ASSEMBLY NOTES

Luke Zeitlin

May 16, 2019

Contents

1 Three Sections

An assembly program has the following three sections: Data, BSS, Text.

1.1 data

The following syntax denotes the beginning of the data section:

```
section .data
```

This section contains initialized data and constants. Does not change at runtime.

1.2 bss

This denotes the beginning of the variable section:

```
section .bss
```

The bss section contains variable declerations.

1.3 text

This secion contains the actual code.

```
section .text
  global _start
_start:
```

Global start lets the kernel know where the program execution begins.

2 Comments

Comments are preceded with a semicolon. Comments can contain any printable character

```
; this is a full line comment
add eax, ebx  ; this is an inline comment
```

3 Statments

There are three types of statments:

- instructions

  These tell the processor what to do.Each instruction has an op-code.

- directives or pseudo-ops

These tell the assembler about the various aspects of the assembly process. These are non-executable.

- macros

Text substitution

4 Tool Chain

The softare tools used in creating assembly programs are as follows:

- Assembler
- Linker
- Loader
- Debugger

4.1 Assembler

The assembler is a program converts assembly code into machine language (binary). The output file is known as an object file, hence the .o suffix. During this process the variable names get removed and converted into addresses.

4.1.1 Assembler commands for yasm

The following command will create a list file for a given program written in a .asm:

```
yasm -g dwarf2 -f elf64 example.asm -l example.lst
```

- `-g dwarf2` instructs the assembler to include debugging information in the final objet file.
- `-f elf64` instructs the assembler to output a ELF64 object file, sutable for 64-bit linux systems.
- `-l example.lst` tells the assembler to create a lise file.

1. List File

   This file gives a line by line mapping of assembly to machine language. It can be useful for debugging.

2. Two Pass Assembler

   Since assembly language can have control flow commands, such as jumps, if statemtents, etc. Assembly is not necesserily executed linerly. To create machine code from an assembly programm the assembler takes two passed over the code.

   (a) First Pass

   Often this includes tasks such as creating a symbol table, expanding macros and evaluating constant expressions.

   (b) Second Pass

   This usually includes the final generation of the code, creation of the list file, if required, and creating the object file.

4.2 Linker

Also known as the linkage editor. This combines object files into a single executable. It also includes any libraries required for execution. The following is a command for the GNU Gold linker:

```
ld -g -o example example.o
```

The `-g` flag tells the linker to include debugging information. The `-o` flag specifies the output file, here `example`. Multiple object files can be linked together. When using a function from another file, the function must be flagged with `extern`.

4.2.1 Dynamic Linking

Linux supports dynamic linking. This allows resolution of some symbols be postponed untill the execution of the program. Under Linux dynamically linked object files have the extention `.so`, shared object. The Windows equivilant is `.dll`.

4.2.2 Assemble/Link Script

The following is an example of a bash script to automate the calls to the assembler and linker into a single call.

```
#!/bin/bash

if [ -z $1 ]; then
  echo "Usage: ./asm64 <asmMainFile> (no extension)"
  exit
fi

# verify no extentions were entered
if [ ! -e "$1.asm" ]; then
  echo "Error, $1.asm not found."
  echo "Note, do not enter file extensions."
  exit
fi

#  Compile, assemble, and link

yasm -Worphan-lables -g dwarf2 -f elf64 $1.asm -l $1.lst ld -g -o $1 $1.o
```

4.3 Loader

This is the part of the operating system that loads the program from secondary storage into memory. Under Linux this is done with the program name. For example, if the program is called `hello_world`, the command will be:

```
./hello_world
```

4.4 Debugger

This is a program that can control the execution of the assembly program in order to inspect how it is (or is not) working.

5 Syntax of Statements

Assembly language has one statement per line

```
[label] mnemonic [operands] [; comment]
```

Fields in the square brackets are optional. There are two basic parts to the instruction - the name (nmemonic) and the operands.For example:

```
INC COUNT ; increment the variable COUNT

MOV TOTAL ; Transfer the total value 48 into memory variable TOTAL
```

6 Assembly Hello World

```
section .text
  global _start  ; must be declared for linker
_start:
  mov edx,len ; message length
  mov ecx,msg ; message to write
  mov ebx,1   ; file descriptor (stdout)
  mov eax,4   ; system call number (sys_write)
  int 0x80    ; call kernel

  mov eax,1   ; system call number (sys_exit)
  int 0x80    ; call kernel

section .data
msg db 'Hello, world!', 0xa ; string to be printed
en equ $ - msg ; length of the string
```

7 Compiling and linking with nasm

- save the above as a file with extension .asm, for example: hello.asm
- assemble program with:

```
nasm -f elf hello.asm
```

- if no errors, hello.o will have been created
- To link the object file and create the executable file named hello:

```
ld -m elf_i386 -s -o hello hello.o
```

- execute with:

```
./hello
```

8 Memory Segments

8.1 Segmented memory model:

In a segmented memory model the system memory is divided into idependent segments. Segments are used to store specific types of data. One segment for instruction codes, one for data elements, etc.

