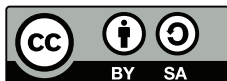


Instruction set of the ULM (Ulm Lecture Machine)



May 17, 2024

Contents

1	Description of the ULM	5
1.1	Data Types	5
1.2	Expressing the Interpretation of a Bit Pattern	5
1.3	Registers and Virtual Memory	5
2	Directives	7
2.1	.align <expr>	7
2.2	.bss	7
2.3	.byte <expr>	7
2.4	.data	7
2.5	.equ <ident>, <expr>	7
2.6	.global <ident>	7
2.7	.globl <ident>	7
2.8	.long <expr>	7
2.9	.space <expr>	8
2.10	.string <string-literal>	8
2.11	.text	8
2.12	.word <expr>	8
2.13	.quad <expr>	8
3	Instructions	9
3.1	addq	10
3.2	andq	12
3.3	getc	13
3.4	halt	14
3.5	ja	15
3.6	jae	16
3.7	jb	17
3.8	jbe	18
3.9	jge	19
3.10	jmp	20
3.11	jng	21
3.12	jnge	22
3.13	jnz	23
3.14	jz	24
3.15	loads	25
3.16	loadz	26
3.17	movb	27
3.18	movq	28
3.19	movsbq	29
3.20	movzbq	30
3.21	mulw	31

3.22	putc	32
3.23	shlq	33
3.24	shrq	34
3.25	subq	35
4	ISA Source File for the ULM Generator	36

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

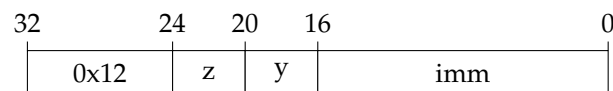
3.1.1 Assembly Notation

addq imm, %y, %z

Purpose

Adds the zero extended immediate value *imm* to register %y and stores the result in register %z.

Format



Effect

$$(u(\%y) + u(imm)) \bmod 2^{64} \rightarrow u(\%z)$$

Updates the status flags:

Flag	Condition
ZF	$u(\%y) + u(imm) \bmod 2^{64} = 0$
CF	$u(\%y) + u(imm) \geq 2^{64}$
OF	$s(\%y) + s(imm) \notin \{-2^{63}, \dots, 2^{63}\}$
SF	$s(\%y) + s(imm) < 0$

3.1.2 Assembly Notation

addq %x, %y, %z

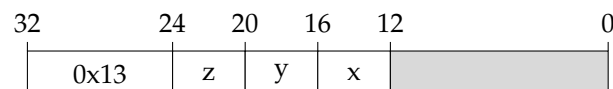
Alternative Assembly Notation

movq %x, %z

Purpose

Adds register %x to register and stores the result in register %z.

Format



Effect

$$(u(\%y) + u(\%x)) \bmod 2^{64} \rightarrow u(\%z)$$

Updates the status flags:

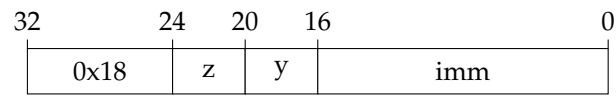
Flag	Condition
ZF	$u(\%y) + u(\%x) \bmod 2^{64} = 0$
CF	$u(\%y) + u(\%x) \geq 2^{64}$
OF	$s(\%y) + s(\%x) \notin \{-2^{63}, \dots, 2^{63}\}$
SF	$s(\%y) + s(\%x) < 0$

3.2 andq

3.2.1 Assembly Notation

andq imm, %y, %z

Format



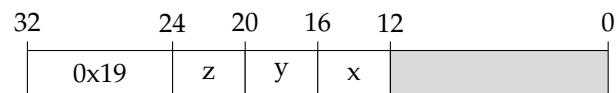
Effect

$$u(\text{imm}) \wedge_b u(\%y) \rightarrow u(\%z)$$

3.2.2 Assembly Notation

andq %x, %y, %z

Format



Effect

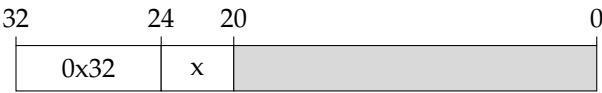
$$u(\%x) \wedge_b u(\%y) \rightarrow u(\%z)$$

