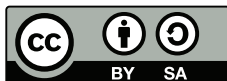


# Instruction set of the ULM (Ulm Lecture Machine)



May 19, 2023



# Contents

|          |                                                          |           |
|----------|----------------------------------------------------------|-----------|
| <b>1</b> | <b>Description of the ULM</b>                            | <b>4</b>  |
| 1.1      | Data Types . . . . .                                     | 4         |
| 1.2      | Expressing the Interpretation of a Bit Pattern . . . . . | 4         |
| 1.3      | Registers and Virtual Memory . . . . .                   | 4         |
| <b>2</b> | <b>Directives</b>                                        | <b>6</b>  |
| 2.1      | .align <expr> . . . . .                                  | 6         |
| 2.2      | .bss . . . . .                                           | 6         |
| 2.3      | .byte <expr> . . . . .                                   | 6         |
| 2.4      | .data . . . . .                                          | 6         |
| 2.5      | .equ <ident>, <expr> . . . . .                           | 6         |
| 2.6      | .global <ident> . . . . .                                | 6         |
| 2.7      | .globl <ident> . . . . .                                 | 6         |
| 2.8      | .long <expr> . . . . .                                   | 6         |
| 2.9      | .space <expr> . . . . .                                  | 7         |
| 2.10     | .string <string-literal> . . . . .                       | 7         |
| 2.11     | .text . . . . .                                          | 7         |
| 2.12     | .word <expr> . . . . .                                   | 7         |
| 2.13     | .quad <expr> . . . . .                                   | 7         |
| <b>3</b> | <b>Instructions</b>                                      | <b>8</b>  |
| 3.1      | addq . . . . .                                           | 9         |
| 3.2      | getc . . . . .                                           | 11        |
| 3.3      | halt . . . . .                                           | 12        |
| 3.4      | imulq . . . . .                                          | 13        |
| 3.5      | ja . . . . .                                             | 14        |
| 3.6      | jb . . . . .                                             | 15        |
| 3.7      | jmp . . . . .                                            | 16        |
| 3.8      | jnz . . . . .                                            | 17        |
| 3.9      | jz . . . . .                                             | 18        |
| 3.10     | ldzwq . . . . .                                          | 19        |
| 3.11     | movzbq . . . . .                                         | 20        |
| 3.12     | putc . . . . .                                           | 21        |
| 3.13     | subq . . . . .                                           | 22        |
| <b>4</b> | <b>ISA Source File for the ULM Generator</b>             | <b>23</b> |

# 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 from 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

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

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

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

### 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

#### Purpose

Adds an immediate value X to register %Y. Stores result in register %Z.

#### Format

|      |    |    |   |   |
|------|----|----|---|---|
| 32   | 24 | 16 | 8 | 0 |
| 0x0A | X  | Y  | Z |   |

#### 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     $s(\%Y) + s(X) \notin \{-2^{63}, \dots, 2^{63}\}$

SF     $s(\%Y) + s(X) < 0$

### 3.1.2 Assembly Notation

addq %X, %Y, %Z

#### Purpose

Adds register %X to register %Y. Stores result in register %Z.

#### Format

|      |    |    |   |   |
|------|----|----|---|---|
| 32   | 24 | 16 | 8 | 0 |
| 0x0E | X  | Y  | Z |   |

**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   | $s(\%Y) + s(\%X) \notin \{-2^{63}, \dots, 2^{63}\}$ |
| SF   | $s(\%Y) + s(\%X) < 0$                               |

### 3.2 getc

Get character

#### 3.2.1 Assembly Notation

getc %X

**Purpose**

Read character into %X

**Format**

|      |    |    |   |   |
|------|----|----|---|---|
| 32   | 24 | 16 | 8 | 0 |
| 0x02 | X  | Y  | Z |   |

**Effect**

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

### 3.3 halt

Halt program

#### 3.3.1 Assembly Notation

halt %X

##### Purpose

Halt with exit code %X

##### Format

|      |    |    |   |   |
|------|----|----|---|---|
| 32   | 24 | 16 | 8 | 0 |
| 0x01 | X  | Y  | Z |   |

##### Effect

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

## 3.4 imulq

64-bit unsigned and signed integer multiplication.

