Contents

asm section .bss #+END$_{\mathrm{SRC}}$ The bss section contains variable declarations.

0.1 text

This section contains the actual code.

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

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

1 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
```

2 Statements

There are three types of statements:

- 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

3 Tool Chain

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

- Assembler
- Linker
- Loader
- Debugger

3.1 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.

3.1.1 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 object file.
- `-f elf64` instructs the assembler to output a ELF64 object file, suitable for 64-bit Linux systems.
- `-l example.lst` tells the assembler to create a list 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 statements, etc. Assembly is not necessarily executed linearly. To create machine code from an assembly program 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.

3.2 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`.

3.2.1 Dynamic Linking

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

3.2.2 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 extensions 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
```

3.3 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
```

3.4 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.

3.5 Compiling and linking with nasm

The above uses yasm. Here is an example of using the nasm assembler:

- 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
```

4 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 (mnemonic) and the operands.For example:

```
INC COUNT ; increment the variable COUNT

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

5 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
```

6 Memory Segments

6.1 Segmented memory model:

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

6.2 Data segment

Represented by the `.data` section and the `.bss` section. The `.data` section is holds static data that remains unchanged 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.

6.3 Code segment

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

6.4 Stack

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

7 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.

7.1 Processor Registers

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

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

7.1.1 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.

7.1.2 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 interrupts are to be ignored or processed. 0 for ignored or 1 for processed.

- TF - `Trap Flag`

Allows the processor to work in single step mode for debug purposes.

- 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 arithmetic 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

7.1.3 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 relative to the starting address of the segment. Since all segments will start at an address that is evenly divisible by 16 (hex 10) there is always a zero in the rightmost hex digit. This zero is not stored in segment registers.

7.2 Example of using registers

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

_start:           ; tell linker 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 '*'
```

8 System Calls

9 Related Topics

9.1 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--|
```

9.2 Memory Hierarchy

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 Gigabytes to 4+ Terabytes	~ 3-15 milliseconds