X86_64 programming

Tutorial #1

CPSC 261

X86_64

- An extension of the IA32 (often called x86 – originated in the Intel 8086 processor) instruction set to 64 bits
- AMD defined it first
- Intel implemented it later

X86_64 data types

C declaration	Intel data type	GAS suffix	x86-64 Size (Bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`unsigned`	Double word	`l`	4
`long int`	Quad word	`q`	8
`unsigned long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`d`	8
`long double`	Extended precision	`t`	16

Basic properties

- Pointers and long integers are 64 bits long. Integer arithmetic operations support 8, 16, 32, and 64-bit data types.
- There are 16 general-purpose registers.
- Much of the program state is held in registers rather than on the stack. Integer and pointer procedure arguments (up to 6) are passed via registers. Some procedures do not need to access the stack at all.
- Conditional move instructions allow conditional operations to be implemented without traditional branching code.
- Floating-point operations are implemented using a register-oriented instruction set.

Registers

- 8 registers with strange names:
  – rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp
- 8 more with "normal" names:
  – r8, r9, r10, r11, r12, r13, r14, r15
- All can be accessed at various sizes
  – 8, 16, 32, 64 bits
- rsp is the stack pointer
- rbp is the frame pointer (for some compilers)

63	31	15	8	7 0	
%rax	%eax	%ax	%ah	%al	Return value
%rbx	%ebx	%ax	%bh	%bl	Callee saved
%rcx	%ecx	%cx	%ch	%cl	4th argument
%rdx	%edx	%dx	%dh	%dl	3rd argument
%rsi	%esi	%si		%sil	2nd argument
%rdi	%edi	%di		%dil	1st argument
%rbp	%ebp	%bp		%bpl	Callee saved
%rsp	%esp	%sp		%spl	Stack pointer
%r8	%r8d	%r8w		%r8b	5th argument
%r9	%r9d	%r9w		%r9b	6th argument
%r10	%r10d	%r10w		%r10b	Callee saved
%r11	%r11d	%r11w		%r11b	Used for linking
%r12	%r12d	%r12w		%r12b	Unused for C
%r13	%r13d	%r13w		%r13b	Callee saved
%r14	%r14d	%r14w		%r14b	Callee saved
%r15	%r15d	%r15w		%r15b	Callee saved

Register strangenesses

- Moving a 32 bit value to a 64 bit register sets the higher order 32 bits to zero (doesn't sign extend!!)
- Moving a 8 or 16 bit value to a 64 bit register doesn't affect the higher order bits of the register

Integer instructions

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow$ `&`S	Load effective address
`incq`	D	$D \leftarrow D$ `+ 1`	Increment
`decq`	D	$D \leftarrow D$ `- 1`	Decrement
`negq`	D	$D \leftarrow$ `-`D	Negate
`notq`	D	$D \leftarrow$ `~`D	Complement
`addq`	S, D	$D \leftarrow D$ `+` S	Add
`subq`	S, D	$D \leftarrow D$ `-` S	Subtract
`imulq`	S, D	$D \leftarrow D$ `*` S	Multiply
`xorq`	S, D	$D \leftarrow D$ `^` S	Exclusive-or
`orq`	S, D	$D \leftarrow D$ `|` S	Or
`andq`	S, D	$D \leftarrow D$ `&` S	And
`salq`	k, D	$D \leftarrow D$ `<<` k	Left shift
`shlq`	k, D	$D \leftarrow D$ `<<` k	Left shift (same as `salq`)
`sarq`	k, D	$D \leftarrow D$ `>>` k	Arithmetic right shift
`shrq`	k, D	$D \leftarrow D$ `>>` k	Logical right shift

Weird integer instructions

Instruction		Effect	Description
`imulq`	S	R[`%rdx`]:R[`%rax`] $\leftarrow S \times$ R[`%rax`]	Signed full multiply
`mulq`	S	R[`%rdx`]:R[`%rax`] $\leftarrow S \times$ R[`%rax`]	Unsigned full multiply
`cltq`		R[`%rax`] $\leftarrow$ SignExtend(R[`%eax`])	Convert `%eax` to quad word
`cqto`		R[`%rdx`]:R[`%rax`] $\leftarrow$ SignExtend(R[`%rax`])	Convert to oct word
`idivq`	S	R[`%rdx`] $\leftarrow$ R[`%rdx`]:R[`%rax`] mod S; R[`%rax`] $\leftarrow$ R[`%rdx`]:R[`%rax`] $\div S$	Signed divide
`divq`	S	R[`%rdx`] $\leftarrow$ R[`%rdx`]:R[`%rax`] mod S; R[`%rax`] $\leftarrow$ R[`%rdx`]:R[`%rax`] $\div S$	Unsigned divide

Condition codes

- Single bit registers
  - CF Carry flag (for unsigned)
  - ZF Zero flag
  - SF Sign flag (for signed)
  - OF Overflow flag (for signed)

Compare instructions

Instruction	Based on	Description
`cmpq` S_2, S_1	S_1 - S_2	Compare quad words
`testq` S_2, S_1	S_1 & S_2	Test quad word

Branches

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Conditional moves

Instruction	Synonym	Move condition	Description
cmove *S*, *D*	cmovz	ZF	Equal / zero
cmovne *S*, *D*	cmovnz	˜ZF	Not equal / not zero
cmovs *S*, *D*		SF	Negative
cmovns *S*, *D*		˜SF	Nonnegative
cmovg *S*, *D*	cmovnle	˜(SF ˆ OF) & ˜ZF	Greater (signed >)
cmovge *S*, *D*	cmovnl	˜(SF ˆ OF)	Greater or equal (signed >=)
cmovl *S*, *D*	cmovnge	SF ˆ OF	Less (signed <)
cmovle *S*, *D*	cmovng	(SF ˆ OF) | ZF	Less or equal (signed <=)
cmova *S*, *D*	cmovnbe	˜CF & ˜ZF	Above (unsigned >)
cmovae *S*, *D*	cmovnb	˜CF	Above or equal (Unsigned >=)
cmovb *S*, *D*	cmovnae	CF	Below (unsigned <)
cmovbe *S*, *D*	cmovna	CF | ZF	below or equal (unsigned <=)

Procedure calls

- Arguments (up to the first six) are passed to procedures via registers.
- The call instruction stores a 64-bit return pointer on the stack.
- Many functions do not require a stack frame. Only functions that cannot keep all local variables in registers need to allocate space on the stack.
- Functions can access storage on the stack up to 128 bytes beyond (i.e., at a lower address than) the current value of the stack pointer.
- There is no frame pointer. Instead, references to stack locations are made relative to the stack pointer.
- Typical functions allocate their total stack storage needs at the beginning of the call and keep the stack pointer at a fixed position.
- Some registers are designated as callee-save registers. These must be saved and restored by any procedure that modifies them.

And then ...

- We should look at some C code compiled to X64_64 assembler to explain how functions are called and parameters are passed.

test.c

```
#include <stdio.h>

long f(long a, long b, long c, long d, long e, long f) {
   return a*b*c*d*e*f;
}

long main(long argc, char **argv) {
   long x = f(1, 2, 3, 4, 5, 6);
   return x;
}
```