2OPM assembly instruction list

v0.1.1

October 22, 2014

2OPM is an assembly language intended for immediate execution on x86-64 CPUs. It is not interpreted but rather translated directly into native machine code.

1 Registers

2OPM Assembly uses the following general-purpose registers:

Name	Purpose
`$v0`	Return **V**alue
`$a0`, `$a1`, `$a2`, `$a3`, `$a4`, `$a5`	**A**rguments
`$s0`, `$s1`, `$s2`, `$s3`	**S**aved registers
`$t0`, `$t1`	**T**emporary registers
`$sp`	**S**tack **P**ointer
`$fp`	**F**rame **P**ointer
`$gp`	**G**lobal **P**ointer

Furthermore, it uses the special-purpose `$pc` (*program counter* register to indicate the address of the next instruction to execute. Regular instructions cannot access this register directly, though jump and branch operations can modify it.

When loading an assembly program, the loader ensures that the following registers are set to reasonable addresses before program start:

- `$sp` is a viable stack address, and initially `$sp` modulo 16 is 0 (as after a subroutine call).
- `$gp` points to a special static memory region for the program.

Furthermore, when entering an assembly program, the loader places a viable return address on the stack, so that assembly programs can terminate with `jreturn`.

2 Memory

The loader provides two special memory segments to the loaded assembly program:

- A custom code region (used implicitly by the program counter)
- A custom static memory region (referenced by `$gp`).

The assembly program re-uses the loader's stack. At present, direct heap access via assembly is not intended.

3 Instructions

Assembly programs consist of sequences of assembly instructions. The instructions are listed below, along with brief explanations. Each register may take a number of arguments. We distinguish between the following kinds of arguments:

- `addr`: A memory address (usually passed in by a label)
- `u8`: An 8-bit unsigned number
- `u32`: A 32-bit unsigned number
- `s32`: A 32-bit signed number
- `u64`: A 64-bit unsigned number
- `$r0`, `$r1`, `$r2`: Any general-purpose register

move	$r0, $r1	r_0$:= r_1$
li	$r0, s64	r_0$:= s64
add	$r0, $r1	r_0$:= r_0$ + r_1$
addi	$r0, u32	r_0$:= r_0$ + u32
sub	$r0, $r1	r_0$:= r_0$ - r_1$
subi	$r0, u32	r_0$:= r_0$ − u32
mul	$r0, $r1	r_0$:= r_0$ * r_1$
div_a2v0	$r0	$v0 := $a2:$v0 / r_0$, $a2 := remainder
not	$r0, $r1	if r_1$ = 0 then r_1$:= 1 else r_1$:= 0
and	$r0, $r1	r_0$:= r_0$ bitwise-and r_1$
andi	$r0, u32	r_0$:= r_0$ bitwise-and u32
or	$r0, $r1	r_0$:= r_0$ bitwise-or r_1$
ori	$r0, u32	r_0$:= r_0$ bitwise-or u32
xor	$r0, $r1	r_0$:= r_0$ bitwise-exclusive-or r_1$
xori	$r0, u32	r_0$:= r_0$ bitwise-exclusive-or u32
sll	$r0, $r1	r_0$:= r_0$ << r_1$[0:7]
slli	$r0, $r1, u8	r_0$:= r_0$ bit-shifted left by u8
srl	$r0, $r1	r_0$:= r_0$ >> r_1$[0:7]
srli	$r0, $r1, u8	r_0$:= r_0$ bit-shifted right by u8
sra	$r0, $r1	r_0$:= r_0$ >> r_1$[0:7], sign-extended
srai	$r0, u8	r_0$:= r_0$ bit-shifted right by u8, sign extension
slt	$r0, $r1, $r2	if r_1$ ¡ r_2$ then r_1$:= 1 else r_1$:= 0
sle	$r0, $r1, $r2	if r_1$ ≤ r_2$ then r_1$:= 1 else r_1$:= 0
seq	$r0, $r1, $r2	if r_1$ = r_2$ then r_1$:= 1 else r_1$:= 0
sne	$r0, $r1, $r2	if r_1$ ≠ r_2$ then r_1$:= 1 else r_1$:= 0
bgt	$r0, $r1, addr	if r_0$ > r_1$, then jump to addr
bge	$r0, $r1, addr	if r_0$ ≥ r_1$, then jump to addr
blt	$r0, $r1, addr	if r_0$ < r_1$, then jump to addr
ble	$r0, $r1, addr	if r_0$ ≤ r_1$, then jump to addr
beq	$r0, $r1, addr	if r_0$ = r_1$, then jump to addr
bne	$r0, $r1, addr	if r_0$ ≠ r_1$, then jump to addr
bgtz	$r0, addr	if r_0$ > 0, then jump to addr
bgez	$r0, addr	if r_0$ ≥ 0, then jump to addr
bltz	$r0, addr	if r_0$ < 0, then jump to addr
blez	$r0, addr	if r_0$ ≤ 0, then jump to addr
bnez	$r0, addr	if r_0$ ≠ 0, then jump to addr
beqz	$r0, addr	if r_0$ = 0, then jump to addr
j	addr	jump to addr
jr	$r0	jump to $r0
jal	addr	push next instruction address, jump to addr
jalr	$r0	push next instruction address, jump to r_0$
jreturn		jump to mem64[$sp]; $sp := $sp + 8
sb	$r0, s32, $r1	mem8[r_1$ + s32] := r_0$[7:0]
lb	$r0, s32, $r1	mem8[r_1$ + s32] := r_0$[7:0]
sd	$r0, s32, $r1	mem64[r_1$ + s32] := r_0$
ld	$r0, s32, $r1	r_0$:= mem64[r_1$ + s32]
syscall		system call
push	$r0	$sp := $sp - 8; mem64[$sp] = r_0$
pop	$r0	r_0$ = mem64[$sp]; $sp := $sp + 8

