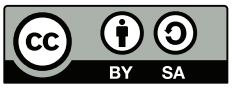
Instruction set of the ULM (Ulm Lecture Machine)



June 9, 2023

Contents

Chapter 1

Description of the ULM

1.1 Data Types

Binary digits are called *bits* and have the value 0 or 1. A *bit pattern* is a sequence of bits. For example

$$X := x_{n-1} \dots x_0 \text{ with } x_k \in \{0, 1\} \text{ for } 0 \leq k < n$$

denotes a bit pattern X with n bits. The number of bits in bit pattern is also called its size or width. The ULM architecture defines a *byte* as a bit pattern with 8 bits. Table 1.1 lists ULM's definitions for *word*, *long word*, *quad word* that refer to specific sizes of bit patterns.

1.2 Expressing the Interpretation of a Bit Pattern

For a bit pattern $X = x_{n-1} \dots x_0$ its *unsigned integer* value is expressed and defined through

$$u(X) = u(x_{n-1} \dots x_0) := \sum_{k=0}^{n-1} x_k \cdot 2^k$$

Signed integer values are represented using the *two's complement* and in this respect the notation

$$s(X) = s(x_{n-1}x_{n-2} \dots x_0) := \begin{cases} u(x_{n-2} \dots x_0), & \text{if } x_{n-1} = 0, \\ u(x_{n-2} \dots x_0) - 2^{n-1}, & \text{else} \end{cases}$$

is used.

1.3 Registers and Virtual Memory

The ULM has 256 registers denoted as %0x00, ..., %0xFF. Each of these registers has a width of 64 bits. The %0x00 is a special purpose register and also denoted as *zero register*. Reading form the zero register always gives a bit pattern where all bits have value 0 (zero bit pattern). Writing to the zero register has no effect.

The (virtual) memory of the ULM is an array of 2^{64} memory cells. Each memory cell can store exactly one byte. Each memory cell has an index which is called its *address*. The address is in the range from 0 to 2^{64-1} and the first memory cell of the array has address 0. In notations $M_1(a)$ denotes the memory cell with address a.

Data Size	Size in Bytes	Size in Number of Bits
Bytes	-	8
Word	2	16
Long Word	4	32
Quad Word	8	64

Table 1.1: Names for specific sizes of bit patterns.

1.3.1 Endianness

For referring to data in memory in quantities of words, long words and quad words the definitions

$$\begin{array}{lcl} M_2(a) & := & M_1(a)M_1(a+1) \\ M_4(a) & := & M_2(a)M_2(a+2) \\ M_8(a) & := & M_4(a)M_4(a+4) \end{array}$$

are used. The ULM architecture is a *big endian* machine. Therefore we have the equalities

$$\begin{array}{lcl} u(M_2(a)) & = & u(M_1(a)M_1(a+1)) \\ u(M_4(a)) & = & u(M_2(a)M_2(a+2)) \\ u(M_8(a)) & = & u(M_4(a)M_4(a+4)) \end{array}$$

1.3.2 Alignment of Data

A quantity of k bytes are aligned in memory if they are stored at an address which is a multiple of k, i. e.

$$M_k(a) \text{ is aligned } \Leftrightarrow\ a \bmod k = 0$$

Chapter 2

Directives

2.1 .align <expr>

Pad the location counter (in the current segment) to a multiple of <expr>.

2.2 .bss

Set current segment to the BSS segment.

2.3 .byte <expr>

Expression is assembled into next byte.

2.4 .data

Set current segment to the data segment.

2.5 .equ <ident>, <expr>

Updates the symbol table. Sets the value of <ident> to <expr>.

2.6 .global <ident>

Updates the symbol table. Makes the symbol <ident> visible to the linker.

2.7 .globl <ident>

Equivalent to *.globl <ident>*:
Updates the symbol table. Makes the symbol <ident> visible to the linker.

2.8 .long <expr>

Expression <expr> is assembled into next long word (4 bytes).

2.9 .space <expr>

Emits <expr> bytes. Each byte with value 0x00.

2.10 .string <string-literal>

Emits bytes for the zero-terminated <string-literal>.

2.11 .text

Set current segment to the text segment.

2.12 .word <expr>

Expression <expr> is assembled into next word (2 bytes).

2.13 .quad <expr>

Expression <expr> is assembled into next quad word (8 bytes).