3.3 getc

3.3.1 Assembly Notation

getc %x

Format



Effect

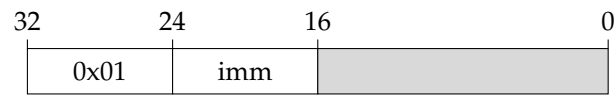
$$s(\text{ulm_readChar}()) \wedge_b 255 \bmod 2^{64} \rightarrow u(\%x)$$

3.4 halt

3.4.1 Assembly Notation

halt imm

Format



Effect

halt program execution with exit code $u(\text{imm}) \bmod 2^8$

3.4.2 Assembly Notation

halt %x

Format



Effect

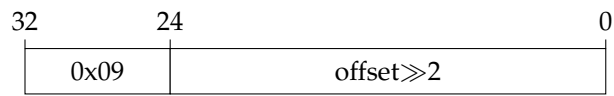
halt program execution with exit code $u(\%x) \bmod 2^8$

3.5 ja

3.5.1 Assembly Notation

ja offset

Format



Effect

If the condition

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

evaluates to true then

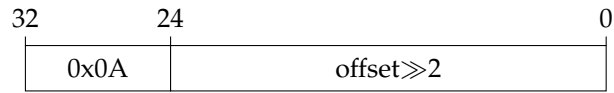
$$(u(\%IP) + s(offset)) \bmod 2^{64} \rightarrow u(\%IP)$$

3.6 jae

3.6.1 Assembly Notation

jae offset

Format



Effect

If the condition

$$CF = 0$$

evaluates to true then

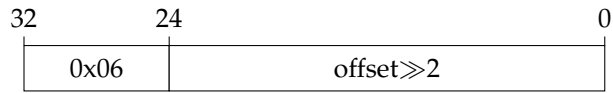
$$(u(\%IP) + s(offset)) \bmod 2^{64} \rightarrow u(\%IP)$$

3.7 jb

3.7.1 Assembly Notation

jb offset

Format



Effect

If the condition

$$CF = 1$$

evaluates to true then

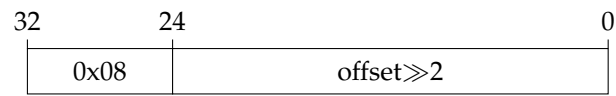
$$(u(\%IP) + s(offset)) \bmod 2^{64} \rightarrow u(\%IP)$$

3.8 jbe

3.8.1 Assembly Notation

jbe offset

Format



Effect

If the condition

$$ZF = 1 \vee CF = 1$$

evaluates to true then

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

3.9 jge

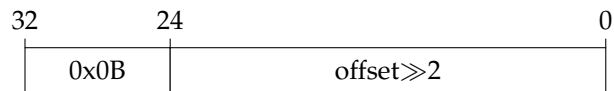
3.9.1 Assembly Notation

jge offset

Alternative Assembly Notation

jnl offset

Format



Effect

If the condition

$$SF = OF$$

evaluates to true then

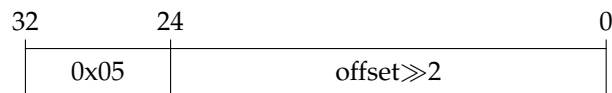
$$(u(\%IP) + s(offset)) \bmod 2^{64} \rightarrow u(\%IP)$$