4 Calling conventions

2OPM follows the x86-64/Linux ABI (Application Binary Interface), translated to 2OPM's register names.

4.1 Preparations before subroutine call

- First six parameters in `$a0`...`$a5`
- Additional parameters in memory:
  - Argument 6 (7th): in memory at `$sp`
  - Argument 7 (8th): in memory at `$sp` + 8
    ...
- `$sp` + 8 is 128 *bit aligned* (divisible by 16)

4.2 When entering a subroutine

- `$sp` is 128 *bit aligned*
- Memory at `$sp` contains return address

The `jal` instruction ensures these properties implicitly if the assembly program makes the correct assumptions prior to the subroutine call.

4.3 During subroutine execution

- `$fp` + 8 is 128 *bit aligned*
- Function has *stack frame*:
  - Argument 7 (8th): at `$fp` + 24 (etc.)
  - Argument 6 (7th): at `$fp` + 16
  - Return address at `$fp` + 8
  - Caller's `$fp` at `$fp`
  - Local variables: start at `$fp` - 8

The ABI permits not storing `$fp` as an optional optimisation, where feasible. In that case, the otherwise alter program behaviour. In that case, local variables start directly at the memory address indicated by `$fp`.

4.4 After return from a subroutine

- The following *callee-saved registers* have the same contents as before the call:
  - `$sp`
  - `$fp`
  - `$gp`
  - `$s0`–`$s3`
- All other registers *may be modified*
- Register `$v0` contains the return value, if any

5 Command-line Assembler Tool

The 2OPM command line assembler (`asm`) can load, link, and run assembly files (using the suffix `.s`, by convention). It also includes a debugger that can step through code, print out registers, stack contents, and static memory, and execute until it hits a breakpoint[1]

5.1 Installing the Assembler

Download the assembler from the specified location. If you unpack it on a UNIX command line and run '`make`', it should compile and link a program '`asm`'. This program is a stand-alone executable and can be run from any location.

5.2 Using the Assembler

To start the assembler, write a small assembly program, store it in the file `myprogram.s` in the same directory that contains your `asm` executable, and run

```
./asm myprogram.s
```

on the command shell in that directory (see below for some sample programs).