Chapter 3

Instructions

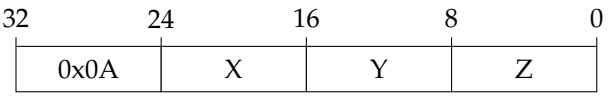
3.1 addq

Integer addition.

3.1.1 Assembly Notation

addq X, %Y, %Z

Format



Effect

$$(u(\%Y) + u(X)) \bmod 2^{64} \rightarrow u\,(\%Z)$$

Updates the status flags:

Flag	Condition
ZF	$u(\%Y) + u(X) = 0$
CF	$u(\%Y) + u(X) \geq 2^{64}$
OF	$s(\%Y) + s(X) \notin \{-2^{63}, \ldots, 2^{63}\}$
SF	$s(\%Y) + s(X) < 0$

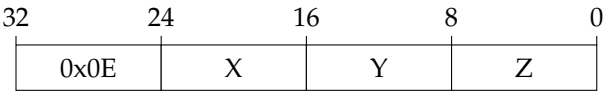
3.1.2 Assembly Notation

addq %X, %Y, %Z

Alternative Assembly Notation

movq %X, %Z

Format



Effect

$$(u(\%Y) + u(\%X)) \bmod 2^{64} \rightarrow u\,(\%Z)$$

Updates the status flags:

Flag	Condition
ZF	$u(\%Y) + u(\%X) = 0$
CF	$u(\%Y) + u(\%X) \geq 2^{64}$
OF	$s(\%Y) + s(\%X) \notin \{-2^{63}, \ldots, 2^{63}\}$
SF	$s(\%Y) + s(\%X) < 0$

3.2 divq

64-bit unsigned integer division

3.2.1 Assembly Notation

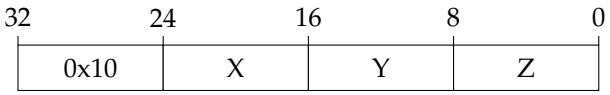
divq X, %Y, %Z

Purpose

Uses the 128-bit unsigned integer divider. The 64-bits stored in %Y are therefore zero extended with %0 to 128 bits and then used as numerator. The immediate value X is used as unsigned denominator.

The result of the operation is the quotient and the remainder. These results will be stored in a register pair: The quotient is stored in %Z and the remainder in %{Z + 1}.

Format



Effect

For the unsigned 128-bit numerator

$$b := u\,(\%0\%Y)$$

and unsigned 64-bia denominator

$$a := u(X)$$

computes the divisor

$$\left\lfloor \frac{b}{a} \right\rfloor \bmod 2^{128} \rightarrow u\,(\%0\%Z)$$

and the remainder

$$b \bmod a \rightarrow u\,(\%\{u(Z)+1\})$$

3.2.2 Assembly Notation

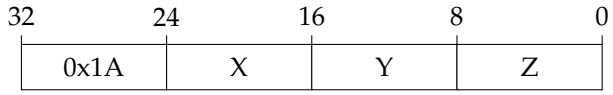
divq %X, %Y, %Z

Purpose

Uses the 128-bit unsigned integer divider. The 64-bits stored in %Y are therefore zero extended with %0 to 128 bits and then used as numerator. The register value %X is used as 64-bit unsigned denominator.

The result of the operation is the quotient and the remainder. These results will be stored in a register pair: The quotient is stored in %Z and the remainder in %{Z + 1}.

Format



Effect

For the unsigned 128-bit numerator

$$b := u\,(\%0\%Y)$$

and unsigned 64-bia denominator

$$a := u(\%X)$$

computes the divisor

$$\left\lfloor \frac{b}{a} \right\rfloor \bmod 2^{128} \rightarrow u\,(\%0\%Z)$$

and the remainder

$$b \bmod a \rightarrow u\,(\%\{u(Z)+1\})$$

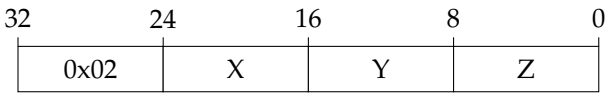
3.3 getc

Get character

3.3.1 Assembly Notation

getc %X

Format



Effect

$s(ulm\_readChar()) \wedge_b 255 \bmod 2^{64} \rightarrow u\ (\%X)$

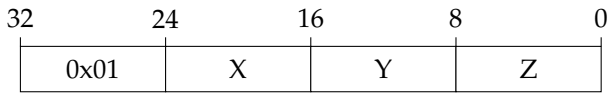
3.4 halt

Halt program

3.4.1 Assembly Notation

halt %X

Format



Effect

halt program execution with exit code $u(\%X) \bmod 2^8$

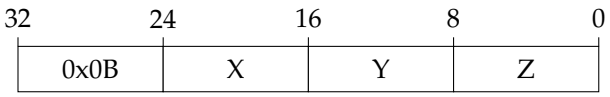
3.5 imulq

64-bit unsigned and signed integer multiplication.