### 3.4.1 Assembly Notation

`imulq %X, %Y, %Z`

#### Format

|      |    |    |   |   |
|------|----|----|---|---|
| 32   | 24 | 16 | 8 | 0 |
| 0x0B | X  | Y  | Z |   |

#### Effect

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

### 3.4.2 Assembly Notation

`imulq X, %Y, %Z`

#### Format

|      |    |    |   |   |
|------|----|----|---|---|
| 32   | 24 | 16 | 8 | 0 |
| 0x0F | X  | Y  | Z |   |

#### Effect

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

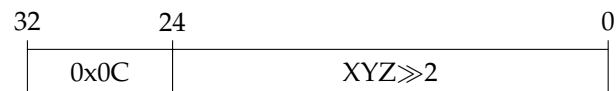
## 3.5 ja

Jump if above (conditional jump).

### 3.5.1 Assembly Notation

ja XYZ

**Format**



**Effect**

If the condition

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

evaluates to true then

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

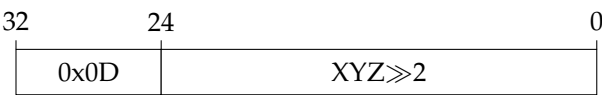
### 3.6 j**b**

Jump if below (conditional jump).

#### 3.6.1 Assembly Notation

j**b** XYZ

**Format**



**Effect**

If the condition

$$CF = 1$$

evaluates to true then

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

## 3.7 jmp

Jump (unconditional jump).

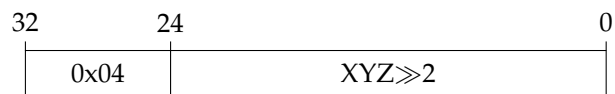
### 3.7.1 Assembly Notation

jmp XYZ

#### Purpose

Jump forward or backward

#### Format



#### Effect

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

## 3.8 jnz

Jump if not zero (conditional jump).

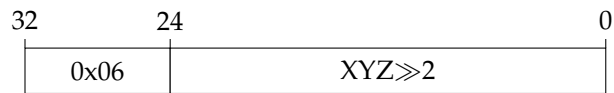
### 3.8.1 Assembly Notation

jnz XYZ

#### Alternative Assembly Notation

jne XYZ

#### Format



#### Effect

If the condition

$$ZF = 0$$

evaluates to true then

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

## 3.9 jz

Jump if zero (conditional jump).

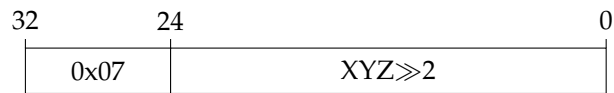
### 3.9.1 Assembly Notation

jz XYZ

#### Alternative Assembly Notation

je XYZ

#### Format



#### Effect

If the condition

$$ZF = 1$$

evaluates to true then

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

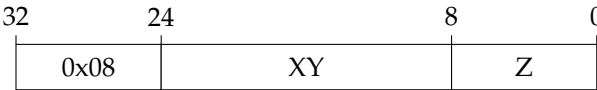
### 3.10 ldzwq

Load zero extended word to quad word register.

#### 3.10.1 Assembly Notation

ldzwq XY, %Z

**Format**



**Effect**

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

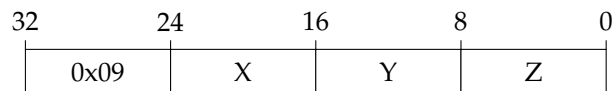
### 3.11 movzbq

Move zero extended byte to quad word register.

#### 3.11.1 Assembly Notation

movzbq (%X), %Z

##### Format



##### Effect

$u(M_1(addr)) \rightarrow u(\%Z)$  with  $addr = u(\%X) \bmod 2^{64}$

## 3.12 putc

Put character

### 3.12.1 Assembly Notation

putc %X

**Purpose**

Print character from %X

**Format**

|      |    |    |   |   |
|------|----|----|---|---|
| 32   | 24 | 16 | 8 | 0 |
| 0x03 | X  | Y  | Z |   |

**Effect**

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

### 3.13 subq

Integer subtraction.