8.2 Data segment

Represented by the `.data` section and the `.bss` section. The `.data` section is holds static data that remains unchanges during the course of the program. The `.bss` section is also for static data. Data here are declared during the course of the program. The `.bss` section is zero filled prior to execution.

8.3 Code segment

Represented by the `.text` section. Fixed data that stores instruction codes.

8.4 Stack

This contains data passed to functions and procedures during the course of a program.

9 Registers

In order to avoid the slow process of reading and storing data in memory, the processor has temporary storage locations called `registers`. These can store data elements for processing without having to access memory.

9.1 Processor Registers

The 32 bit processor has 10 registers. These are grouped into the following categories:

- General (Data, Pointer, Index)
- Control
- Segment

9.1.1 General Registers

1. Data

   These are used for arithmetic, logic and other operations. They have three different modes of usage:

   - As complete 32-bit registers: EAX, EBX, ECX, EDX
   - The lower halves can be used as four 16 bit data registers: AX, BX, CX, DX
   - The lower halves of the above 16 bit registers can be used as eight 8-bit registers: AH, AL, BH, BL, CH, CL, DH, DL

```
.....................+AX++Accumulator+
EAX |----------------|---AH---|---AL---|

.....................+++++BX++Base++++
EBX |----------------|---BH---|---BL---|

.....................+++CX++Counter+++
ECX |----------------|---CH---|---CL---|

.....................++++DX++Data+++++
EDX |----------------|---DH---|---DL---|
```

   - AX - `Primary Accumulator`

   I/O for most arithmetic instructions, for example multiplication. One operand is stored in other EAX, AX or AL depending on size.

   - BX - `Base`

Sometimes used in index addressing.

- CX - `Count`

Stores loop counts in various iterative operations

- DX - `Data`:

Also used in I/O. Notably when large numbers are involved.

2. Pointer Registers

   Stores addresses in memory. In 32-bit these are EIP, ESP and EBP. In 16-bit these correspond to IP, SP and BP.

   - IP - `Instruction Pointer`

   Stores the `offset address` of the next instruction to be executed. In combination with the `CS` register (CS:IP) gives the full address of the current instruction in code segment.

   - SP - `Stack Pointer`

   Provides the offset value in the program stack. In combination with the `SS` register (SS:SP) gives the current position of data or address in the program stack.

   - BP - `Base Pointer`

   Helps in referencing the parameter variables passed to a subroutine. The address in `SS` in combination with the offset BP gives the location of a parameter. Can also be combined with DI and SI as a base register for special addressing.

3. Index Registers

   ESI and EDI in 32-bit, or SI and DI in 16-bit.

   - SI - `Source Index`

   Source index for string operations

   - DI - `Destination Index`

   Destination index for string operations.

9.1.2 Control

For comparisons and conditional instructions that control flags.

- OF - `Overflow Flag`

Indicates overflow of leftmost bit in a signed math operation

- DF - `Direction Flag`

In string comparison operations, indicates left or right direction of movement. 0 for left-to-right and 1 is right-to-left

- IF - `Interrupt Flag`

Flags if keyboard or other interupts are to be ignored or processed. 0 for ignored or 1 for processed.

- TF - `Trap Flag`

Allows the procssor to wor in single step mode for debug perposes.

- SF - `Sign Flag`

Indicates the sign of a arithmetic result.

- ZF - `Zero Flag`

Indicates whether a result of an arithmetic expression is zero.

- AF - `Auxiliary Carry Flag`

Used for specialized arithmetic to carry from bit 3 to bit 4.

- PF - `Parity Flag`

Indicates the total number of 1 (on) bits in the result of an arithmetic expression. If even then 0, odd then 1.

- CF - `Carry Flag`

Contains the carry from the leftmost bit after an atrithmetic operation. It also stores the contents of the last bit of a `shift` or `rotate` operation.

Table 1: Positions of flag in the flags register

Flag					O	D	I	T	S	Z		A		P		C
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

9.1.3 Segment Registers

These refer to specific areas defined for data, code and stack.

- CS - `Code Segment`

Contains the starting address of the code segment.

