osX Assembly (asm)

Written by Curtis Ray Welborn, Ph.D., Utah Valley University

If you find a bug in the osX Assembler please contact Dr. Welborn (Curtis.Welborn@uvu.edu) to have it fixed. If you have questions, please come and talk to me— I would love to help you. If you don't like the error messages, formatting of instructs, or just want additional features write your own assembler. You're a UVU Graduate student in the MCS if you are using this program and as such you should be able to duplicate this entire program in a matter of days as the original version was written in one weekend. The documentation I've provided took much longer to produce than the code itself.

Note on Language usage: The osX Assember was written using C. No complex data structures or advanced libraries were needed. I find writing an assembler and vm in C to be the simplest solution (students who have tried this agree with me). **This doesn't mean you should write your programs in C**, you should use the language you feel most comfortable using and the language you feel is best for the problem. In an actual commercial product C/C++ would be a great choice but you're not being asked to implement a commercial product you are in a classroom setting where topics on software design, development, and testing will be every bit as important as the topics on operating systems. Don't get bogged down in language issues. Focus on solving the problem, your instructor will introduce optimization concept as they see fit. Make sure your program works and that sound design and development practices have been used before wasting time worrying about efficiency and optimization.

Instruction Notation

address(label) – returns the address of a label
byte(register) – returns the first byte in a register
memory[address] – a location in memory

Comment

; This is a comment

Instruction Formats

All osX instructions are of fixed length (6 bytes long). The addressing mode of all operands can always be determined by checking the op code of the instruction. The <op code> of an instruction is always encoded as a single unsigned byte (e.g., a char in C). An operand which is a register is always encoded as a single unsigned byte (e.g., a char in C). Immediate operands and addresses are encoded as 4 byte integers. There are 7 different instruction formats:

Encoding

Two data types (char, int) were used to encode the instructions. When only a single byte was required the C data type of char or more specifically unsigned char. When 4 bytes were needed a simple int (4 byte signed integer) was used. The means you only need to worry about reading two data types from the .osx file, chars and integers that is it. No nibbles, no bit arrays, nothing overly complex.

1. 1 byte op code | 1 byte register | 4 bytes unused
2. 1 byte op code | 1 byte register | 4 byte immediate
3. 1 byte op code | 1 byte register | 4 byte address
4. 1 byte op code | 4 byte address | 1 byte unused
5. 1 byte op code | 4 byte immediate | 1 byte unused
6. 1 byte op code | 1 byte register | 1 byte register | 3 bytes unused
7. 1 byte op code | 1 byte register | 1 byte register | 1 byte register | 2 bytes unused

Arithmetic

[<label>] ADD <reg1> < reg2> < reg3> ; Add
Encode: 1 byte op code | 1 byte register | 1 byte register | 1 byte register | 2 bytes unused
ADD = 16

Suggested Usage: <reg1> ← < reg2> + < reg3>

[<label>] SUB <reg1> < reg2> < reg3> ; Subtract
Encode: 1 byte op code | 1 byte register | 1 byte register | 1 byte register | 2 bytes unused
SUB = 17

Suggested Usage: <reg1> ← < reg2> - < reg3>

[<label>] MUL <reg1> < reg2> < reg3> ; Multiple
Encode: 1 byte op code | 1 byte register | 1 byte register | 1 byte register | 2 bytes unused
MUL = 18

Suggested Usage: <reg1> ← < reg2> * < reg3>

[<label>] DIV <reg1> < reg2> < reg3> ; Divide
Encode: 1 byte op code | 1 byte register | 1 byte register | 1 byte register | 2 bytes unused
DIV = 19

Suggested Usage: <reg1> ← < reg2> / < reg3>

Move Data

[<label>] MOV <reg1> < reg2> ; Load data from register
Encode: 1 byte op code | 1 byte register | 1 byte register | 3 bytes unused
MOV = 1

Suggested Usage: <reg1> ← < reg2>

[<label>] MVI <reg1> <imm> ; Load register with immediate value
Encode: 1 byte op code | 1 byte register | 4 byte immediate
MVI = 22

Suggested Usage: <reg1> ← <imm>

[<label>] ADR <reg1> <label> ; Get address of label
Encode: 1 byte op code | 1 byte register | 4 byte address
ADR = 0

Suggested Usage: <reg1> ← address(<label>)

[<label>] STR <reg1> < reg2> ; Store word (int) using register indirect addressing
Encode: 1 byte op code | 1 byte register | 1 byte register | 3 bytes unused
STR=2

Suggested Usage: memory[<reg2>] ← <reg1>

[<label>] STRB <reg1> < reg2> ; Store byte using register indirect addressing
Encode: 1 byte op code | 1 byte register | 1 byte register | 3 bytes unused
STRB=3

Suggested Usage: memory[< reg2>] ← byte(memory[<reg1>])

[<label>] LDR <reg1> < reg2> ; Load word (int) using register indirect addressing
Encode: 1 byte op code | 1 byte register | 1 byte register | 3 bytes unused
LDR=4

Suggested Usage: <reg1> ← memory[<reg2>]

[<label>] LDRB <reg1> < reg2> ; Load byte using register indirect addressing
Encode: 1 byte op code | 1 byte register | 1 byte register | 3 bytes unused
LDRB=5

Suggested Usage: < reg1> ← byte(memory[<reg2>]

Branch

[<label>] B <label> ; Jump to label
Encode: 1 byte op code | 4 byte address | 1 byte unused
B = 7

Suggested Usage: PC ← address(<label>)

[<label>] BL <label> ; Jump to label and link (place the PC of the next instruction into a register)
Encode: 1 byte op code | 4 byte address | 1 byte unused
BL = 21

