X86-64 Architecture Guide

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 decaf.

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 `printf` and `get_int_035`. When calling `printf`, we must set the value of register %rax to 0 before issuing the call instruction. This is because `printf` 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_035()`, 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.