CMSC 22620 Spring 2009	Implementation of Computer Languages	Handout 3 April 14, 2009

x86-64 Instructions and ABI

1 Introduction

You will be generating assembly code for the x86-64 architecture, which is the 64-bit extension to Intel's venerable x86 architecture. Most instructions in this architecture have two operands: a source and a destination that specifies the second operand and the location of the result. Operands can be registers, immediate values, or memory locations, but not all combinations are supported. The code you generate will be in the so-called AT&T syntax, which places the source on the left and the destination on the right.[1] Most instruction names include a single-letter suffix that specifies the size of the operands. We will be using the 64-bit instructions, which use the letter "`q`" (for quadword) as a suffix. The other suffixes are `b` for 8 bits, `w` for 16 bits, and `l` for 32 bits.

The sample code includes the `Instruction` module that supports a subset of x86-64 instructions. The implementation of this module checks for the well-formedness of operands, and will raise an exception when it detects an error.

2 Registers

The x86-64 has sixteen 64-bit registers. In the AT&T assembler syntax registers are denoted with a leading "`%`" character. Some registers have special roles, for example the `%rdx` and `%rax` register pair is used in the `idivq` instruction. The table in Figure 1 lists the registers and describes their use. It also marks those registers that are *callee save*.

3 Calling conventions

Both Mac OS X and Linux follow the *System V ABI* for their x86-64 calling conventions.[2] There are three x86-64 instructions used to implement procedure calls and returns.

- The `call` instruction pushes the address of the next instruction (*i.e.*, the return address) onto the stack and then transfers control to the address specified by its operand.
- The `leave` instruction sets the stack pointer (`%rsp`) to the frame pointer (`%rbp`) and then sets the frame pointer to the saved frame pointer, which is popped from the stack.

[1] Note that this is the *opposite* of the Intel syntax, so be careful when reading descriptions of the instructions.

[2] The ABI specification is available at `http://www.x86-64.org/documentation/abi.pdf`.

Register	Callee Save	Description
%rax		result register; also used in idiv and imul instructions.
%rbx	yes	miscellaneous register
%rcx		fourth argument register
%rdx		third argument register; also used in idiv and imul instructions.
%rsp		stack pointer
%rbp	yes	frame pointer
%rsi		second argument register
%rdi		first argument register
%r8		fifth argument register
%r9		sixth argument register
%r10		miscellaneous register
%r11		miscellaneous register
%r12-%r15	yes	miscellaneous registers

Figure 1: The x86-64 general-purpose registers

- The `ret` instruction pops the return address off the stack and jumps to it.

The registers `%rbp`, `%rbx`, and `%r12-%r15` are callee save.

3.1 Arguments

The first six arguments to a function are passed in registers. Any additional arguments are passed on the stack in the memory-argument area (see Figure 2). The `%rax` register is used to return the first result and the `%rdx` register is used to return a second result.

3.2 Stack frames

The stack grows from higher addresses to lower addresses. The ABI uses two registers to access the stack: the frame pointer (`%rbp`), which points to the base of the frame, and the stack pointer (`%rsp`), which points to the top of the frame. Figure 2 shows the layout of the frame. Normally, the frame pointer is used to address data in the frame, such as the incoming parameters and local variables.

The ABI requires that stack frames be aligned on 16-byte boundaries. Specifically, the end of the argument area (`%rbp+16`) must be a multiple of 16. This requirement means that the frame size should be padded out to a multiple of 16 bytes.

3.3 Procedure-calling protocol

The protocol for procedure calls can be broken into four pieces; two each for the caller and callee. We describe these in the order that the happen.