- DS - `Data Segment`

Contains the starting address of the data segment.

- SS - `Stack Segment`

Contains the starting address of the stack segment.

There are additional segment registers: ES, FS, GS.

All memory locations within a segment are relitive to the starting address of the segment. Since all segments will start at an address that is evently divisable by 16 (hex 10) there is always a zero in the rightmost hex digit. This zero is not stored in segment registers.

9.2 Example of using registers

```
section .text
  global _start   ; must be declared for linker (gcc)

_start:           ; tell linter entry point
  mov edx,len     ; message length
  mov ecx,msg     ; message to write
  mov ebx,1       ; file descriptor (stout)
  mov eax,4       ; system call number (sys_write)
  int 0x80        ; call kernel

  mov edx,9       ; message length
  mov ecx,s2      ; message to write
  mov ebx,1       ; file descriptor (stout)
  mov eax,4       ; system call number (sys_write)
```

```
  int 0x80        ; call kernel

  mov eax,1       ; system call number (sys_exit)
  int 0x80        ; call

section .data
msg db 'Displaying 9 stars',0xa  ; a message
len equ $ - msg ; length of message
s2 times 9 db '*'
```

10 System Calls

API between the the `user space` and the `system space`. System calls are used by putting the number associated with that call into `EAX` and the arguments to that system call into other specific registers.

For example, this is the call to exit the program. `sys_exit`:

```
mov  eax,1 ; system call number moved into eax
int  0x80   ; call kernel
```

Here is an example for a syscall that has arguments, `sys_write`:

```
mov  edx,4    ; message length
mov  ecx,msg  ; some message that has been defined in the data section
mov  ebx,1    ; file descriptor (1 is for standard out)
mov  eax,4    ; system call number (sys_write)
int  0x08     ; call kernel
```

All syscalls are listed in /usr/include/asm/unistd.h which can be used to look up their numbers. The following is a table of commonly used system calls with their arguments:

EAX (number)	Name	EBX	ECX	EDX	ESX	EDI
1	sys_{exit}	int‘				
2	sys_{fork}	struct pt_{regs}				
3	sys_{read}	unsigned‘int	char	$size_t$		
4	sys_{write}	unsigned int‘	const char	$size_t$		
5	sys_{open}	const char*	int	int		
6	sys_{close}	unsigned int				

11 Instructions

11.1 Move

```
mov <dest>, <src>
; for example

mov ax, 42 ; the integer 42 is put into the 16 bit ax register
mov cl, byte [bvar]  ; into the lower c register, a byte is copied from the address of bvar
mov qword [qvar], rdx ; a quad word from the address of qvar is copied into the 64 bit d reg
```

- Copies data
- Source and destination cannot both be in memory.
- when copying a double word into a 64 bit register, the upper portion of the register is set to zeros.

11.2 Address

The load effective address command `lea` is used to put the address of a variable into a register.

```
lea <reg64>, <mem>

; for example
lea rcx, byte [bvar] ; put the location of bvar into the rcx register
```

11.3 Convert

Conversion instructions change a variable from one size to another. Narrowing conversions require no specific instructions since the lower portions of registers are directly accessable.

```
mov rax, 50
mov byte [bval], al
```

Widening conversions vary depending on the data types involved.

11.3.1 widening - unsigned

Unsigned numbers only take poisitive values, therefore when dealing with unsigned numbers the upper part of the memory location or register must be set to zero.

```
mov al, 50
mov rbx, 0
mov bl, al
```

There is an instruction especially for performing this: `movzx`

```
movzx  <dest>, <src>
```

NB: This does not work when converting a quadword destination with a double word source operand. However, simply using mov in this situation will achive the desired result since it will set the upper portion of the register or memory location to zeros.

11.3.2 widening - signed

When the data is signed, the upper portion must be set to ether zeros or ones depending on the sign of the number.

```
movsx <dest>, <src> ;general form, used always except when converting between double and qua
movsxd <dest>, <src> ; used then converting from double to quadword
```

Specific registers also have their own signed widening conversion instructions:

instruction	use
cbw	from byte in al to word in ax
cwd	from word in ax to double word in dx:ax
cwde	from word in ax to double word in eaxy
cdq	from double word in eax to quadword in edx:eax
cdqe	from double word in eax to quadword in rax
cqo	from quadword in rax to double quadword in rdx:rax

11.4 Arithmetic

11.4.1 Addition

```
add <dest>, <src> ; this results in: <dest> = <dest> + <src>
```

Operands must be of the same type. Memory to memory addition cannot use the above. One of the operands must be moved into a register.

```
; Num1 + Num2 (memory to memory) assuming that both are byte size.
mov  al, byte [Num1]
add  al, byte [Num2]
mov  byte [Ans], al
```

