MiniASM Specification

Version: 0.0.1

Contents

1 Purpose **2**

2 Architecture **2**

2.1 Instruction Types 2

2.1.1 Two-Register (R2) Type 2

2.1.2 One-Register (R1) Type 2

2.1.3 Zero-Register (R0) Type 2

2.1.4 Immediate (I) Type 3

2.1.5 Jump (J) Type 3

2.2 Special-Purpose Registers 3

2.2.1 Program Counter 3

2.2.2 Stack Pointer 3

2.2.3 Status Register 4

3 Instruction Listing **4**

4 Program Execution **5**

4.1 Program Termination 6

4.2 Error Conditions 6

5 Sample Programs **6**

5.1 Factorial 6

5.2 String reversal 7

6 Acknowledgements **8**

1 Purpose

This document serves as a specification for a hypothetical instruction set, designed specifically for teaching low-level programming concepts.

2 Architecture

We assume a big-endian Von Neumann architecture wherein instructions are stored contiguously in memory starting at address 128. The word size of this hypothetical architecture is 16 bits. There are 32 1-word registers – $R_0, R_1, \ldots, R_{31}$ – of which the first 25 are general-purpose. Additionally, there is exactly 1024 bytes of contiguous, byte-addressable memory, starting from address zero (which will henceforth be referred to as "Mem" and treated as a 0-based array of bytes).

There is also an output device for displaying textual output (to which data written to the *output stream* is sent), in addition to an input device for reading textual input (from which data read from the *input stream* comes).

Finally, single instructions are always represented with single words. There are several different instruction formats, each pertaining to a specific instruction type.

2.1 Instruction Types

2.1.1 Two-Register (R2) Type

6 bits	**5 bits**	**5 bits**
Opcode	R_D	R_S

An example R2-type instruction is `add R2 R3`, which adds the contents of R_2 and R_3, then stores the result back into R_2. Another example is `mov R5 R4`, which copies the contents of R_4 into R_5.

2.1.2 One-Register (R1) Type

6 bits	**5 bits**	**5 bits**
Opcode	R_D	?

An example R1-type instruction is `not R6`, which performs a bitwise-NOT operation on the contents of R_6, then puts the result back into R_6.

2.1.3 Zero-Register (R0) Type

6 bits	**10 bits**
Opcode	?

An example R0-type instruction is `halt`, which terminates program execution.

2.1.4 Immediate (I) Type

6 bits	**5 bits**	**5 bits**
Opcode	R_D	I

An example I-type instruction is `movi R7 42`, which places the decimal value 42 into R_7.

2.1.5 Jump (J) Type

6 bits	**10 bits**
Opcode	J

An example J-type instruction is `jmp -4`, which jumps backwards 3 instructions relative to the current instruction.

2.2 Special-Purpose Registers

Registers $R_{26}, R_{37}, \ldots, R_{32}$ are specific-purpose registers.

Register	**Alt. Name**	**Purpose**
R_{26}	R_{PC}	Program counter
R_{27}	R_{SP}	Stack pointer
R_{28}	R_S	Status register
R_{29}	-	Currently unused
R_{30}	-	Currently unused
R_{31}	-	Currently unused

It is important to note that although R_{29}, R_{30} and R_{31} are currently unused, they could be assigned a meaning in future updates to this specification. Hence, programs still should not use these registers as general-purpose registers.

2.2.1 Program Counter

R_{26} stores the program counter, which always points to the *next* instruction to be executed. The program counter is always incremented when an instruction is fetched. Jump instructions modify this register.

2.2.2 Stack Pointer

The stack grows downwards from the highest addressable memory location. R_{27} stores the stack pointer, which always points to the element after the top of the stack. "Element" in this context means "word", because the stack uses words as its fundamental unit of transaction.

2.2.3 Status Register

R_{28} is the status register, which has the following layout:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U	U	U	U	U	U	U	U	U	U	U	U	U	U	S	Z

where

- Z is the *zero bit*, which is 1 if and only if the result of the last arithmetic instruction was zero.
- S is the *sign bit*, which is 1 if and only if the result of the last arithmetic instruction was negative.
- U indicates that the specified bit is currently unused.

3 Instruction Listing

Instruction	Opcode	Type	Effect
`halt`	000000	R0	Terminate program
`break`	111111	R0	Breakpoint
`not`	000001	R1	$R_D \leftarrow \texttt{~}\, R_D$
`push`	000010	R1	$\text{Mem}[R_{SP}] \leftarrow \text{Hi}(R_D)$, $\text{Mem}[R_{SP}+1] \leftarrow \text{Lo}(R_D)$, $R_{SP} \leftarrow R_{SP} - 2$
`pop`	000011	R1	$R_{SP} \leftarrow R_{SP} + 2$, $\text{Hi}(R_D) \leftarrow \text{Mem}[R_{SP}]$ $\text{Lo}(R_D) \leftarrow \text{Mem}[R_{SP}+1]$
`print`	000100	R1	See below
`read`	000101	R1	See below
`sl`	000110	R2	$R_D \leftarrow R_D \texttt{ << } R_S$
`sru`	000111	R2	$R_D \leftarrow R_D \texttt{ >> } R_S$
`srs`	001000	R2	$R_D \leftarrow R_D \texttt{ >>> } R_S$
`mov`	001001	R2	$R_D \leftarrow R_S$
`add`	001010	R2	$R_D \leftarrow R_D + R_S$
`sub`	001011	R2	$R_D \leftarrow R_D - R_S$
`and`	001100	R2	$R_D \leftarrow R_D \texttt{ \& } R_S$
`or`	001101	R2	$R_D \leftarrow R_D \texttt{ \| } R_S$
`xor`	001110	R2	$R_D \leftarrow R_D \texttt{ \^{} } R_S$
`cmp`	001111	R2	$Z \leftarrow R_D \stackrel{?}{=} R_S$, $S \leftarrow R_D \stackrel{?}{<} R_S$
`sw`	010000	R2	$\text{Mem}[R_D] \leftarrow \text{Hi}(R_S)$

