Useful x86_64 instructions

This is a very small subset of the available instructions but should be enough for your purposes.

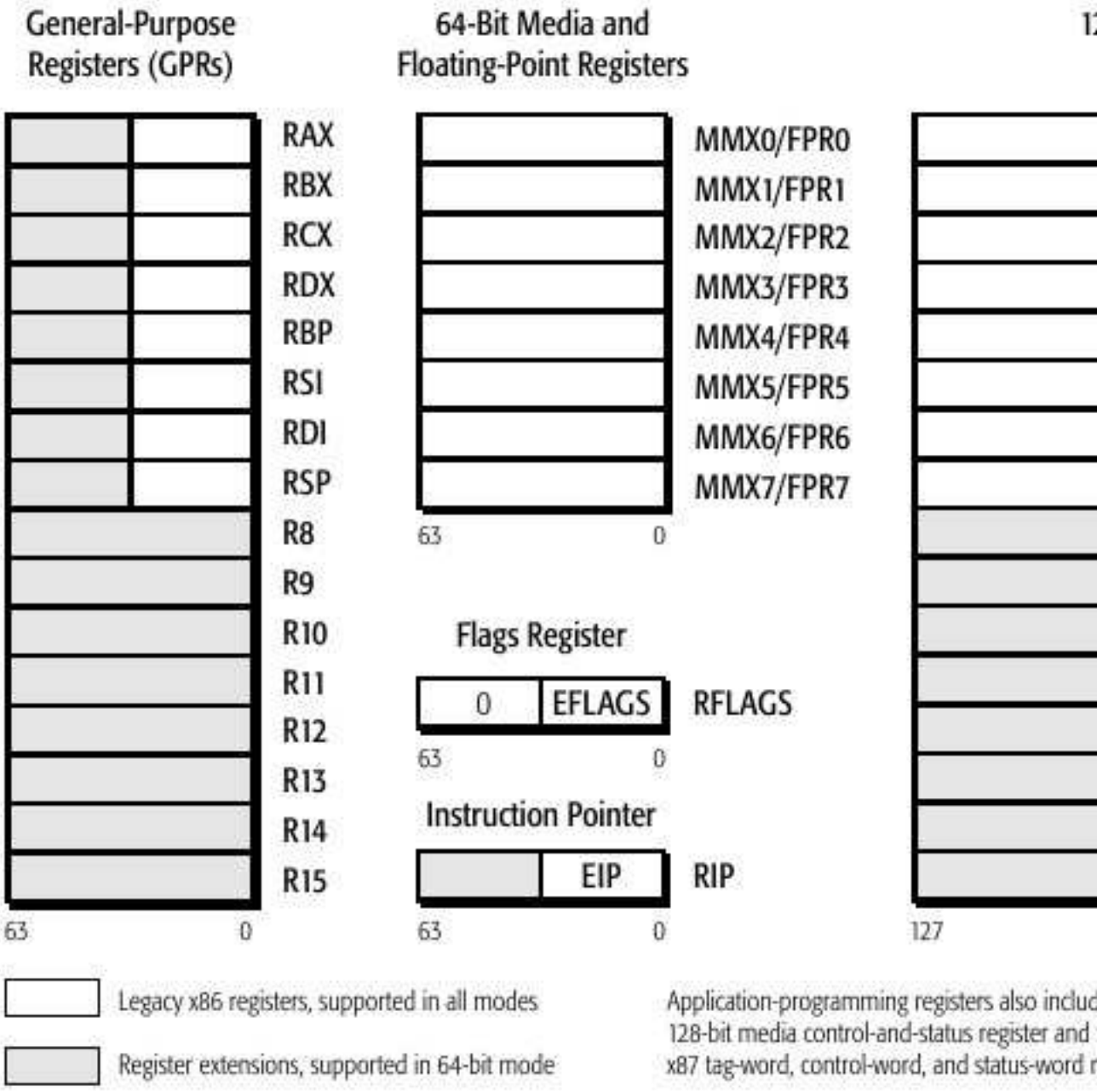

Figure 1-1. Application-Programming Reg

Data movement

mov mem, reg/lit

example

```
mov 12(%rbp),%rax
movd $12, 4(%rsi)
```

means

store the right operand in the memory location on the left

if the length of the value being moved is ambiguous the mov instruction must be converted to movb, movw, movd, movl etc.