3.10 jmp

3.10.1 Assembly Notation

jmp offset

Format



Effect

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

3.10.2 Assembly Notation

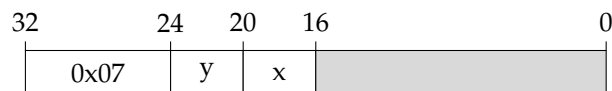
jmp %y, %x

Alternative Assembly Notation

call %y, %x

ret %y

Format



Effect

$$(u(\%IP) + 4) \bmod 2^{64} \rightarrow u(\%x)$$

$$u(\%y) \rightarrow u(\%IP)$$

3.11 jng

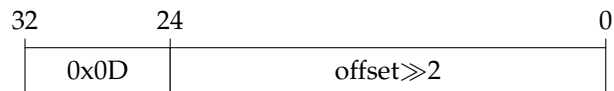
3.11.1 Assembly Notation

jng offset

Alternative Assembly Notation

jle offset

Format



Effect

If the condition

$$ZF = 1 \vee SF \neq OF$$

evaluates to true then

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

3.12 jnge

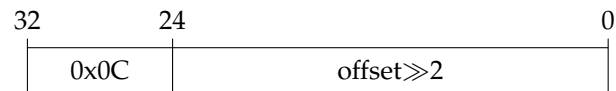
3.12.1 Assembly Notation

jnge offset

Alternative Assembly Notation

jl offset

Format



Effect

If the condition

$$SF \neq OF$$

evaluates to true then

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

3.13 jnz

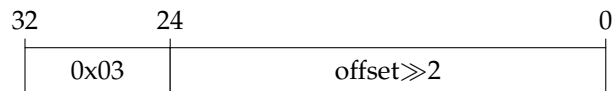
3.13.1 Assembly Notation

jnz offset

Alternative Assembly Notation

jne offset

Format



Effect

If the condition

$$ZF = 0$$

evaluates to true then

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

3.14 jz

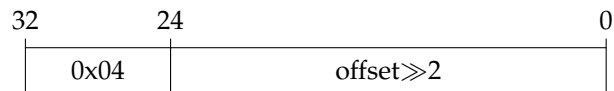
3.14.1 Assembly Notation

jz offset

Alternative Assembly Notation

je offset

Format



Effect

If the condition

$$ZF = 1$$

evaluates to true then

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

3.15 loads

3.15.1 Assembly Notation

loads imm, %dest

Purpose

Load sign extended immediate value into destination register.

Format



Effect

$$s(\text{imm}) \bmod 2^{64} \rightarrow u(\%dest)$$

3.16 loadz

3.16.1 Assembly Notation

loadz imm, %dest

Purpose

Load zero extended immediate value into destination register.

Format



Effect

$$u(\text{imm}) \bmod 2^{64} \rightarrow u(\%dest)$$

3.17 movb

3.17.1 Assembly Notation

movb %data, offset(%addr)

Alternative Assembly Notation

movb %data, (%addr)

Format

32	24	20	16	0
0x22	data	addr	offset	

Effect

$u(\%data) \bmod 2^8 \rightarrow u(M_1(addr))$ with $addr = (s(offset) + u(\%addr)) \bmod 2^{64}$

3.18 movq

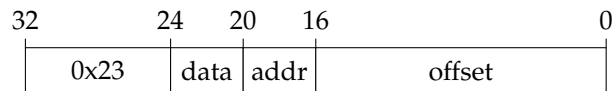
3.18.1 Assembly Notation

movq offset(%addr), %data

Alternative Assembly Notation

movq (%addr), %data

Format



Effect

$u(M_8(addr)) \rightarrow u(\%data)$ with $addr = (s(offset) + u(\%addr)) \bmod 2^{64}$

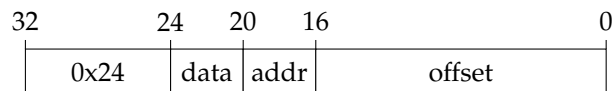
3.18.2 Assembly Notation

movq %data, offset(%addr)

Alternative Assembly Notation

movq %data, (%addr)

Format



Effect

$u(\%data) \bmod 2^{64} \rightarrow u(M_8(addr))$ with $addr = (s(offset) + u(\%addr)) \bmod 2^{64}$

