mov     -8(%rbp), %r10
mov     $15, %r11
cmp     %r10, %r11
mov     $0, %r11
mov     $1, %r10
cmove   %r10, %r11
mov     %r11, -16(%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., the return value of `foo` 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 local 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 local variable at `-16(%rbp)`.

```
    mov     -16(%rbp), %r10
    mov     $1, %r11
    cmp     %r10, %r11
    je      .fifteen
    ...
    jmp     .fifteen_done
.fifteen:
    ...
.fifteen_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.

```
.indeed:
    .string "Indeed, \'tis 15!\n"

.what:
    .string "What! %d\n"
```

- These labels point to static strings defined in the program. They are used as arguments to functions.

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 `%`. For the first part of the project, we shall use only five of the x86-64's sixteen general-purpose registers. All of these registers are 64 bits wide.

Register	Purpose	Saved across calls
`%rax`	return value	No
`%rsp`	stack pointer	Yes
`%rbp`	base pointer	Yes
`%r10`	temporary	No
`%r11`		

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 `%esp` in `%ebp`, and finally decreasing `%esp` to make room for `x` quadword-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	
`jmp target`	Jump unconditionally to `target`, which is specified as a memory location (for example, a label).
`je target` `jne target`	Jump to `target` if the last comparison had the corresponding result (`je`: equality; `jne`: inequality).
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 src, dest`	Divide `dest` by `src`.
`shr src, dest` `shl src, dest`	Shift `dest` to the left or right by `src` bits.
`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 0	
`8(%rbp)`	return address	Current
`0(%rbp)`	previous `%rbp` value	
`-8(%rbp)`	locals and temps	
...		
`0(%rsp)`		

Calling Convention

The caller pushes arguments on to the stack in reverse order. Finally, it pushes the return address and transfers control to 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. It is *not* responsible for removing the arguments.

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

The standard calling convention used by C programs under Linux on x86-64 is a little different; see *System V Application Binary Interface—AMD64 Architecture Processor Supplement* for details. Specifically, it optimizes calls by passing the first few arguments in registers instead of on the stack. As a result, your program cannot directly call out to arbitrary C procedures yet. Instead, we have provided two functions, `printf_035` and `get_int_035`, which have been specifically adapted to this simplified convention.

Function	Description
`printf_035(fmt, arg1, ...)`	Print a formatted string to standard output, exactly like `printf(3)`.
`get_int_035()`	Read a single signed decimal integer from standard input and return its value.