mov mem/reg/lit,reg

example

```
mov  1,%ebx
mov 0(%rsi,%rbp), %r9
mov %eax,%ebx
```

means

load the left operand into the register on the right

Floating point

The AMD64 architecture provides three floating-point instruction subsets, using three distinct register sets:

- 128-Bit Media Instructions support 32-bit single-precision and 64-bit double-precision floating- point operations, in addition to integer operations. Operations on both vector data and scalar data are supported, with a dedicated floating-point exception-reporting mechanism. These floating- point operations comply with the IEEE-754 standard.

- 64-Bit Media Instructions (the subset of 3DNow! technology instructions) support single- precision floating-point operations. Operations on both vector data and scalar data are supported, but these instructions do not support floating-point exception reporting.

- x87 Floating-Point Instructions support single-precision, double-precision, and 80-bit extended-precision floating-point operations. Only scalar data are supported, with a dedicated floating-point exception-reporting mechanism. The x87 floating-point instructions contain special instructions for performing trigonometric and logarithmic transcendental operations. The single-precision and double-precision floating-point operations comply with the IEEE-754 standard.

Maximum floating-point performance can be achieved using the 128-bit media instructions. One of these vector instructions can support up to four single-precision (or two double-precision) operations in parallel.

movss mem, reg

example

```
movss 12(%rbp),%xmm0
```

means

store the left operand in right. The right operand is the bottom 32 bits of an xmm register.

movss reg, mem/reg

example

```
movss %xmm1,0(%rsi,%rbx)
movss %xmm1,xxm2
```

means

load the left operand into the right, the left operand is the lower 32 bits of a xmm register and the data should be a 32 bit float

movups mem, reg

example

```
movss 12(%rbp),%xmm0
```

means

copy the leftt operand to the right operatnd. The right operand is a 128 bit xmm register.

movss reg, mem

example

```
movups %xmm1,0(%rsi,%rdi)
```

means

store the leftt operand in the register in the rightt, the left operand is a 128 bit xmm register

push mem/reg/lit

example

```
pushq $10
pushq 40(%rsi)
push %rcx
```

means

push the operand on stack, pre-decrementing the esp register by 8

pop mem/reg

example

```
popq 32(%rsi)
pop %rcx
```

means

the operand is assigned the value on the top of stack and the stack pointer is then incremented by 8

fld<len> mem

example

```
    flds 40(%rsi)
```

means

the operand which is pushed on the fpu stack <len> takes on the value `s` for single precision and `l` for double precision, the number must be in 32 bit or 64 bit floating point

fild mem

example

```
fild  40(%rsi)
```

means

the 32bit integer operand is pushed on the fpu stack as a floating point number

fstp<len> mem

example

```
    fstps   40(%rsi)
```

means

the operand is assigned the 32bit floating point value on the fpu stack the fpu stack is then popped

fistp<len> mem

example

```
fistpd 40(%rsi)
```

means

the 32bit floating point value on the fpu stack is converted to an integer and stored in the operand, the fpu stack is then popped.

Arithmetic

Integer arithmetic instructions can be divided into 3 classes

1. Add, subtract, and, or, xor.These are treated absolutely regularly as two operand instructions as shown below in section .

2. Multiply, this comes in both 2 and 3 operand forms.

3. Divide and Modulus, these are irregular and make use of specific registers

Regular integer arithmetic

These take the form

operation *src*,*dest*

and mean *dest*:= *dest* operation *src*

the following operation codes are allowed

```
add, sub, and, or, xor
```

The table shows the allowed combinations of destination and source

Operand combinations for regular arithmetic

dest	src
register	register
register	constant
register	memory
memory	register
memory	constant

Examples

```
add $5,%rsp
sub %eax, %ebx
and 12(%rax),%rdi
addq $1,0(%rsi)
add %esi,0(%rdi)
```

Multiply

imul reg/mem,reg

This is functionally the same as the regular 2 operand integer arithmetic instructions.