Suggested Usage: PC ← address(<label>) R5 ← PC+6

[<label>] BX < reg> ; Jump to address in register
Encode: 1 byte op code | 1 byte register | 4 bytes unused
BX = 6

Suggested Usage: PC ← <reg>

[<label>] BNE <label> ; Jump to label if Z register is not zero
Encode: 1 byte op code | 4 byte address | 1 byte unused

BNE = 8

Suggested Usage: PC ← address(<label>) if Z != 0

[<label>] BGT <label> ; Jump to label if Z register is greater than zero
Encode: 1 byte op code | 4 byte address | 1 byte unused
BGT = 9

Suggested Usage: PC ← address(<label>) if Z > 0

[<label>] BLT <label> ; Jump to label if Z register is less than zero
Encode: 1 byte op code | 4 byte address | 1 byte unused
BLT = 10

BLT = 10

Suggested Usage: PC ← address(<label>) if Z < 0

[<label>] BEQ <label> ; Jump to label if Z register is zero
Encode: 1 byte op code | 4 byte address | 1 byte unused
BEQ = 11

Suggested Usage: PC ← address(<label>) if Z == 0

Logical

[<label>] CMP < reg1> < reg2> ; Compare <reg1> and <reg2> place result in Z register
Encode: 1 byte op code | 1 byte register | 1 byte register | 3 bytes unused
CMP = 12

Suggested Usage: Z ← < reg1> - < reg2>

[<label>] AND < reg1> < reg2> ; Perform an AND operation on <reg1> and <reg2> result in Z register
Encode: 1 byte op code | 1 byte register | 1 byte register | 3 bytes unused
AND = 13

Note: The AND operation can be logical or bitwise either will work

Suggested Usage: Z ← < reg1> & < reg2> bitwise AND operation

Suggested Usage: Z ← <reg1> && <reg2> logical AND operation

[<label>] ORR < reg1> < reg2> ; Perform an OR operation on <reg1> and <reg2> result in Z register
Encode: 1 byte op code | 1 byte register | 1 byte register | 3 bytes unused
ORR = 14

Note: The ORR operation can be logical or bitwise either will work

Suggested Usage: Z ← < reg1> | < reg2> bitwise OR operation

Suggested Usage: Z ← <reg1> || <reg2> logical OR operation

[<label>] EOR **< reg1> < reg2> ; Exclusive OR operation on <reg1> and <reg2> result in Z register**
Encode: 1 byte op code | 1 byte register | 1 byte register | 3 bytes unused
EOR = 15

Note: The EOR operation can be logical or bitwise, there is little actual need for this instruction

Suggested Usage: Z ← < reg1> ^ < reg2> bitwise Exclusive OR operation

Suggested Usage: Z ← (<reg1> ||<reg2>) && !(<reg1> && <reg2>)

Interrupts

SWI <imm> ; Software Interrupt
Encode: 1 byte op code | 4 byte immediate | 1 byte unused
SWI = 20

Suggested Usage: Execute interrupt <imm>

Directives

[<label>] .WORD <integer> ; Allocate space for a positive or negative 4 byte integer

[<label>] .BYTE '<char>' ; Allocate space for printable ASCII character

[<label>] .BYTE \<value> ; Allocate space for a byte value 0…255

[<label>] .SPACE <value> ; Allocate 1…1024 bytes of space

Registers

R0 ; General Purpose Register, Encode as 0

> ***Suggested Usage: General purpose Register***

R1 ; General Purpose Register, Encode as 1

> ***Suggested Usage: general purpose Register***

R2 ; General Purpose Register, Encode as 2

> ***Suggested Usage: general purpose Register***

R3 ; General Purpose Register, Encode as 3

> ***Suggested Usage: general purpose Register***

R4 ; General Purpose Register, Encode as 4

> ***Suggested Usage: general purpose Register***

R5 ; General Purpose Register, Encode as 5

> ***Suggested Usage: general purpose Register***

SP ; Space Pointer Register, Encode as 6

> ***Suggested Usage: Point at top of run-time stack, used with SL register to test for Stack-Overflow and Out-Of-Memory.***

FP ; Frame Pointer Register, Encoded as 7

> ***Suggested Usage: Point at current frame on run-time stack***

SL ; Stack Limit Register, Encode as 8

> ***Suggested Usage: Limit the size of the run-time stack, set this register in your VM to the byteSize value stored as the first value in the osX byte file. Can be used to test for Stack-Overflow and Out-Of-Memory***

Z ; Flag Register, Encode as 9

Suggested Usage: Set when CMP instruction is run, used by conditional branch instructions

---

SB ; Stack Base Register, Encode as 10

Suggested Usage: Limit the size of memory in your VM, set this register in your VM to the maximum size of memory. Can be used to test for Stack-Underflow.

PC ; Program Counter, Encode as 11

Suggested Usage: Point to next instruction to execute. Never directly set this register using assembly code.

---

osX byte code file

The osX Assembler will read an osX .asm file and convert this into byte code. The osX byte code file contains a three integer header before the first line of assembly code in encoded.

- Integer One – Is the size of the byte code file, minus the three integer header.
- Integer Two – Is the location of the first instruction found in the assembly file. You should set the PC of your VM to this value. As before this value doesn't account for the three integer header.
- Integer Three – Is a loader address you can specify at the command-line of the osX Assembler. **Your professor** will explain this value and how to use as it; not really related to the assembler or vm directly, it just provide you and your instructor some flexibility when loading byte code.

NOTE: It is suggested you write your osX Assembly code as directives followed by instructions, the osX Assembler will convert your even if you write it instructions followed by directives. You should never write assembly code with instructions and directives intermixed.