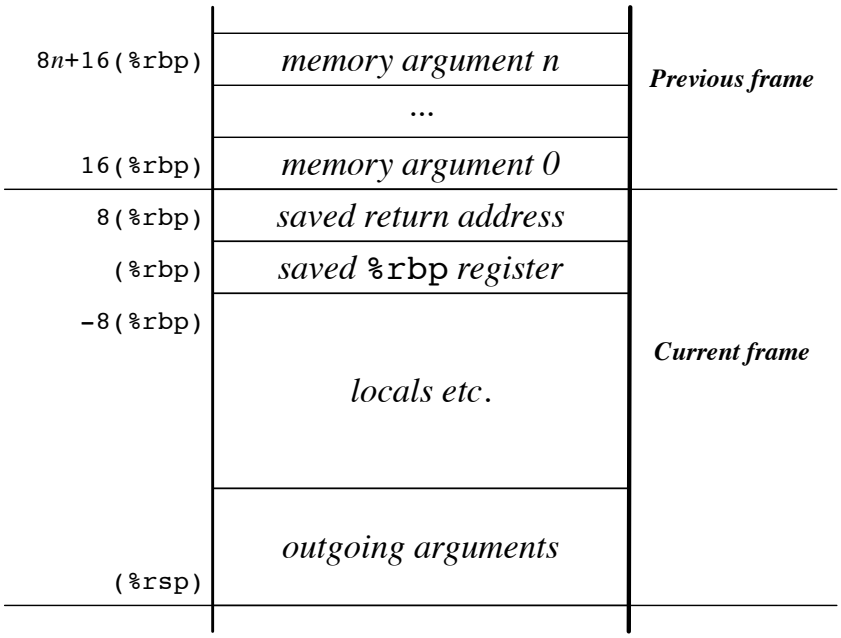

Figure 2: Stack-frame layout

3.3.1 Procedure call

The caller-side of a procedure call involves saving any caller-save registers that are live across the call, loading the arguments into the appropriate registers and stack locations, and then executing the `call` instruction. In the Figure 2, the stack frame includes an area for outgoing arguments. It is also possible to allocate this memory for each call, in which case the caller is responsible for deallocating it upon return.

3.3.2 Procedure entry

Upon entry, a callee needs to initialize its linkage and stack frame. This initialization is accomplished by the following sequence:

```
pushq   %rbp
movq    %rsp,%rbp
subq    $N,%rsp
```

where N is the size of the callee's stack frame. Once the linkage is established, the callee may choose to save any callee-save registers that it uses.

3.3.3 Procedure exit

Once the procedure has finished execution, the callee performs the procedure-exit protocol. This protocol involves putting the result in `%rax`, deallocating the stack frame, and returning control to the caller. The following code sequence handles the latter two steps:

```
leave
ret
```

3.3.4 Procedure return

If the caller is allocating stack space for arguments on a per-call basis, then it is responsible for deallocating the space upon return.

4 Instructions

For the project, you use a small subset of the x86-64 instructions (mostly the 64-bit integer operations plus control-flow operations).

4.1 Operands

There are three basic kinds of operands: registers, immediates (which are numbers preceded by the "`$`" character in assembly code), and memory addresses. We use reg to denote registers, $imm32$ and $imm64$ to denote immediates, and mem to denote memory addresses in the discussion below. Since we are computing with 64-bit values, 32-bit immediates ($imm32$) are sign-extended to 64 bits.

The x86-64 supports a number of different modes for addressing memory. The following table describes the syntax of these modes and the effective addresses that they define:

Syntax	Address	Description
(reg)	reg	Base addressing
$d(reg)$	$reg + d$	Base plus displacement addressing
$d(reg, s)$	$(s \times reg) + d$	Scaled index plus displacement; $s \in \{2, 4, 8\}$
$d(reg_1, reg_2, s)$	$reg_1 + (s \times reg_2) + d$	Base plus scaled index and displacement; $s \in \{2, 4, 8\}$

In this syntax, d and s are numbers (without the leading "`$`").

4.2 Opcodes

The following table lists the x86-64 instructions that you will need for the project. For each instruction, we have included the various formats that are supported. and a description of the operation.