3.5.1 Assembly Notation

imulq %X, %Y, %Z

Format



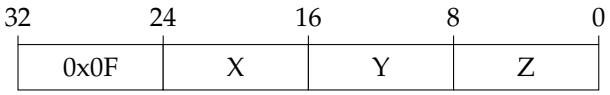
Effect

$$u(\%X) \cdot u(\%Y) \bmod 2^{64} \rightarrow u\,(\%Z)$$

3.5.2 Assembly Notation

imulq X, %Y, %Z

Format



Effect

$$u(X) \cdot u(\%Y) \bmod 2^{64} \rightarrow u\,(\%Z)$$

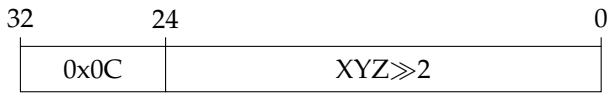
3.6 ja

Jump if above (conditional jump).

3.6.1 Assembly Notation

ja XYZ

Format



Effect

If the condition

$$ZF = 0 \wedge CF = 0$$

evaluates to true then

$$(u(\%IP) + s(XYZ)) \bmod 2^{64} \rightarrow u(\%IP)$$

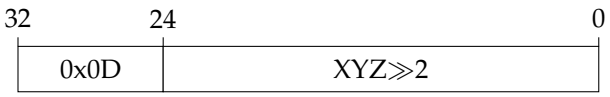
3.7 jb

Jump if below (conditional jump).

3.7.1 Assembly Notation

jb XYZ

Format



Effect

If the condition

$$CF = 1$$

evaluates to true then

$$(u(\%IP) + s(XYZ)) \bmod 2^{64} \rightarrow u(\%IP)$$

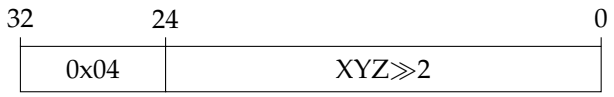
3.8 jmp

Jump (unconditional jump).

3.8.1 Assembly Notation

jmp XYZ

Format



Effect

$$(u(\%IP) + s(XYZ)) \bmod 2^{64} \rightarrow u(\%IP)$$

3.8.2 Assembly Notation

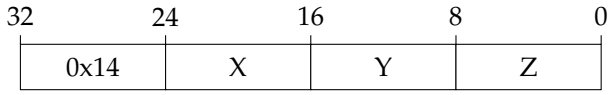
jmp %X, %Y

Alternative Assembly Notation

call %X, %Y

ret %X

Format



Effect

$$\begin{aligned} (u(\%IP) + 4) \bmod 2^{64} &\rightarrow u(\%Y) \\ u(\%X) &\rightarrow u(\%IP) \end{aligned}$$

3.9 jnz

Jump if not zero (conditional jump).

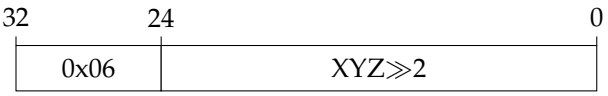
3.9.1 Assembly Notation

jnz XYZ

Alternative Assembly Notation

jne XYZ

Format



Effect

If the condition

$$ZF = 0$$

evaluates to true then

$$(u(\%IP) + s(XYZ)) \bmod 2^{64} \rightarrow u(\%IP)$$

3.10 jz

Jump if zero (conditional jump).

