APPENDIX A

Assembly Syntax Translation

Objective

Appendix A provides some general rules and a table for translating code between Assemblers and syntaxes.

Rules

- GAS prefixes registers with %
- GAS prefixes immediate values with $
- GAS also uses the $ prefix to indicate an address of a variable
- MASM and NASM use $ as the *current location counter*, while GAS uses the dot (.)
- GAS operands are source first, destination second
- MASM and NASM operands are destination first, source second
- GAS denotes operand sizes with *b*, *w*, *l*, and *q* suffixes on the instruction
- GAS and NASM identifiers are case-sensitive
- MASM identifiers are not case-sensitive by default, but can be by adding `option casemap:none` (usually after the .MODEL directive)
- NASM writes FPU stack registers as *ST0*, *ST1*, etc…without parentheses
- GAS/MASM usually write FPU stack registers as *%st(1)*/*ST(1)*, *%st(2)*/*ST(2)*, etc…with parentheses
- GAS uses .equ to set a symbol to an expression, NASM uses EQU, and MASM uses = or EQU
- All three Assemblers can use single or double quotes for strings
- MASM relies more on assumptions (e.g., data sizes), so sometimes interpreting what an instruction does can be difficult
- NASM usually does not require a size directive for source operands, but a size directive can be used. A size directive is required for destination operands.

```
mov eax, [test]          ; source size is not required
mov eax, DWORD [test]    ; but can be used
mov DWORD [test], eax    ; required for destination
```

Operation	GAS	NASM	MASM
Clear a register (*eax*)	`xorl %eax, %eax`	`xor eax, eax`	
Move contents of *eax* to *esi*	`movl %eax, %esi`	`mov esi, eax`	
Move contents of *ax* to *si*	`movw %ax, %si`	`mov si, ax`	
Move immediate byte value 4 to *al*	`movb $4, %al`	`mov al, 4`	
Move contents of address `0xf` into *eax*	`movl 0x0f, %eax`	`mov eax, [0x0f]`	`mov eax, ds:[0fh]`

Operation	GAS	NASM	MASM
Move contents of variable `temp` into *eax*	`movl temp, %eax`	`mov eax, DWORD [temp]`	`mov eax, temp`
Move address of variable `temp` into *eax* using MOV	`movl $temp, %eax`	`mov eax, temp`	`mov eax, OFFSET temp`
Load address of variable `temp` into *eax* using LEA (32-bit)	`leal temp, %eax`	`lea eax, [temp]`	`lea eax, temp`
Load address of variable `temp` into *rax* using LEA (64-bit)	`leaq temp(%rip), %rax`	`lea rax, [rel temp]`	`lea rax, temp`
Move contents of *eax* into variable `temp`	`movl %eax, temp`	`mov DWORD [temp], eax`	`mov temp, eax`
Move immediate byte value 2 into `temp`	`movl $2, temp`	`mov BYTE [temp], 2`	`mov temp, 2`
Move immediate byte value 2 into memory pointed to by *eax*	`movb $2, (%eax)`	`mov BYTE [eax], 2`	`mov BYTE PTR [eax], 2`
Move immediate word value 4 into memory pointed to by *eax*	`movw $4, (%eax)`	`mov WORD [eax], 4`	`mov WORD PTR [eax], 4`
Move immediate doubleword value 6 into memory pointed to by *eax*	`movl $6, (%eax)`	`mov DWORD [eax], 6`	`mov DWORD PTR [eax], 6`
Include file syntax	`.include "file.ext"`	`%include "file.ext"`	`INCLUDE file.ext`
Identifier syntax	`identifier: type value`		`identifier type value`
Get size of array in bytes using *current location counter* (code directly after array declaration)	`aSize: .long (. - array)`	`aSize: EQU ($ - array)`	`aSize = ($ - array)`

Create and use a symbol with EQU	`.equ temp, (2 * 6 / 3)`
	`mov $temp, %eax`
		`temp: EQU (2 * 6 / 3)`
		`mov eax, temp`
			`temp EQU (2 * 6 / 3)`
			`mov eax, temp`

Reserve 64 bytes of memory	`.space 64`	`resb 64`	`db 64 DUP (?)`
Create uninitialized 32-bit variable `temp`	`.lcomm temp, 4`	`temp: resd 1`	`temp DWORD ?`
Create initialized 32-bit variable `temp` with value 5	`temp: .long 5`	`temp: dd 5`	`temp DWORD 5`
Create an array w/ 32-bit values	`temp: .long 5, 10, 15`	`temp: dd, 5, 10, 15`	`temp DWORD 5, 10, 15`
Create "Hello, World" string (code on one line)	`identifier: .ascii "Hello, World"`	`identifier: db 'Hello, World'`	`identifier BYTE "Hello, World"`
Create "Hello, World" w/ newline and null (code on one line)	`identifier: .asciz "Hello, World\n"`	`identifier: db 'Hello, World', 10, 0`	`identifier BYTE "Hello, World", 10, 0`

Function structure	`identifier:`
	…
	`ret`
			`identifier PROC`
			…
			`ret`
			`identifier ENDP`

Operation	GAS	NASM	MASM
Program segments (sections)	.data .bss .text	SECTION .data SECTION .bss SECTION .text	.data .code
Data types	.byte .word .long .quad	db dw dd dq	BYTE WORD DWORD QWORD
Repetition (code on one line)	identifier: .fill count, size, value	identifier: TIMES count type value	identifier type count DUP (value)
Macro definition	.macro identifier arg1, arg2… args referenced as \arg1 .endm	%macro identifier argcount args referenced as [%1] %endmacro	identifier MACRO arg1, arg2… args referenced as arg1 ENDM
Macro usage	identifier param1, param2, etc...		
Comment (single-line)	# this is a comment	; this is a comment	
32-bit _main exit routine	# for GAS/Clang on Mac pushl $0 subl $4, %esp movl $1, %eax int $0x80 # for GAS/Clang on Linux mov $1, %eax mov $0, %ebx int $0x80	; for NASM on Linux mov eax, 1 mov ebx, 0 int 80h ; for NASM on Mac push DWORD 0 sub esp, 4 mov eax, 1 int 80h	; before .data segment ExitProcess PROTO, dwExitCode:DWORD ; before _main ENDP INVOKE ExitProcess, 0
64-bit _main exit routine	# for GAS/Clang on Mac movq $0x2000001, %rax movq $0, %rdi syscall # for GAS/Clang on Linux mov $60, %rax xor %rdi, %rdi syscall	; for NASM on Linux mov rax, 60 xor rdi, rdi syscall ; for NASM on Mac mov rax, 2000001h mov rdi, 0 syscall	; before .data segment extrn ExitProcess:proc ; before _main ENDP mov rcx, 0 call ExitProcess