Example

```
imuld  26(%rbp),%rdx
rdx * mem(26+rbp) →rdx
```

imul const,reg,reg

This three operand form is particularly useful for computing array offsets.

Example

```
imul  $16,%rbx,%rax
16 * rbx →rax
```

Divide/modulus

A single instruction is used for both division and modulus.

idiv reg

The 128 bit value in rdx:rax is divided by the operand, the quotient is placed in rax, and the remainder is placed in rdx.

Example

```
idiv %r11
```

Floating point arithmetic

The floating point stack can be used to perform arithmetic in a postfix manner. The following fpu opcodes operate on the top two items on the fpu stack:

```
faddp st1
fsubp st1
fdivp st1
fmulp st1
```

These perfrom an operation between the top of the fpu stack (st0) and st1, store the result in st1, then pop the stack so that st1 becomes the new top of stack. Bear in mind that the maximum depth of the fpu stack is 8. Operations are performed using 80bit internal floating point representation.

Vector arithmetic

It is possible to perform parallel operations on vectors of 32 bit floats using the xmm registers. These instructions have the general format

*operation*PS *xmmreg,xmmreg*

For example

```
mulps %xmm5,%xmm0
```

the suffix PS stands for Packed Single precison floats. In this case the 4 floats in xmm0 are multiplied by the corresponding floats in xmm5 and the result stored in xmm0.

The other useful vector arithmetic instructions in this context are:

```
addps, subps, divps
```

These instructions also exist in a memory to register form but for these to be used you have to guarantee that the operands are aligned on 16 byte memory boundaries. Since this is complicated to ensure, I suggest that you restrict yourself to the register to register forms of these instructions.

Scalar arithmetic

It is also possible to perform scalar arithmetic in the low order 32 bit words of the xmm registers. For instance, you can do all of the vector operations by using the subscript SS standing for Scalar Single precision after the operation thus:

```
addss %xmm0,%xmm2
```

would add the bottom 32 bit float in xmm0 to the bottom float in xmm2 and leave the result in xmm2.

Conversion instructions

operation	dest	src
cvtsi2ss	xmm register	general register
cvtss2si	general register	xmm register

If you are going to use these scalar instructions it is worth taking note of the conversion instructions `cvtsi2ss` and `cvtss2si` which convert signed doubleword integers to single precision floats and vice versa.

Examples

```
cvtss2si %xmm4, %ebx
cvtsi2ss %eax,%xmm3
```

Integer comparisons

Comparison instructions exist which will place the results of comparison in the flags. The `cmp` instruction compares two integers.

Examples

```
cmp 12,%rax
cmp %rax, %rcx
```

Set

The result of the comparison is written to the flags and can be used either by a SET instruction or by a conditional jump instruction.

For instance to test if the eax register was less then 10 we could write

```
cmp %rbx,%rax
setl %al        #al:= rax< rbx
```

At the end of this the al register will contain a boolean value of 1 if rax had been less than rbx and 0 if it had been greater than rbx. The suffixes used by the SET instruction indicate which comparison is being tested. The suffixes that are most likely to be of use to you are L, G and E standing for Less than, Greater than, and Equal.

Branches

Branches can be unconditional and direct:

```
jmp lab
```

or uncoditional and indirect:

```
jmp dword[ebp+10]
```

or conditional on a condition code and direct:

```
jl lab1
jg lab3
je lab4
```

Calls

Calls can be direct:

```
call lab
```

or indirect:

```
call 10(%rbp)
```

in either case the current value of the `rip` register is pushed on the stack and the `rip` register loaded from the operand. Returns are perfomed using the `ret` instruction which pops the top of stack into the `rip` register.