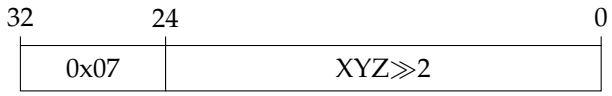
3.10.1 Assembly Notation

jz XYZ

Alternative Assembly Notation

je XYZ

Format



Effect

If the condition

$$ZF = 1$$

evaluates to true then

$$(u(\%IP) + s(XYZ)) \bmod 2^{64} \rightarrow u(\%IP)$$

3.11 ldfp

Load from Pool

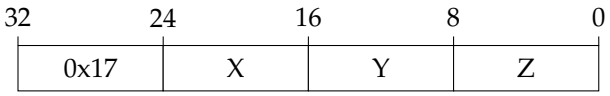
3.11.1 Assembly Notation

ldfp Y(%X), %Z

Alternative Assembly Notation

ldfp (%X), %Z

Format



Effect

$$u\left(\mathrm{M}_8\left(addr\right)\right) \rightarrow u\left(\%Z\right) \text{ with } addr = \left(u(Y) \cdot 8 + u(\%X)\right) \bmod 2^{64}$$

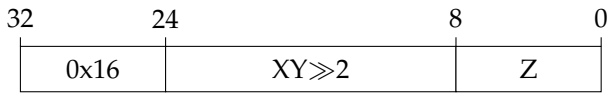
3.12 ldpa

Load Pool Address

3.12.1 Assembly Notation

ldpa XY, %Z

Format



Effect

$$(u(\%IP) + s(XY)) \bmod 2^{64} \rightarrow u\,(\%Z)$$

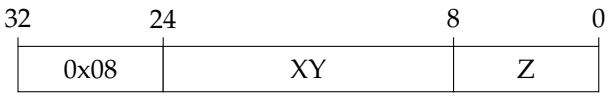
3.13 ldzwq

Load zero extended word to quad word register.

3.13.1 Assembly Notation

ldzwq XY, %Z

Format



Effect

$u(XY) \bmod 2^{64} \rightarrow u\,(\%Z)$

3.14 movb

Move a byte from a register to memory (store instruction).

3.14.1 Assembly Notation

movb %X, Y(%Z)

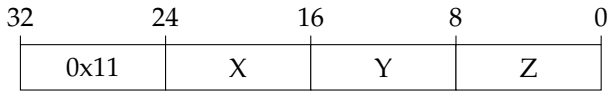
Alternative Assembly Notation

movb %X, (%Z)

Purpose

The least significant byte in %X gets stored at address %Z.

Format



Effect

$$u(\%X) \bmod 2^8 \rightarrow u\left(\mathrm{M}_1\left(addr\right)\right) \text{ with } addr = \left(s(Y) + u(\%Z)\right) \bmod 2^{64}$$

3.15 movq

Move quad word

3.15.1 Assembly Notation

movq Y(%X), %Z

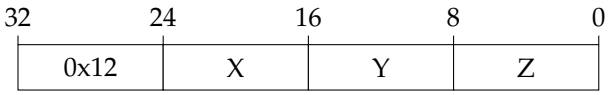
Alternative Assembly Notation

movq (%X), %Z

Purpose

Fetches a quad word from address %X into register %Z

Format



Effect

$$u\left(\mathrm{M}_8\left(addr\right)\right) \rightarrow u\left(\%Z\right) \text{ with } addr = \left(s(Y) + u(\%X)\right) \bmod 2^{64}$$

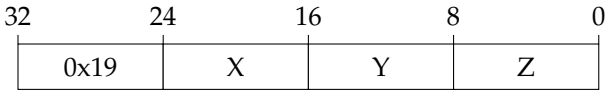
3.15.2 Assembly Notation

movq %X, Y(%Z)

Alternative Assembly Notation

movq %X, (%Z)

Format



Effect

$$u(\%X) \bmod 2^{64} \rightarrow u\left(\mathrm{M}_8\left(addr\right)\right) \text{ with } addr = \left(s(Y) + u(\%Z)\right) \bmod 2^{64}$$

3.16 movzbq

Move zero extended byte to quad word register (fetch instruction).

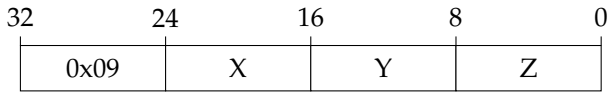
3.16.1 Assembly Notation

movzbq Y(%X), %Z

Alternative Assembly Notation

movzbq (%X), %Z

Format



Effect

$$u\left(\mathrm{M}_1\left(addr\right)\right) \rightarrow u\left(\%Z\right) \text{ with } addr = \left(s(Y) + u(\%X)\right) \bmod 2^{64}$$