`addq`	reg_1, reg_2	$reg_2 \leftarrow reg_2 + reg_1$
`addq`	reg, mem	$\mathrm{M}[mem] \leftarrow \mathrm{M}[mem] + reg$
`addq`	$imm32, reg$	$reg \leftarrow reg + imm32$
`addq`	$imm32, mem$	$\mathrm{M}[mem] \leftarrow \mathrm{M}[mem] + imm32$
`addq`	mem, reg	$reg \leftarrow reg + \mathrm{M}[mem]$

`andq`	reg_1, reg_2	$reg_2 \leftarrow reg_2$ **AND** reg_1
`andq`	reg, mem	$\mathrm{M}[mem] \leftarrow \mathrm{M}[mem]$ **AND** reg
`andq`	$imm32, reg$	$reg \leftarrow reg$ **AND** $imm32$
`andq`	$imm32, mem$	$\mathrm{M}[mem] \leftarrow \mathrm{M}[mem]$ **AND** $imm32$
`andq`	mem, reg	$reg \leftarrow reg$ **AND** $\mathrm{M}[mem]$
`call`	lab	procedure call
`call`	$*reg$	procedure call (register indirect)
`call`	$*(reg)$	procedure call (memory indirect)
`cmpq`	reg_1, reg_2	$ccode \leftarrow \mathrm{TEST}(reg_2 - reg_1)$
`cmpq`	reg, mem	$ccode \leftarrow \mathrm{TEST}(\mathrm{M}[mem] - reg)$
`cmpq`	mem, reg	$ccode \leftarrow \mathrm{TEST}(reg - \mathrm{M}[mem])$
`cmpq`	$imm32, reg$	$ccode \leftarrow \mathrm{TEST}(reg - imm32)$
`cmpq`	$imm32, mem$	$ccode \leftarrow \mathrm{TEST}(\mathrm{M}[mem] - imm32)$
`idivq`	reg	$\%\mathtt{rax} \leftarrow \%\mathtt{rdx} : \%\mathtt{rax}$ **DIV** reg $\%\mathtt{rdx} \leftarrow \%\mathtt{rdx} : \%\mathtt{rax}$ **MOD** reg
`idivq`	mem	$\%\mathtt{rax} \leftarrow \%\mathtt{rdx} : \%\mathtt{rax}$ **DIV** $\mathrm{M}[mem]$ $\%\mathtt{rdx} \leftarrow \%\mathtt{rdx} : \%\mathtt{rax}$ **MOD** $\mathrm{M}[mem]$
`imulq`	reg_1, reg_2	$reg_2 \leftarrow reg_2 \times reg_1$
`imulq`	mem, reg	$reg \leftarrow reg \times \mathrm{M}[mem]$
`imulq`	$imm32, reg$	$reg \leftarrow reg \times imm32$
`ja`	lab	jump to lab if above ($\mathrm{CF} = 0 \wedge \mathrm{ZF} = 0$)
`jae`	lab	jump to lab if above or equal ($\mathrm{CF} = 0$)
`jb`	lab	jump to lab if below ($\mathrm{CF} = 1$)
`jbe`	lab	jump to lab if below or equal ($\mathrm{CF} = 1 \wedge \mathrm{ZF} = 1$)
`je`	lab	jump to lab if equal ($\mathrm{ZF} = 1$)
`jg`	lab	jump to lab if greater ($\mathrm{ZF} = 0 \wedge \mathrm{SF} = \mathrm{OF}$)
`jge`	lab	jump to lab if greater or equal ($\mathrm{SF} = \mathrm{OF}$)
`jl`	lab	jump to lab if less ($\mathrm{SF} \neq \mathrm{OF}$)
`jle`	lab	jump to lab if less or equal ($\mathrm{ZF} = 1 \wedge \mathrm{SF} \neq \mathrm{OF}$)
`jne`	lab	jump to lab if not equal ($\mathrm{ZF} = 0$)
`jns`	lab	jump to lab if not sign flag ($\mathrm{SF} = 0$)
`js`	lab	jump to lab if if sign flag ($\mathrm{SF} = 1$)
`jmp`	lab	jump to lab
`jmp`	$*reg$	jump to the address in reg
`jmp`	$*(reg)$	jump to the address in $\mathrm{M}[reg]$
`leaq`	mem, reg	$reg \leftarrow mem$ (load effective address)
`leave`		$\%\mathtt{rsp} \leftarrow \%\mathtt{rbp}$; $\%\mathtt{rbp} \leftarrow \mathrm{M}[\%\mathtt{rsp}]$; $\%\mathtt{rsp} \leftarrow \%\mathtt{rsp} + 8$.
`movabsq`	lab, reg	$reg \leftarrow lab$
`movq`	reg_1, reg_2	$reg_2 \leftarrow reg_1$
`movq`	reg, mem	$\mathrm{M}[mem] \leftarrow reg$
`movq`	mem, reg	$reg \leftarrow \mathrm{M}[mem]$
`movq`	$imm32, reg$	$reg \leftarrow imm32$
`movq`	$imm64, reg$	$reg \leftarrow imm64$
`orq`	reg_1, reg_2	$reg_2 \leftarrow reg_2$ **OR** reg_1
`orq`	reg, mem	$\mathrm{M}[mem] \leftarrow \mathrm{M}[mem]$ **OR** reg

