1 x86-64 Assembly Language

1.1 Registers & operand indicators

Table 1: x86-64 registers & operand indicators

63	31	15	7	function
%rax	%eax	%ax	%al	return value
%rbx	%ebx	%bx	%bl	preserved by callee
%rcx	%ecx	%cx	%cl	4th argument
%rdx	%edx	%dx	%dl	3rd argument
%rsi	%esi	%si	%sil	2nd argument
%rdi	%edi	%di	%dil	1st argument
%rbp	%ebp	%bp	%bpl	preserved by callee
%rsp	%esp	%sp	%spl	stack pointer
%r8	%r8d	%r8w	%r8b	5th parameter
%r9	%r9d	%r9w	%r9b	6th parameter
%r10	%r10d	%r10w	%r10b	preserved by caller
%r11	%r11d	%r11w	%r11b	preserved by caller
%r12	%r12d	%r12w	%r12b	preserved by callee
%r13	%r13d	%r13w	%r13b	preserved by callee
%r14	%r14d	%r14w	%r14b	preserved by callee
%r15	%r15d	%r15w	%r15b	preserved by callee
$\$Imm$	Immediate number	Imm		
r_a	Value of register r_a	$R[r_a]$		
$Imm(r_b, r_s, s)$	Value at memory	$M[Imm{+}R[r_b]{+}R[r_s]{*}s]$, $s = 1, 2, 4, 8$		

1.2 Data movement

- Direct move: `MOV S D, D`←`S`. `S` is immediate number, register or memory position. `D` is register or memory position. `S,D` cannot both be memory positions.

- Move with zero expansion: `MOVZ S R, R`←`Zero expansion(S)`. `S` can be register or memory position. `D` must be register. There is no `movzlq` because `movl` can set upper bits to 0, which is equivalent to zero expansion from 32 bits to 64 bits.

- Move with sign expansion: `MOVS S R, R`←`Sign expansion(S)`. `S` can be register or memory position. `D` must be register.

- Push/pop stack: push 4 words (64 bits) on or pop 4 words from the stack. Special cases: `pushq %rsp` pushes the original value of `%rsp` on the stack; `popq %rsp` puts the value read from memory in `%rsp`.

Table 2: Data movement instructions

`movb`	Move byte (1B = 8bit, char)
`movw`	Move word (2B = 16bit, short)
`movl`	Move long word (4B = 32bit, int). Also set upper 32 bits of the register to 0
`movq`	Move quad word (8B = 64bit, long, pointer). When `S` is immediate number, only expansion of 32-bit 2's component can be used. For 64-bit immediate value, use `movabsq`
`movabsq`	move 64-bit immediate number to register
`movzbw`	byte→word (1B→2B)
`movzbl`	byte→long word (1B→4B)
`movzbq`	byte→quad word (1B→8B)
`movzwl`	word→long word (2B→4B)
`movzwq`	word→quad word (2B→8B)
`movsbw`	byte→word (1B→2B)
`movsbl`	byte→long word (1B→4B)
`movsbq`	byte→quad word (1B→8B)
`movswl`	word→long word (2B→4B)
`movswq`	word→quad word (2B→8B)
`movslq`	long word→quad word (4B→8B)
`cltq`	`%rax`←`Sign expansion(%eax)`, i.e. `movslq %eax %rax`
`pushq S`	`R[%rsp]`←`R[%rsp]-8; M[R[%rsp]]`←`S`
`popq D`	`D`←`M[R[%rsp]]; R[%rsp]`←`R[%rsp]+8`

1.3 Arithmetic & logical operations

- Load address (only 1 version, `q`)
- Unary operations, binary operations, bitwise shifts (4 versions, `bwlq`)
- 128-bit integer manipulation

Table 3: Arithmetic & logical operation instructions

`leaq S D`	`D ← &S`		
`inc D`	`D ← D + 1`	`dec D`	`D ← D - 1`
`neg D`	`D ← -D`	`not D`	`D ← ˜D`
`add S D`	`D ← D + S`	`sub S D`	`D ← D - S`
`imul S D`	`D ← D * S`	`or S D`	`D ← D | S`
`and S D`	`D ← D & S`		
`sal k D`	`D ← D ≪ k`	`shl k D`	`D ← D ≪ k`

to be continued

continue

`sar k D`	`D ← D` $\gg_A$ `k`	`shr k D`	`D ← D` $\gg_L$ `k`
`imulq S`	`R[%rdx]:R[%rax] ← S × R[%rax]`		`signed multiplication`
`mulq S`	`R[%rdx]:R[%rax] ← S × R[%rax]`		`unsigned multiplication`
`cqto`	`R[%rdx]:R[%rax] ← Signed expansion (R[%rax])`		`4 words to 8 words`

`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S`
	`R[%rax] ← R[%rdx]:R[%rax] ÷ S`
			`signed division`

`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S`
	`R[%rax] ← R[%rdx]:R[%rax] ÷ S`
			`unsigned division`

1.4 Control flow

- All arithemtic & logical operations except `leaq` can make changes to condition codes `CF, ZF, SF, OF`.
- `cmp` and `test` instructions set the condition codes without changing values of registers. Both have 4 versions(`bwlq`).
- `set` instructions can set a **byte** to 0 or 1 according to different combinations condition codes.
- `jump` instructions can make the execution jump to a specified position according to different combinations of condition codes
- `cmov` (conditional move) instructions can move the value at the source (memory position or register) to the destination register. They can be applied to 16, 32 or 64 bits (i.e. single byte conditional move is not supported!).
- For `set, jump` and `cmov` instructions, g/l (greater/less) are for signed integers, while a/b (above/below) are for unsigned integers.