			$\mathrm{Mem}[R_D + 1] \leftarrow \mathrm{Lo}(R_S)$
`lw`	010001	R2	$\mathrm{Hi}(R_D) \leftarrow \mathrm{Mem}[R_S]$
			$\mathrm{Lo}(R_D) \leftarrow \mathrm{Mem}[R_S + 1]$
`sb`	010010	R2	$\mathrm{Mem}[R_D] \leftarrow \mathrm{Lo}(R_S)$
`lb`	010011	R2	$R_D \leftarrow \mathrm{Mem}[R_S]$
`movi`	010100	I	$R_D \leftarrow I$
`addi`	010101	I	$R_D \leftarrow R_D + I$
`subi`	010110	I	$R_D \leftarrow R_D - I$
`andi`	010111	I	$R_D \leftarrow R_D \mathbin{\&} I$
`ori`	011000	I	$R_D \leftarrow R_D \mid I$
`xori`	011001	I	$R_D \leftarrow R_D\ \hat{}\ I$
`jmp`	011010	J	$R_{PC} \leftarrow R_{PC} + J$
`jmpeq`	011011	J	`if` $Z = 1$ `then:`
			$R_{PC} \leftarrow R_{PC} + J$
`jmpne`	011100	J	`if` $Z = 0$ `then:`
			$R_{PC} \leftarrow R_{PC} + J$
`jmpgt`	011101	J	`if` $S = 1$ `and` $Z = 0$ `then:`
			$R_{PC} \leftarrow R_{PC} + J$
`jmplt`	011110	J	`if` $S = 1$ `then:`
			$R_{PC} \leftarrow R_{PC} + J$
`jmpge`	011111	J	`if` $S = 0$ `then:`
			$R_{PC} \leftarrow R_{PC} + J$
`jmple`	100000	J	`if` $S = 1$ `or` $Z = 0$ `then:`
			$R_{PC} \leftarrow R_{PC} + J$

The `print` instruction prints the ASCII character located at $\mathrm{Mem}[R_d]$.

The `read` instruction performs a non-blocking 1-byte read from the input stream and places the result into $\mathrm{Mem}[R_d]$. If there is nothing to be read, a 0-byte is placed in $\mathrm{Mem}[R_d]$

Note that overflow in addition or subtraction is defined behavior, and simply results in a wrap-around.

4 Program Execution

After the program code is loaded at memory address 128, program execution commences as follows:

1. $R_{PC} \leftarrow 128$
2. Fetch instruction $I = \mathrm{Mem}[R_{PC}]$.
3. $R_{PC} \leftarrow R_{PC} + 2$

4. Execute I.

5. Return to step 2, unless program terminated from execution of `halt` or from error.

4.1 Program Termination

Program termination is always accompanied by a *status*, which is either "success" or "failure". A "success" status is obtained upon the execution of a `halt` instruction, and a "failure" status is obtained when an error condition is encountered.

4.2 Error Conditions

Encountering an error condition results in immediate program termination with a "failure" status. The following error conditions exist:

- Attempting to execute an unknown opcode.
- Attempting to access a nonexistent memory location. This includes jumping to a nonexistent address.
- Attempting to execute a `sw` or `lw` instruction with a memory address that is not word-aligned (i.e. divisible by 2).

5 Sample Programs

5.1 Factorial

The following program finds the factorial of the value in register R_1 and places the result back into R_1.

```
# Factorial program
# Compiled: 500050275061244350803
    c406c06288358416bf624443c616c0454616bea24220000

MOVI  R0 0  # constant 0
MOVI  R1 7  # number to factorial-ize

MOVI  R3 1  # counter to target
MOV   R2 R3 # holds running factorial

fact:
MOVI  R4 0  # result of this multiplication

mul:  # mul R2 R3; destroy R2
CMP   R2 R0
JMPEQ out_mul
ADD   R4 R3
SUBI  R2 1
```

```
JMP    mul
out_mul:
MOV    R2 R4

CMP    R3 R1
JMPEQ  out_fact
ADDI   R3 1
JMP    fact

out_fact:
mov    R1 R2

HALT
```

5.2 String reversal

The following program reads the standard input and prints the reverse to the standard output. For example, “hello world” would become “dlrow olleh”.

```
# String reversal program
# Compiled: 5000502014204
    c413c406c0454216bf410203c206c0458216bf60000

MOVI   R0 0  # constant 0
MOVI   R1 0  # pointer into memory

# read stdin
loop1:
READ   R1
LB     R2 R1
CMP    R2 R0
JMPEQ  out1
ADDI   R1 1
JMP    loop1
out1:

# write reverse to stdout
loop2:
PRINT  R1
CMP    R1 R0
JMPEQ  out2
SUBI   R1 1
JMP    loop2
out2:

HALT
```

6 Acknowledgements

This concept is based on work done by Christopher Woodall. See `https://github.com/cwoodall/sx86-emulator`.