3.19 movsbq

3.19.1 Assembly Notation

movsbq offset(%addr), %data

Alternative Assembly Notation

movsbq (%addr), %data

Format

32	24	20	16	0
0x21	data	addr	offset	

Effect

$s(M_1(addr)) \rightarrow u(\%data)$ with $addr = (s(offset) + u(\%addr)) \bmod 2^{64}$

3.20 movzbq

3.20.1 Assembly Notation

movzbq offset(%addr), %data

Alternative Assembly Notation

movzbq (%addr), %data

Format

32	24	20	16	0
0x20	data	addr	offset	

Effect

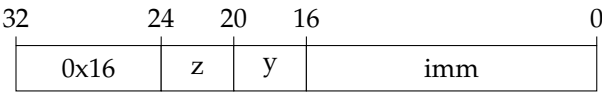
$u(M_1(addr)) \rightarrow u(\%data)$ with $addr = (s(offset) + u(\%addr)) \bmod 2^{64}$

3.21 mulw

3.21.1 Assembly Notation

mulw imm, %y, %z

Format



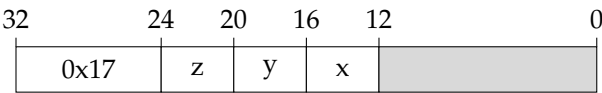
Effect

$u(imm) \cdot u(\%y) \wedge_b 65535 \bmod 2^{64} \rightarrow u(\%z)$

3.21.2 Assembly Notation

mulw %x, %y, %z

Format



Effect

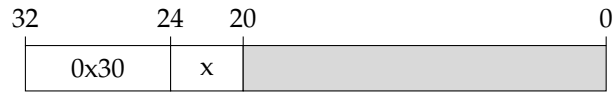
$u(\%x) \wedge_b 65535 \cdot u(\%y) \wedge_b 65535 \bmod 2^{64} \rightarrow u(\%z)$

3.22 putc

3.22.1 Assembly Notation

putc %x

Format



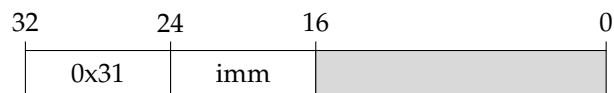
Effect

ulm_printChar(%x)

3.22.2 Assembly Notation

putc imm

Format



Effect

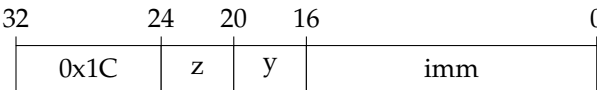
ulm_printChar(imm)

3.23 shlq

3.23.1 Assembly Notation

shlq imm, %y, %z

Format



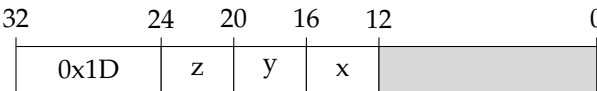
Effect

TODO: ulm_shiftLeft64

3.23.2 Assembly Notation

shlq %x, %y, %z

Format



Effect

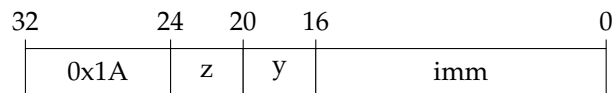
TODO: ulm_shiftLeft64

3.24 shrq

3.24.1 Assembly Notation

shrq imm, %y, %z

Format



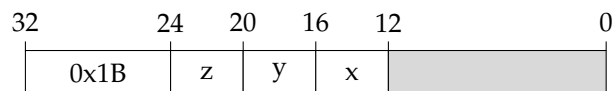
Effect

TODO: ulm_shiftRightUnsigned64

3.24.2 Assembly Notation

shrq %x, %y, %z

Format



Effect

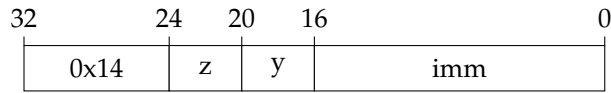
TODO: ulm_shiftRightUnsigned64

3.25 subq

3.25.1 Assembly Notation

subq imm, %y, %z

Format



Effect

$$(u(\%y) - u(imm)) \bmod 2^{64} \rightarrow u(\%z)$$

Updates the status flags:

