Instruction set of the ULM (Ulm Lecture Machine)

March 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

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

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 %x, %y, %z

Alternative Assembly Notation

movq %x, %z

Format

32–24	24–20	20–16	16–12	12–0
0x11	z	y	x	

Effect

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

Updates the status flags:

Flag	Condition
ZF	$u(\%y) + u(\%x) = 0$
CF	$u(\%y) + u(\%x) \geq 2^{64}$
OF	$s(\%y) + s(\%x) \notin \{-2^{63}, \ldots, 2^{63} - 1\}$
SF	$s(\%y) + s(\%x) < 0$

3.1.2 Assembly Notation

addq imm, %y, %z

Format

32–24	24–20	20–16	16–0
0x12	z	y	imm

Effect

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

Updates the status flags:

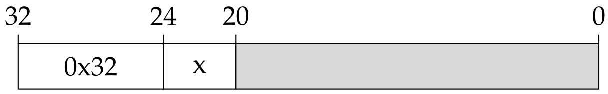
Flag	Condition
ZF	$u(\%y) + u(imm) = 0$
CF	$u(\%y) + u(imm) \geq 2^{64}$
OF	$s(\%y) + s(imm) \notin \{-2^{63}, \ldots, 2^{63} - 1\}$
SF	$s(\%y) + s(imm) < 0$

3.2 getc

3.2.1 Assembly Notation

getc %x

Format



Effect

$$s(\mathit{ulm\_readChar()}) \wedge_b 255 \bmod 2^{64} \to u\,(\%x)$$

3.3 halt

3.3.1 Assembly Notation

halt imm

Format

32–24	24–16	16–0
0x01	imm	

Effect

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

3.3.2 Assembly Notation

halt %x

Format

32–24	24–20	20–0
0x02	x	

Effect

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

3.4 jb

3.4.1 Assembly Notation

jb offset

Format

32	24	0
0x06	offset≫2	

Effect

If the condition

$$CF = 1$$

evaluates to true then

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

3.5 jmp

3.5.1 Assembly Notation

jmp offset

Format

32	24	0
0x05	offset≫2	

Effect

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

3.5.2 Assembly Notation

jmp %y, %x

Alternative Assembly Notation

call %y, %x

ret %y

Format

32	24	20	16	0
0x07	y	x		

Effect

$$\begin{array}{lcl} (u(\%IP) + 4) \bmod 2^{64} & \to & u(\%x) \\ u(\%y) & \to & u(\%IP) \end{array}$$

3.6 jnz

3.6.1 Assembly Notation

jnz offset

Alternative Assembly Notation

jne offset

Format

32	24	0
0x03	offset$\gg$2	

Effect

If the condition

$$ZF = 0$$

evaluates to true then

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

3.7 jz

3.7.1 Assembly Notation

jz offset

Alternative Assembly Notation

je offset

Format

32	24	0
0x04	offset≫2	

Effect

If the condition

$$ZF = 1$$

evaluates to true then

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

3.8 load

3.8.1 Assembly Notation

load imm, %dest

Format

32	24	20	0
0x10	dest	imm	

Effect

$$u(imm) \bmod 2^{64} \to u\,(\%dest)$$

3.9 movb

3.9.1 Assembly Notation

movb %data, offset(%addr)

Alternative Assembly Notation

movb %data, (%addr)

Format

32	24	20	16	0
0x22	data	addr	offset	

Effect

$$u(\%data) \bmod 2^8 \to u\left(\mathrm{M}_1\left(addr\right)\right) \text{ with } addr = (s(offset) + u(\%addr)) \bmod 2^{64}$$

3.10 movq

3.10.1 Assembly Notation

movq offset(%addr), %data

Alternative Assembly Notation

movq (%addr), %data

Format

32–24	24–20	20–16	16–0
0x23	data	addr	offset

Effect

$$u\left(\mathrm{M}_8\left(addr\right)\right) \to u\left(\%data\right) \text{ with } addr = \left(s(offset) + u(\%addr)\right) \bmod 2^{64}$$

3.10.2 Assembly Notation

movq %data, offset(%addr)

Alternative Assembly Notation

movq %data, (%addr)

Format

32–24	24–20	20–16	16–0
0x24	data	addr	offset

Effect

$$u(\%data) \bmod 2^{64} \to u\left(\mathrm{M}_8\left(addr\right)\right) \text{ with } addr = \left(s(offset) + u(\%addr)\right) \bmod 2^{64}$$

3.11 movsbq

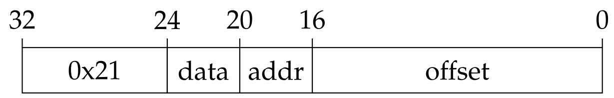
3.11.1 Assembly Notation

movsbq offset(%addr), %data

Alternative Assembly Notation

movsbq (%addr), %data

Format



Effect

$$s\left(\mathrm{M}_1\left(addr\right)\right) \to u\left(\%data\right) \text{ with } addr = \left(s(offset) + u(\%addr)\right) \bmod 2^{64}$$