Table 4: Condition codes & control flow instructions

`CF`	`carry`	Unsigned overflow (carry at highest bit).		
`ZF`	`zero`	Most recent result is 0.		
`SF`	`sign`	Most recent result is negative.		
`OF`	`overflow`	Complement overflow (+ or -).		
`cmp` S_1 S_2		Change condition codes according to $S_2 - S_1$.		
`test` S_1 S_2		Change condition codes according to S_1&S_2.		
	`jmp`		1	Unconditional jump
`sete setz`	`je jz`	`cmove cmovz`	==	`ZF`
`setne setnz`	`jne jnz`	`cmovne cmovnz`	!=	`~ZF`
`sets`	`js`	`cmovs`	negative	`SF`
`setns`	`jns`	`cmovns`	not negative	`~SF`
`setg setnle`	`jg jnle`	`cmovg cmovnle`	>	`~(SF^OF) & ~ZF`

to be continued

continue

setge setnl	jge jnl	cmovge cmovnl	>=	~(SF^OF)
setl setnge	jl jnge	cmovl cmovnge	<	SF^OF
setle setng	jle jng	cmovle cmovng	<=	(SF^OF) | ZF
seta setnbe	ja jnbe	cmova cmovnbe	>	~CF & ~ZF
setae setnb	jae jnb	cmovae cmovnb	>=	~CF
setb setnae	jb jnae	cmovb cmovnae	<	CF
setbe setna	jbe jna	cmovbe cmovna	<=	CF | ZF

Using `jump` and `cmov` instructions, we can translate C structs into structures easier to implement with assembly language.

Table 5: Translation of C constructs

C construct	Assembly code logic	Implementation details

`if (test-expr)`
`  then-statement`
`else`
`  else-statement`
	`t = test-expr;`
	`if (!t)`
	`  goto false;`
	`  then-statement`
	`  goto done;`
	`false:`
	`  else-statement`
	`done:`
		Use `jump` instructions.

`if (test-expr)`
`  then-statement`
`else`
`  else-statement`
	`t = test-expr;`
	`v = then-statement;`
	`ve = else-statement;`
	`if(!t) v = ve;`
		Use `cmov` instructions. Typically only when both statements are easy to calculate and have no side effect.

`do`
`  body-statement`
`  while (test-expr);`
	`loop:`
	`  body-statement;`
	`  t = test-expr;`
	`  if(t) goto loop;`
		Use `jump` instructions.

`while (test-expr)`
`  body-statement;`
	`  goto test;`
	`loop:`
	`  body-statement;`
	`test:`
	`  t = test-expr;`
	`  if(t) goto loop;`
		Use `jump` instructions.

to be continued

continue

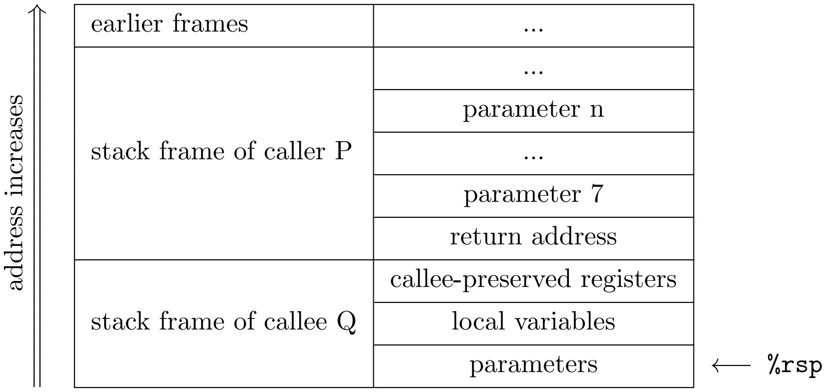
while (test-expr)
  body-statement;
	t = test-expr;
	  if(!t) goto done;
	loop:
	  body-statement;
	  t = test-expr;
	  if(t) goto loop;
	done:
		Use jump instructions.

for(init-expr;
  test-expr;
  update-expr)
  body-statement;
	init-expr;
	while(test-expr) {
	  body-statement;
	update:
	  update-expr;
	}
		Use jump instructions. update is useful only when body-statement contains continue.

switch(n) {
case 100:
  statement-0; break;
case 101:
  statement-1;
case 103:
case 104:
  statement-4; break;
default:
  statement-default;
}
	static void *jt[5] = {
	  &&loc_0,&&loc_1,
	  &&loc_def,&&loc_34,
	  &&loc_34
	};
	unsigned long i = n - 100;
	if(i > 4) goto loc_def;
	goto *jt[i];
	loc_0:
	  statement-0; goto done;
	loc_1:
	  statement-1;
	loc_34:
	  statement-4; goto done;
	loc_def:
	  statement-default;
	done:
		1. && (pointer to code location) is an extension defined by GCC.
		2. unsigned long handles the case of n < 100 (n - 100 overflows to a large integer.)