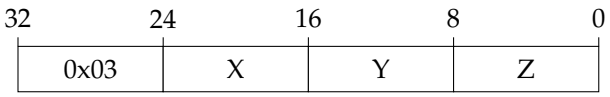
3.17 putc

Put character

3.17.1 Assembly Notation

putc %X

Format



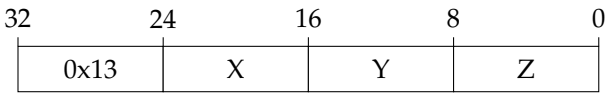
Effect

$ulm\_printChar(u(\%X) \wedge_b 255)$

3.17.2 Assembly Notation

putc X

Format



Effect

$ulm\_printChar(u(X) \wedge_b 255)$

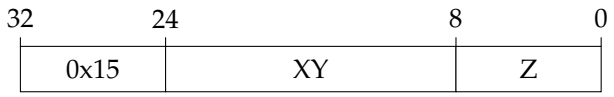
3.18 shldwq

Shift (destination register) and load word

3.18.1 Assembly Notation

shldwq XY, %Z

Format



Effect

$$u(\%Z) << 16 | u(XY) \bmod 2^{64} \rightarrow u\,(\%Z)$$

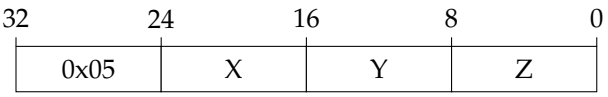
3.19 subq

Integer subtraction.

3.19.1 Assembly Notation

subq X, %Y, %Z

Format



Effect

$$(u(\%Y) - u(X)) \bmod 2^{64} \rightarrow u\,(\%Z)$$

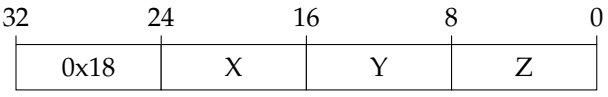
Updates the status flags:

Flag	Condition
ZF	$u(\%Y) - u(X) = 0$
CF	$u(\%Y) - u(X) < 0$
OF	$s(\%Y) - s(X) \notin \{-2^{63}, \ldots, 2^{63}\}$
SF	$s(\%Y) - s(X) < 0$

3.19.2 Assembly Notation

subq %X, %Y, %Z

Format



Effect

$$(u(\%Y) - u(\%X)) \bmod 2^{64} \rightarrow u\,(\%Z)$$

Updates the status flags:

Flag	Condition
ZF	$u(\%Y) - u(\%X) = 0$
CF	$u(\%Y) - u(\%X) < 0$
OF	$s(\%Y) - s(\%X) \notin \{-2^{63}, \ldots, 2^{63}\}$
SF	$s(\%Y) - s(\%X) < 0$