5.2.1 Using the Debugger

To activate the debugger, start the assembler with the command line option `-d`. The debugger has a built-in help facility that can be activated by writing `help` as soon as the debugger command prompt appears.

[1] Only one breakpoint is supported at present.

5.3 Assembler Programs

The assembler takes four kinds of input:

- *assembly instructions*,
- *labels*,
- *directives*, which control the meaning of subsequent input, and
- *data*.

A functional program must provide at least the first three; providing data is optional.

As an example, consider this program:

```
.text
main:
        push $t0        ; align stack for subroutine call
        li   $a0, 42
        jal  print_int  ; call built-in function to print
        pop  $t0
        jreturn
```

(Note the comment syntax.)

This program consists of five assembly instructions, one of which calls a built-in function (see below). The first two lines, however, are not assembly instructions. Here, `.text` indicates that the following output should go into the text segment. The assembler will permit assembly instructions if and only if the text segment has been selected. The label `main` marks the main entry point. Each executable assembly program MUST define a `main` entry point. Any further labels are optional.

5.4 Directives

2OPM supports five directives. The two most important ones are `.text` and `.data`.

`.text` indicates that any following information goes into the text segment, i.e., is intended for execution. The following information must be assembly instructions and may include label definitions and label references (for suitable instructions).

`.data` indicates that any following information is pure data. No assembly instructions are permitted (this is for simplicity; in principle, the computer could represent assembly instructions in static memory). The data section permits label definitions, and freely mixes all forms of data; however, introducing data requires selecting a *data mode*.

5.5 Data modes

The following data modes are permitted:

`.byte` allows inserting single bytes, separated by commas.

`.word` allows inserting 64-bit words, separated by commas. This section also permits label references: the labels' memory address are included verbatim.

`.asciiz` allows ASCII character strings. All strings are automatically zero-terminated.

As an example for using the data segment, consider the following:

```
.text
main:
        push $fp      ; align stack
        move $fp, $sp

        la   $a0, hello
        jal  print_string  ; print out
        ld   $a0, number($gp)
        jal  print_int
        la   $t0, more_numbers
        ld   $v0, 0($t0)
        ld   $a0, 8($t0)
        add  $a0, $v0
        jal  print_int     ; print sum

        pop  $fp
        jreturn
.data
hello:
.asciiz "Hello, World!"
.word
number:
    23
more_numbers:
    3,4
```

5.6 Labels

Labels are defined by writing the label's name, followed by a colon, as in `label:`. References to labels are written by omitting the colon. Each label may be defined only once, but may be referenced arbitrarily many times.

Label references are permitted in `.data` sections in `.word` mode, and in assembly instructions such as `j` or `blt`. The assembler automatically figures out whether the references are relative or absolute and relocates suitably.

To load a memory address of any label directly, the assembler provides a pseudo-instruction:

```
la $r, addr
```

This instruction loads the absolute memory address of the specified label into register `$r`, no matter whether the address is in text or (static) data memory.

5.6.1 Built-in Operations

The 2OPM runtime comes with a small number of built-in subroutines to facilitate input and output. Each of them can be called using `jal`:

`print_int`	Print parameter as signed integer
`print_string`	Print parameter as null-terminated ascii string
`read_int`	Read and return a single 64 bit integer
`exit`	Stop the program

These functions use system calls (cf. the `syscall` operation) to achieve special effects that require interaction with the operating system; they are abstracted for convenience.

For example, the following program will read two numbers, add them, and print the resultant output:

```
.text
main:
        push $fp
        move $fp, $sp ; align stack
        jal  read_int
        move $s0, $v0
        jal  read_int
        move $a0, $s0
        add  $a0, $v0
        jal  print_int
        pop  $fp
        jreturn
```

6 Implementation Notes

Most 2OPM instructions correspond directly to x86-64 instructions. However, some of the instruction choices are not optimal: for example, 2OPM always uses 64 bit load operations, even if the number loaded can be represented in 32 or fewer bits. Some other operations are emulated: x86-64 only permits bit shifting by register `cl` (`$a2`[7:0]), so the 2OPM implementation of this instruction introduces additional register-swap operations, if needed. This may result in less-than-optimal performance.