APPENDIX A

Assembly Syntax Translation

Objective

Appendix A provides some general rules and a table for translating code between Assemblers and syntaxes. Examples are provided for x86 and x86_64.

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 in 32-bit programs).
- 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 *rip*-relative addressing can be noted explicitly, but is only required if external functions are used: `mov rax, [rel test]`; GAS always requires explicit notation: `movq test(%rip),%rax`.
- 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 rax, [test]         ; source size is not required
mov rax, DWORD [test]   ; but can be used
mov DWORD [test], rax   ; required for destination
```

Operation	GAS	NASM	MASM
Clear a register (*rax*)	`xorq %rax, %rax`	`xor rax, rax`	
Move contents of *rax* to *rsi*	`movq %rax, %rsi`	`mov rsi, rax`	
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`	

Operation	GAS	NASM	MASM
Move contents of address `0xf` into *eax*	`movl 0x0f, %eax`	`mov eax, [0x0f]`	`mov eax, ds:[0fh]`
Move contents of variable `temp` into *rax*	`movq temp(%rip), %rax`	`mov rax, QWORD [temp]`	`mov rax, temp`

Move address of variable `temp` into *eax/rax* using MOV	`movl $temp, %eax`
	`# no absolute addressing`
	`# in 64-bit, use LEA`
		`mov eax, temp`
		`mov rax, temp`
			`mov eax, OFFSET temp`
			`mov rax, 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]`
		`; rel required if`
		`; external functions used`
			`lea rax, temp`

Move contents of *rax* into variable `temp`	`movq %rax, temp(%rip)`	`mov QWORD [temp], rax`	`mov temp, rax`

Move immediate byte value 2 into `temp`	`movb $2, temp(%rip)`	`mov BYTE [temp], 2`	`mov temp, 2`
			`mov BYTE PTR temp, 2`

Move immediate byte value 2 into memory pointed to by *eax/rax*	`movb $2, (%eax)`
	`movb $2, (%rax)`
		`mov BYTE [eax], 2`
		`mov BYTE [rax], 2`
			`mov BYTE PTR [eax], 2`
			`mov BYTE PTR [rax], 2`

Move immediate word value 4 into memory pointed to by *eax/rax*	`movw $4, (%eax)`
	`movw $4, (%rax)`
		`mov WORD [eax], 4`
		`mov WORD [rax], 4`
			`mov WORD PTR [eax], 4`
			`mov WORD PTR [rax], 4`

Move immediate doubleword value 6 into memory pointed to by *eax/rax*	`movl $6, (%eax)`
	`movl $6, (%rax)`
		`mov DWORD [eax], 6`
		`mov DWORD [rax], 6`
			`mov DWORD PTR [eax], 6`
			`mov DWORD PTR [rax], 6`

Move immediate quadword value 8 into memory pointed to by *eax/rax*	`movq $8, (%eax)`
	`movq $8, (%rax)`
		`mov QWORD [eax], 8`
		`mov QWORD [rax], 8`
			`mov QWORD PTR [eax], 8`
			`mov QWORD PTR [rax], 8`

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: .quad (. - array)`	`aSize: EQU ($ - array)`	`aSize = ($ - array)`

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

Reserve 64 bytes of memory	`.space 64`	`resb 64`	`db 64 DUP (?)`

Create uninitialized 32-bit/64-bit variable `temp`	`.lcomm temp, 4`
	`.lcomm temp, 8`
		`temp: resd 1`
		`temp: resq 1`
			`temp DWORD ?`
			`temp QWORD ?`

Create initialized 32-bit/64-bit variable `temp` with value 5	`temp: .long 5`
	`temp: .quad 5`
		`temp: dd 5`
		`temp: dq 5`
			`temp DWORD 5`
			`temp QWORD 5`

Create an array w/ 32-bit/64-bit values	`temp: .long 5, 10, 15`
	`temp: .quad 5, 10, 15`
		`temp: dd, 5, 10, 15`
		`temp: dq, 5, 10, 15`
			`temp DWORD 5, 10, 15`
			`temp QWORD 5, 10, 15`

Create "Hello, World" string (code on one line)	`identifier: .ascii "Hello, World"`	`identifier: db 'Hello, World'`	`identifier BYTE "Hello, World"`

Operation	GAS	NASM	MASM
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`

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`
	`xorq %rdi, %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`
		`xor rdi, rdi`
		`syscall`
			`; before .data segment`
			`extrn ExitProcess:proc`
			
			`; before _main ENDP`
			`xor rcx, rcx`
			`call ExitProcess`