There is also a command for incrementing a value by 1.

```
inc <operand>
; for example:
inc rax
; when incrementing an operand in memory, specify the size:
inc byte [bNum]
```

When the numbers being added will result in a sum greater than the register size of the machine, it is necessery to add with a carry. In this situation the `Least Significant Quadword` is added with an `add` instruction, then the `Most Significant Quadword` is added with an `adc` (add with carry). The second addition must immediatly follow the first so that the `carry flag` is not altered by anything else.

```
dquad1  ddq 0x1A000000000000000
dquad2  ddq 0x2C000000000000000
dqsum   ddq 0

; using the declarations above:


mov  rax, qword [dquad1]    ; the first 64 bits of dquad1
mov  rdx, qword [dquad1+8]  ; the last 64 bits of dquad1


add  rax, qword [dquad2]    ; add the first 64 bits of dquad2
adc  rdx, qword [dquad2+8]  ; add with carry the last 64 bits of dquad2

mov qword [dqSum], rax      ; result is put into dqSum
mov qword [dqSum+8], rdx
```

11.4.2 Subtraction

The subtraction commands are self-explanitory when taken with the above information on addition.

```
sub <dest>, <src>
dec <operand>
```

11.4.3 Multiplication

There are different commands for multiplying signed or unsigned integers. Both typically produce double sized results.

1. Unsigned Integer Multiplication The genral form is as follows:

   ```
   mul  <src>
   ```

   One of the operands must use an A register (al, ax, eax, rax) depending on size. The result is placed in the A (and possibly D) registers.

size	register	operand	output registers
byte	al	op8	ah, al
word	ax	op16	dx, ax
double word	eax	op32	edx, eax
quad word	rax	op64	rdx, rax

   For example, if two double words are multiplied, the result will be a quad word in dx:ax

   ```
   dNumA dd 42000
   dNumB dd 73000
   Ans   dq 0

   ; Using the above declarations
   ; dNumA * dNumB

   mov  eax, word [wNumA]
   mul  dword [wNumB]    ; result goes to edx:eax
   mov  dword [Ans], ax
   mov  dword [Ans+2], bx
   ```

2. Signed Integer multiplication Signed integer multiplication is more flexable with its operands / sizes. The destination must always be a register.

```
imul <src>
imul <dest>, <src/imm>
imul <dest>, <src>, <imm>
```

   - When one operand is used then `imul` works like `mul`, but the operands are interpreted as signed.
   - If two operands are used then the source and destination values are multiplied and the destination value is overwritten. In this case, the source may be an immediate value, a register or a location in memory. A byte size destination operand is not supported.
   - When three operands are used, the last two are multiplied and the product is placed in the destination. The `src` must not be an immediate value. The `imm` must be an immediate value. The result is truncated to the size of the destination operand. Byte size destination is not supported.

12 Related Topics

12.1 Addressing data in memory

The process through which execution is controlled is called the `fetch-decode-execute cycle`. The instruction is fetched from memory. The processor can access one or more bytes of memory at a given time. The processor stores data in `reverse-byte sequence`.

For example, for hex number 0725H:

```
In register:
|--07--|--25--|
In memory:
|--25--|--07--|
```

Table 2: Shows access speeds for different types of storage

Memory Unit	Example Size	Typical Speed
Processor Registers	16 to 64 bit registers	~ 1 nanosecond
Cache Memory	4 - 8+ Megabytes (L1 and L2)	~ 5 to 60 nanoseconds
Primary Storage (RAM)	2 - 32 Gigabytes	~ 100 to 150 nanoseconds
Secondary storage (HDD)	500 Gigabtes to 4+ Terabytes	~ 3-15 miliseconds

12.2 Memory Hierachy

12.3 Integer representation

size name	size	unsigned range	signed range
byte	2^8	0 - 255	-128 - 127
word	2^{16}	0 - 65535	-32,768 - 32767
double word	2^{32}	0 - 429497294	-2147483648 2147483647
quadword	2^{64}	0 - 2^{64} -1	-(2^{63}) - 2^{63} -1
double quadword	2^{128}	0 - 2^{128} -1	-(2^{127}) - 2^{127} -1

12.4 Two's Complement

Signed numbers are often represented in twos complement form. A negitive representation of a positive number can be made by flipping the bits and then adding 1. For example:

9	00001001
step 1	11110110
step 2	11110111
-9	11110111