1.5 Procedure

Procedure is an important abstraction having different forms: function, subroutine, method, handler, etc. Each procedure has its own stack frame. For most procedures, stack frames are aligned to 16.



earlier frames	...	
stack frame of caller P	...	
	parameter n	
	...	
	parameter 7	
	return address	
stack frame of callee Q	callee-preserved registers	
	local variables	
	parameters	⟵ `%rsp`

Figure 1: Stack frame structure

1.5.1 Transferring control

Return address is the address of the instruction after `call`.

Table 6: Control transfer instructions

`call`	Push return address onto stack; Set PC(`%rip`) to starting address of callee.
`ret`	Pop return address off stack; Set PC(`%rip`) to return address.

1.5.2 Passing arguments

- Arguments 1-6 are respectively put inside registers `%rdi, %rsi, %rdx, %rcx, %r8, %r9` or their counterparts of smaller sizes.
- Other arguments are on the stack, with argument 7 at the top. All on-stack arguments are aligned to 8.
- Argument k (k>6) is at address `%rsp+8*(k-6)`.

1.5.3 Local storage

Local data needs to be stored on stack in the following cases:

- registers are not enough to hold all local data;
- the & operator is used on a variable;
- arrays or structs are used as local variables.

On-stack local variables do not have to be aligned to 8. In general, basic variables (integers, pointers) of size K bytes should be aligned to K.

Callee-preserved registers (%rbx, %rbp, %r12-%r15) should be saved on stack before being used inside the callee. Other registers are caller-preserved, i.e. if the caller expects their values to be available after calling a procedure, the caller should save them on stack before calling the procedure.

1.6 Floating point instructions

- 16 `%ymm` registers, each of 256 bits, are used to store floating pointer numbers.
- When dealing with scalar FP numbers, we use `%xmm` registers, i.e. lowest 128 bits of `%ymm` registers to store them. Only the lowest 32 (float) or 64 (double) bits are used.
- `%xmm0` is used to store the FP return value.
- `%xmm0-7` are used to store 1st-8th FP arguments.
- `%xmm8-15` are caller-preserved registers.
- `aps` = aligned packed single, `apd` = aligned packed double.
- `vcvttss2si` = Vector ConVerT Truncation Scalar Single-precision 2 Signed Int.
- Floating point comparison sets 3 condition codes: `CF, ZF` and `PF` (`P` = parity). If at least one of the two arguments is `NaN`, then there is no order, and `PF` is set to 1.

Table 7: Condition codes of FP comparison

	`CF`	`ZF`	`PF`
No order	1	1	1
$S_2 < S_1$	1	0	0
$S_2 = S_1$	0	1	0
$S_2 > S_1$	0	0	0

Table 8: Floating point instructions

`vmovss`	M_{32}		X	float, memory → register
`vmovss`	X		M_{32}	float, register → memory
`vmovsd`	M_{64}		X	double, memory → register
`vmovsd`	X		M_{32}	double, register → memory
`vmovaps`	X		X	float, register → register.
`vmovapd`	X		X	double, register → register.
`vcvttss2si`	X/M_{32}		R_{32}	float → int.
`vcvttsd2si`	X/M_{64}		R_{32}	double → int.
`vcvttss2siq`	X/M_{32}		R_{64}	float → long.
`vcvttsd2siq`	X/M_{64}		R_{64}	double → long.
`vcvtsi2ss`	M_{32}/R_{32}	X	X	int → float.
`vcvtsi2sd`	M_{32}/R_{32}	X	X	int → double.
`vcvtsi2ssq`	M_{64}/R_{64}	X	X	long → float.
`vcvtsi2sdq`	M_{64}/R_{64}	X	X	long → double.
`vunpcklps %xmm0 %xmm0 %xmm0`				float → double (weird but it is what gcc does)
`vcvtps2pd %xmm0 %xmm0`				

to be continued

continue

`vmovddup %xmm0 %xmm0`
`vcvtpd2psx %xmm0 %xmm0`
	double → float(weird but it is what gcc does)

`vaddss vaddsd`	$D \leftarrow S_2 + S_1$
`vsubss vsubsd`	$D \leftarrow S_2 - S_1$
`vmulss vmulsd`	$D \leftarrow S_2 \times S_1$
`vdivss vdivsd`	$D \leftarrow S_2 \div S_1$
`vmaxss vmaxsd`	$D \leftarrow \max(S_2, S_1)$
`vminss vminsd`	$D \leftarrow \min(S_2, S_1)$
`sqrtss sqrtsd`	$D \leftarrow \sqrt{S_1}$
`vxorps vxorpd`	$D \leftarrow S_2\hat{}\,S_1$
`vandps vandpd`	$D \leftarrow S_2 \& S_1$
`ucomiss`	compare float according to $S_2 - S_1$
`ucomisd`	compare double according to $S_2 - S_1$