`orq`	*imm32*, *reg*	*reg* $\leftarrow$ *reg* **OR** *imm32*
`orq`	*imm32*, *mem*	M[*mem*] $\leftarrow$ M[*mem*] **OR** *imm32*
`orq`	*mem*, *reg*	*reg* $\leftarrow$ *reg* **OR** M[*mem*]
`pushq`	*reg*	%`rsp` $\leftarrow$ %`rsp` $-$ 8; M[%`rsp`] $\leftarrow$ *reg*
`pushq`	*mem*	%`rsp` $\leftarrow$ %`rsp` $-$ 8; M[%`rsp`] $\leftarrow$ M[*mem*]
`pushq`	*imm32*	%`rsp` $\leftarrow$ %`rsp` $-$ 8; M[%`rsp`] $\leftarrow$ *imm32*
`ret`		return from procedure call
`salq`	*imm32*, *reg*	*reg* $\leftarrow$ *reg* $<<$ *imm32*
`salq`	*imm32*, *mem*	*mem* $\leftarrow$ *mem* $<<$ *imm32*
`sarq`	*imm32*, *reg*	*reg* $\leftarrow$ *reg* $>>$ *imm32* (arithmetic shift)
`sarq`	*imm32*, *mem*	*mem* $\leftarrow$ *mem* $>>$ *imm32*
`shrq`	*imm32*, *reg*	*reg* $\leftarrow$ *reg* $>>$ *imm32* (logical shift)
`shrq`	*imm32*, *mem*	*mem* $\leftarrow$ *mem* $>>$ *imm32*
`subq`	reg_1, reg_2	$reg_2 \leftarrow reg_2 - reg_1$
`subq`	*reg*, *mem*	M[*mem*] $\leftarrow$ M[*mem*] $-$ *reg*
`subq`	*imm32*, *reg*	*reg* $\leftarrow$ *reg* $-$ *imm32*
`subq`	*imm32*, *mem*	M[*mem*] $\leftarrow$ M[*mem*] $-$ *imm32*
`subq`	*mem*, *reg*	*reg* $\leftarrow$ *reg* $-$ M[*mem*]
`xorq`	reg_1, reg_2	$reg_2 \leftarrow reg_2$ **XOR** reg_1
`xorq`	*reg*, *mem*	M[*mem*] $\leftarrow$ M[*mem*] **XOR** *reg*
`xorq`	*imm32*, *reg*	*reg* $\leftarrow$ *reg* **XOR** *imm32*
`xorq`	*imm32*, *mem*	M[*mem*] $\leftarrow$ M[*mem*] **XOR** *imm32*
`xorq`	*mem*, *reg*	*reg* $\leftarrow$ *reg* **XOR** M[*mem*]

Revision history

April 26, 2009 Fixed description of `imulq` instruction to match code.

April 25, 2009 Fixed stack-frame picture.

April 15, 2009 The `pushq` instruction also supports *imm32* operands. Also added note about sign extension of 32-bit immediates.

April 14, 2009 Fixed table in Section 4.2: changed `addq` to `andq` and changed `popq` to `pushq`.