3.12 movzbq

3.12.1 Assembly Notation

movzbq offset(%addr), %data

Alternative Assembly Notation

movzbq (%addr), %data

Format

32	24	20	16 … 0
0x20	data	addr	offset

Effect

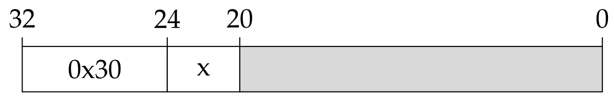
$$u\left(\mathrm{M}_1\left(addr\right)\right) \to u\left(\%data\right) \text{ with } addr = \left(s(offset) + u(\%addr)\right) \bmod 2^{64}$$

3.13 putc

3.13.1 Assembly Notation

putc %x

Format



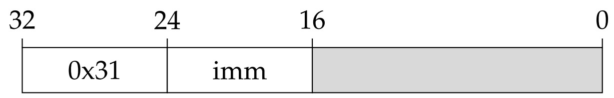
Effect

ulm_printChar(%*x*)

3.13.2 Assembly Notation

putc imm

Format



Effect

ulm_printChar(*imm*)

3.14 subq

3.14.1 Assembly Notation

subq %x, %y, %z

Format

32–24	24–20	20–16	16–12	12–0
0x13	z	y	x	

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	$s(\%y) - s(\%x) \notin \{-2^{63}, \ldots, 2^{63} - 1\}$
SF	$s(\%y) - s(\%x) < 0$

3.14.2 Assembly Notation

subq imm, %y, %z

Format

32–24	24–20	20–16	16–0
0x14	z	y	imm

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	$s(\%y) - s(imm) \notin \{-2^{63}, \ldots, 2^{63} - 1\}$
SF	$s(\%y) - s(imm) < 0$

Chapter 4

ISA Source File for the ULM Generator

```
U8 (OP u 8) (imm u 8)
R (OP u 8) (x u 4)
RR (OP u 8) (y u 4) (x u 4)
REL_JMP (OP u 8) (offset j 24)
U20_R (OP u 8) (dest u 4) (imm u 20)
R_R_R (OP u 8) (z u 4) (y u 4) (x u 4)
U16_R_R (OP u 8) (z u 4) (y u 4) (imm u 16)
MR_R (OP u 8) (data u 4) (addr u 4) (offset s 16)
R_MR (OP u 8) (data u 4) (addr u 4) (offset s 16)

#
# CU (control unit) instructions
#

0x01 U8
: halt imm
    ulm_halt(imm);

0x02 R
: halt %x
    ulm_halt(ulm_regVal(x));

0x03 REL_JMP
: jnz offset
: jne offset
    ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 0, offset);

0x04 REL_JMP
: jz offset
: je offset
    ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 1, offset);

0x05 REL_JMP
: jmp offset
    ulm_unconditionalRelJump(offset);
```

```

0x06 REL_JMP
: jb offset
      ulm_conditionalRelJump(ulm_statusReg[ULM_CF] == 1, offset);

0x07 RR
: jmp %y, %x
: call %y, %x
: ret %y
      ulm_absJump(ulm_regVal(y), x);


#
# ALU (arithmetic logic unit)
#

0x10 U20_R
: load imm, %dest
      ulm_setReg(imm, dest);

0x11 R_R_R
: addq %x, %y, %z
: movq %x, %z
      ulm_add64(ulm_regVal(x), ulm_regVal(y), z);

0x12 U16_R_R
: addq imm, %y, %z
      ulm_add64(imm, ulm_regVal(y), z);

0x13 R_R_R
: subq %x, %y, %z
      ulm_sub64(ulm_regVal(x), ulm_regVal(y), z);

0x14 U16_R_R
: subq imm, %y, %z
      ulm_sub64(imm, ulm_regVal(y), z);

#
# bus instructions
#

0x20 MR_R
: movzbq offset(%addr), %data
: movzbq (%addr), %data
      ulm_fetch64(offset, addr, 0, 1, ULM_ZERO_EXT, 1, data);

0x21 MR_R
: movsbq offset(%addr), %data
: movsbq (%addr), %data
      ulm_fetch64(offset, addr, 0, 1, ULM_SIGN_EXT, 1, data);

0x22 R_MR
: movb %data, offset(%addr)
```

```
: movb %data, (%addr)
      ulm_store64(offset, addr, 0, 0, 1, data);

0x23 MR_R
: movq offset(%addr), %data
: movq (%addr), %data
      ulm_fetch64(offset, addr, 0, 1, ULM_ZERO_EXT, 8, data);

0x24 R_MR
: movq %data, offset(%addr)
: movq %data, (%addr)
      ulm_store64(offset, addr, 0, 0, 8, data);

#
# i/o instructions
#
0x30 R
: putc %x
      ulm_printChar(ulm_regVal(x));

0x31 U8
: putc imm
      ulm_printChar(imm);

0x32 R
: getc %x
      ulm_setReg(ulm_readChar() & 0xFF, x);
```