#### 3.13.1 Assembly Notation

subq X, %Y, %Z

##### Purpose

Subtract immediate value X from register %Y. Store result in %Z

##### Format

|      |    |    |   |   |
|------|----|----|---|---|
| 32   | 24 | 16 | 8 | 0 |
| 0x05 | X  | Y  | Z |   |

##### 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     $s(\%Y) - s(X) \notin \{-2^{63}, \dots, 2^{63}\}$

SF     $s(\%Y) - s(X) < 0$

## Chapter 4

# ISA Source File for the ULM Generator

---

```
1  RRR (OP u 8) (X u 8) (Y u 8) (Z u 8)
2  J26 (OP u 8) (XYZ j 24)
3  U16R (OP u 8) (XY u 16) (Z u 8)
4
5
6  0x01 RRR
7  # Halt with exit code %X
8  : halt %X
9      ulm_halt(ulm_regVal(X));
10
11 0x02 RRR
12 # Read character into %X
13 : getc %X
14     ulm_setReg(ulm_readChar() & 0xFF, X);
15
16 0x03 RRR
17 # Print character from %X
18 : putc %X
19     ulm_printChar(ulm_regVal(X) & 0xFF);
20
21 0x04 J26
22 # Jump forward or backward
23 : jmp XYZ
24     ulm_unconditionalRelJump(XYZ);
25
26 0x05 RRR
27 # Subtract immediate value X from register %Y. Store result in %Z
28 : subq X, %Y, %Z
29     ulm_sub64(X, ulm_regVal(Y), Z);
30
31 0x06 J26
32 : jnz XYZ
33 : jne XYZ
34     ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 0, XYZ);
35
```

```

36 0x07 J26
37 : jz XYZ
38 : je XYZ
39     ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 1, XYZ);
40
41 0x08 U16R
42 : ldzwq XY, %Z
43     ulm_setReg(XY, Z);
44
45 0x09 RRR
46 : movzbq (%X), %Z
47     ulm_fetch64(0, X, 0, 0, ULM_ZERO_EXT, 1, Z);
48
49 0x0A RRR
50 # Adds an immediate value X to register %Y. Stores result in register %Z.
51 : addq X, %Y, %Z
52     ulm_add64(X, ulm_regVal(Y), Z);
53
54 0x0B RRR
55 : imulq %X, %Y, %Z
56     ulm_mul64(ulm_regVal(X), ulm_regVal(Y), Z);
57
58 0x0C J26
59 : ja XYZ
60     ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 0
61                             && ulm_statusReg[ULM_CF] == 0, XYZ);
62
63 0x0D J26
64 : jb XYZ
65     ulm_conditionalRelJump(ulm_statusReg[ULM_CF] == 1, XYZ);
66
67 0x0E RRR
68 # Adds register %X to register %Y. Stores result in register %Z.
69 : addq %X, %Y, %Z
70     ulm_add64(ulm_regVal(X), ulm_regVal(Y), Z);
71
72 0x0F RRR
73 : imulq X, %Y, %Z
74     ulm_mul64(X, ulm_regVal(Y), Z);
75
76
77 # General meaning of the mnemonics
78
79 @addq
80 # Integer addition.
81 @getc
82 # Get character
83 @halt
84 # Halt program
85 @imulq
86 # 64-bit unsigned and signed integer multiplication.
87 @ja
88 # Jump if above (conditional jump).

```

---

```
89  @jb
90  # Jump if below (conditional jump).
91  @je
92  # Jump if equal (conditional jump).
93  @jmp
94  # Jump (unconditional jump).
95  @jne
96  # Jump if not equal (conditional jump).
97  @jnz
98  # Jump if not zero (conditional jump).
99  @jz
100 # Jump if zero (conditional jump).
101 @ldzwq
102 # Load zero extended word to quad word register.
103 @putc
104 # Put character
105 @subq
106 # Integer subtraction.
107 @movzbq
108 # Move zero extended byte to quad word register.
```

---