Instruction set of the ULM (Ulm Lecture Machine)

May 26, 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

Integer addition.

3.1.1 Assembly Notation

addq X, %Y, %Z

Format

32 24 16 8 0

0x0A	X	Y	Z

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

3.1.2 Assembly Notation

addq %X, %Y, %Z

Format

32 24 16 8 0

0x0E	X	Y	Z

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

32	24	16	8	0
0x10	X	Y	Z	

Effect

For the unsigned 128-bit numerator

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

and unsigned 64-bia denominator

$$a := u(X)$$

computes the divisor

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

and the remainder

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

3.3 getc

Get character

3.3.1 Assembly Notation

getc %X

Format

32	24	16	8	0
0x02	X	Y	Z	

Effect

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

3.4 halt

Halt program

3.4.1 Assembly Notation

halt %X

Format

32	24	16	8	0
0x01	X	Y	Z	

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

32 24 16 8 0

0x0B	X	Y	Z

Effect

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

3.5.2 Assembly Notation

imulq X, %Y, %Z

Format

32 24 16 8 0

0x0F	X	Y	Z

Effect

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

3.6 ja

Jump if above (conditional jump).

3.6.1 Assembly Notation

ja XYZ

Format

32–24	24–0
0x0C	XYZ≫2

Effect

If the condition

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

evaluates to true then

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

3.7 jb

Jump if below (conditional jump).

3.7.1 Assembly Notation

jb XYZ

Format

32	24 0
0x0D	XYZ≫2

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

32	24	0
0x04	XYZ$\gg$2	

Effect

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

3.9 jnz

Jump if not zero (conditional jump).

3.9.1 Assembly Notation

jnz XYZ

Alternative Assembly Notation

jne XYZ

Format

32 24 0

0x06	XYZ$\gg$2

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

32 – 24	24 – 0
0x07	XYZ≫2

Effect

If the condition

$$ZF = 1$$

evaluates to true then

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

3.11 ldzwq

Load zero extended word to quad word register.

3.11.1 Assembly Notation

ldzwq XY, %Z

Format

32–24	24–8	8–0
0x08	XY	Z

Effect

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

3.12 movb

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

3.12.1 Assembly Notation

movb %X, (%Z)

Purpose

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

Format

32–24	24–16	16–8	8–0
0x11	X	Y	Z

Effect

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

3.13 movq

Move quad word

3.13.1 Assembly Notation

movq (%X), %Z

Purpose

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

Format

32–24	24–16	16–8	8–0
0x12	X	Y	Z

Effect

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

3.13.2 Assembly Notation

movq %X, (%Z)

Format

32–24	24–16	16–8	8–0
0x13	X	Y	Z

Effect

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

3.14 movzbq

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

3.14.1 Assembly Notation

movzbq (%X), %Z

Format

32 24	24 16	16 8	8 0
0x09	X	Y	Z

Effect

$$u\left(\mathrm{M}_1\left(addr\right)\right) \to u\left(\%Z\right) \;\text{ with } addr = u(\%X) \bmod 2^{64}$$

3.15 putc

Put character

3.15.1 Assembly Notation

putc %X

Format

32–24	24–16	16–8	8–0
0x03	X	Y	Z

Effect

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

3.16 subq

Integer subtraction.

3.16.1 Assembly Notation

subq X, %Y, %Z

Format

32–24	24–16	16–8	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

```
RRR (OP u 8) (X u 8) (Y u 8) (Z u 8)
J26 (OP u 8) (XYZ j 24)
U16R (OP u 8) (XY u 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
    ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 1, XYZ);
```

```
0x08 U16R
: ldzwq XY, %Z
    ulm_setReg(XY, Z);

0x09 RRR
: movzbq (%X), %Z
    ulm_fetch64(0, X, 0, 0, ULM_ZERO_EXT, 1, Z);

0x0A RRR
: addq X, %Y, %Z
    ulm_add64(X, ulm_regVal(Y), Z);

0x0B RRR
: imulq %X, %Y, %Z
    ulm_mul64(ulm_regVal(X), ulm_regVal(Y), Z);

0x0C J26
: ja XYZ
    ulm_conditionalRelJump(ulm_statusReg[ULM_ZF] == 0
                           && ulm_statusReg[ULM_CF] == 0, XYZ);

0x0D J26
: jb XYZ
    ulm_conditionalRelJump(ulm_statusReg[ULM_CF] == 1, XYZ);

0x0E RRR
: addq %X, %Y, %Z
    ulm_add64(ulm_regVal(X), ulm_regVal(Y), Z);

0x0F RRR
: imulq X, %Y, %Z
    ulm_mul64(X, ulm_regVal(Y), Z);

0x10 RRR
# 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}.
: divq X, %Y, %Z
    ulm_div128(X, ulm_regVal(Y), ulm_regVal(0), Z, 0, Z + 1);

0x11 RRR
# The least significant byte in %X gets stored at address %Z.
: movb %X, (%Z)
    ulm_store64(0, Z, 0, 0, 1, X);

0x12 RRR
# Fetches a quad word from address %X into register %Z
: movq (%X), %Z
    ulm_fetch64(0, X, 0, 0, ULM_ZERO_EXT, 8, Z);
```

```

0x13 RRR
: movq %X, (%Z)
    ulm_store64(0, Z, 0, 0, 8, X);

# General meaning of the mnemonics

@addq
# Integer addition.
@getc
# Get character
@divq
# 64-bit unsigned integer division
@halt
# Halt program
@imulq
# 64-bit unsigned and signed integer multiplication.
@ja
# Jump if above (conditional jump).
@jb
# Jump if below (conditional jump).
@je
# Jump if equal (conditional jump).
@jmp
# Jump (unconditional jump).
@jne
# Jump if not equal (conditional jump).
@jnz
# Jump if not zero (conditional jump).
@jz
# Jump if zero (conditional jump).
@ldzwq
# Load zero extended word to quad word register.
@putc
# Put character
@subq
# Integer subtraction.
@movb
# Move a byte from a register to memory (store instruction).
@movzbq
# Move zero extended byte to quad word register (fetch instruction).
@movq
# Move quad word
```