Chapter 4

ISA Source File for the ULM Generator

```
RRR (OP u 8) (X u 8) (Y u 8) (Z u 8)
RSR (OP u 8) (X u 8) (Y s 8) (Z u 8)
J26 (OP u 8) (XYZ j 24)
U16R (OP u 8) (XY u 16) (Z u 8)
J18R (OP u 8) (XY j 16) (Z u 8)


0x01 RRR
: halt %X
        ulm_halt(ulm_regVal(X));

0x02 RRR
: getc %X
        ulm_setReg(ulm_readChar() & 0xFF, X);

0x03 RRR
: putc %X
        ulm_printChar(ulm_regVal(X) & 0xFF);

0x04 J26
: jmp XYZ
        ulm_unconditionalRelJump(XYZ);

0x05 RRR
: subq X, %Y, %Z
        ulm_sub64(X, ulm_regVal(Y), Z);

0x06 J26
: jnz XYZ
: jne XYZ
        ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 0, XYZ);

0x07 J26
: jz XYZ
: je XYZ
```

```
36      ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 1, XYZ);
37
38  0x08 U16R
39  : ldzwq XY, %Z
40      ulm_setReg(XY, Z);
41
42  0x09 RSR
43  : movzbq Y(%X), %Z
44  : movzbq (%X), %Z
45      ulm_fetch64(Y, X, 0, 0, ULM_ZERO_EXT, 1, Z);
46
47  0x0A RRR
48  : addq X, %Y, %Z
49      ulm_add64(X, ulm_regVal(Y), Z);
50
51  0x0B RRR
52  : imulq %X, %Y, %Z
53      ulm_mul64(ulm_regVal(X), ulm_regVal(Y), Z);
54
55  0x0C J26
56  : ja XYZ
57      ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 0
58                             && ulm_statusReg[ULM_CF] == 0, XYZ);
59
60  0x0D J26
61  : jb XYZ
62      ulm_conditionalRelJump(ulm_statusReg[ULM_CF] == 1, XYZ);
63
64  0x0E RRR
65  : addq %X, %Y, %Z
66  : movq %X, %Z
67      ulm_add64(ulm_regVal(X), ulm_regVal(Y), Z);
68
69  0x0F RRR
70  : imulq X, %Y, %Z
71      ulm_mul64(X, ulm_regVal(Y), Z);
72
73  0x10 RRR
74  # Uses the 128−bit unsigned integer divider. The 64−bits stored in %Y are
75  # therefore zero extended with %0 to 128 bits and then used as numerator.
76  # The immediate value X is used as unsigned denominator.
77  #
78  # The result of the operation is the quotient and the remainder. These results
79  # will be stored in a register pair: The quotient is stored in %Z and the
80  # remainder in %{Z + 1}.
81  : divq X, %Y, %Z
82      ulm_div128(X, ulm_regVal(Y), ulm_regVal(0), Z, 0, Z + 1);
83
84  0x11 RSR
85  # The least significant byte in %X gets stored at address %Z.
86  : movb %X, Y(%Z)
87  : movb %X, (%Z)
88      ulm_store64(Y, Z, 0, 0, 1, X);
```

```
89
90  0x12 RSR
91  # Fetches a quad word from address %X into register %Z
92  : movq Y(%X), %Z
93  : movq (%X), %Z
94      ulm_fetch64(Y, X, 0, 0, ULM_ZERO_EXT, 8, Z);
95
96  0x13 RRR
97  : putc X
98      ulm_printChar(X & 0xFF);
99
100 # void ulm_absJump(uint64_t destAddr, ulm_Reg ret);
101 0x14 RRR
102 : jmp %X, %Y
103 : call %X, %Y
104 : ret %X
105     ulm_absJump(ulm_regVal(X), Y);
106
107 0x15 U16R
108 : shldwq XY, %Z
109     ulm_setReg(ulm_regVal(Z) << 16 | XY, Z);
110
111 0x16 J18R
112 : ldpa XY, %Z
113     ulm_setReg(ulm_ipVal() + XY, Z);
114
115 0x17 RRR
116 : ldfp Y(%X), %Z
117 : ldfp (%X), %Z
118     ulm_fetch64(Y * 8, X, 0, 0, ULM_ZERO_EXT, 8, Z);
119
120 0x18 RRR
121 : subq %X, %Y, %Z
122     ulm_sub64(ulm_regVal(X), ulm_regVal(Y), Z);
123
124 0x19 RSR
125 : movq %X, Y(%Z)
126 : movq %X, (%Z)
127     ulm_store64(Y, Z, 0, 0, 8, X);
128
129 0x1A RRR
130 # Uses the 128-bit unsigned integer divider. The 64-bits stored in %Y are
131 # therefore zero extended with %0 to 128 bits and then used as numerator.
132 # The register value %X is used as 64-bit unsigned denominator.
133 #
134 # The result of the operation is the quotient and the remainder. These results
135 # will be stored in a register pair: The quotient is stored in %Z and the
136 # remainder in %{Z + 1}.
137 : divq %X, %Y, %Z
138     ulm_div128(ulm_regVal(X), ulm_regVal(Y), ulm_regVal(0), Z, 0, Z + 1);
139
140
141 # General meaning of the mnemonics
```

```
142
143 @addq
144 # Integer addition.
145 @getc
146 # Get character
147 @divq
148 # 64–bit unsigned integer division
149 @halt
150 # Halt program
151 @imulq
152 # 64–bit unsigned and signed integer multiplication.
153 @ja
154 # Jump if above (conditional jump).
155 @jb
156 # Jump if below (conditional jump).
157 @je
158 # Jump if equal (conditional jump).
159 @jmp
160 # Jump (unconditional jump).
161 @jne
162 # Jump if not equal (conditional jump).
163 @jnz
164 # Jump if not zero (conditional jump).
165 @jz
166 # Jump if zero (conditional jump).
167 @ldzwq
168 # Load zero extended word to quad word register.
169 @putc
170 # Put character
171 @subq
172 # Integer subtraction.
173 @movb
174 # Move a byte from a register to memory (store instruction).
175 @movzbq
176 # Move zero extended byte to quad word register (fetch instruction).
177 @movq
178 # Move quad word
179 @shldwq
180 # Shift (destination register) and load word
181 @ldpa
182 # Load Pool Address
183 @ldfp
184 # Load from Pool
```