Flag Condition

ZF $u(\%y) - u(imm) = 0$

CF $u(\%y) - u(imm) < 0$

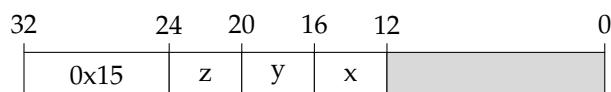
OF $s(\%y) - s(imm) \notin \{-2^{63}, \dots, 2^{63}\}$

SF $s(\%y) - s(imm) < 0$

3.25.2 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 $s(\%y) - s(\%x) \notin \{-2^{63}, \dots, 2^{63}\}$

SF $s(\%y) - s(\%x) < 0$

Chapter 4

ISA Source File for the ULM Generator

```
1  U8 (OP u 8) (imm u 8)
2  R (OP u 8) (x u 4)
3  RR (OP u 8) (y u 4) (x u 4)
4  REL_JMP (OP u 8) (offset j 24)
5  U20_R (OP u 8) (dest u 4) (imm u 20)
6  S20_R (OP u 8) (dest u 4) (imm s 20)
7  R_R_R (OP u 8) (z u 4) (y u 4) (x u 4)
8  U16_R_R (OP u 8) (z u 4) (y u 4) (imm u 16)
9  S16_R_R (OP u 8) (z u 4) (y u 4) (imm s 16)
10 MR_R (OP u 8) (data u 4) (addr u 4) (offset s 16)
11 R_MR (OP u 8) (data u 4) (addr u 4) (offset s 16)
12
13 #
14 # CU (control unit) instructions
15 #
16
17 0x01 U8
18 : halt imm
19     ulm_halt(imm);
20
21 0x02 R
22 : halt %x
23     ulm_halt(ulm_regVal(x));
24
25 0x03 REL_JMP
26 : jnz offset
27 : jne offset
28     ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 0, offset);
29
30 0x04 REL_JMP
31 : jz offset
32 : je offset
33     ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 1, offset);
34
35 0x05 REL_JMP
```

```

36 : jmp offset
37     ulm_unconditionalRelJump(offset);
38
39 0x06 REL_JMP
40 : jb offset
41     ulm_conditionalRelJump(ulm_statusReg[ULM_CF] == 1, offset);
42
43 0x07 RR
44 : jmp %y, %x
45 : call %y, %x
46 : ret %y
47     ulm_absJump(ulm_regVal(y), x);
48
49 0x08 REL_JMP
50 : jbe offset
51     ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 1
52                             || ulm_statusReg[ULM_CF] == 1, offset);
53
54 0x09 REL_JMP
55 : ja offset
56     ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 0
57                             && ulm_statusReg[ULM_CF] == 0, offset);
58
59 0x0A REL_JMP
60 : jae offset
61     ulm_conditionalRelJump(ulm_statusReg[ULM_CF] == 0, offset);
62
63 0x0B REL_JMP
64 : jge offset
65 : jnl offset
66     ulm_conditionalRelJump(ulm_statusReg[ULM_SF] == ulm_statusReg[ULM_OF],
67                             offset);
68
69 0x0C REL_JMP
70 : jnge offset
71 : jl offset
72     ulm_conditionalRelJump(ulm_statusReg[ULM_SF] != ulm_statusReg[ULM_OF],
73                             offset);
74
75 0x0D REL_JMP
76 : jng offset
77 : jle offset
78     ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 1 ||
79                             ulm_statusReg[ULM_SF] != ulm_statusReg[ULM_OF],
80                             offset);
81
82 #
83 #
84 # ALU (arithmetic logic unit)
85 #
86
87 0x10 U20_R
88 # Load zero extended immediate value into destination register.

```

```

89 : loadz imm, %dest
90     ulm_setReg(imm, dest);
91
92 0x11 S20_R
93 # Load sign extended immediate value into destination register.
94 : loads imm, %dest
95     ulm_setReg(imm, dest);
96
97 0x12 U16_R_R
98 # Adds the zero extended immediate value \\texttt{imm} to register
99 # \\texttt{%y} and stores the result in register \\texttt{%z}.
100 : addq imm, %y, %z
101     ulm_add64(imm, ulm_regVal(y), z);
102
103 0x13 R_R_R
104 # Adds register \\texttt{%x} to register and stores the result in
105 # register \\texttt{%z}.
106 : addq %x, %y, %z
107 : movq %x, %z
108     ulm_add64(ulm_regVal(x), ulm_regVal(y), z);
109
110 0x14 U16_R_R
111 : subq imm, %y, %z
112     ulm_sub64(imm, ulm_regVal(y), z);
113
114 0x15 R_R_R
115 : subq %x, %y, %z
116     ulm_sub64(ulm_regVal(x), ulm_regVal(y), z);
117
118 0x16 U16_R_R
119 : mulw imm, %y, %z
120     ulm_mul64(imm, ulm_regVal(y) & 0xFFFF, z);
121
122 0x17 R_R_R
123 : mulw %x, %y, %z
124     ulm_mul64(ulm_regVal(x) & 0xFFFF, ulm_regVal(y) & 0xFFFF, z);
125
126 0x18 U16_R_R
127 : andq imm, %y, %z
128     ulm_and64(imm, ulm_regVal(y), z);
129
130 0x19 R_R_R
131 : andq %x, %y, %z
132     ulm_and64(ulm_regVal(x), ulm_regVal(y), z);
133
134 0x1A U16_R_R
135 : shrq imm, %y, %z
136     ulm_shiftRightUnsigned64(imm, ulm_regVal(y), z);
137
138 0x1B R_R_R
139 : shrq %x, %y, %z
140     ulm_shiftRightUnsigned64(ulm_regVal(x), ulm_regVal(y), z);
141

```

```

142 0x1C U16_R_R
143 : shlq imm, %y, %z
144     ulm_shiftLeft64(imm, ulm_regVal(y), z);
145
146 0x1D R_R_R
147 : shlq %x, %y, %z
148     ulm_shiftLeft64(ulm_regVal(x), ulm_regVal(y), z);
149
150 #
151 # bus instructions
152 #
153
154 0x20 MR_R
155 : movzbq offset(%addr), %data
156 : movzbq (%addr), %data
157     ulm_fetch64(offset, addr, 0, 1, ULM_ZERO_EXT, 1, data);
158
159 0x21 MR_R
160 : movsbq offset(%addr), %data
161 : movsbq (%addr), %data
162     ulm_fetch64(offset, addr, 0, 1, ULM_SIGN_EXT, 1, data);
163
164 0x22 R_MR
165 : movb %data, offset(%addr)
166 : movb %data, (%addr)
167     ulm_store64(offset, addr, 0, 0, 1, data);
168
169 0x23 MR_R
170 : movq offset(%addr), %data
171 : movq (%addr), %data
172     ulm_fetch64(offset, addr, 0, 1, ULM_ZERO_EXT, 8, data);
173
174 0x24 R_MR
175 : movq %data, offset(%addr)
176 : movq %data, (%addr)
177     ulm_store64(offset, addr, 0, 0, 8, data);
178
179 #
180 # i/o instructions
181 #
182 0x30 R
183 : putc %x
184     ulm_printChar(ulm_regVal(x));
185
186 0x31 U8
187 : putc imm
188     ulm_printChar(imm);
189
190 0x32 R
191 : getc %x
192     ulm_setReg(ulm_readChar() & 0xFF, x);

```
