Assembly Language Programming I

x86-64 Architecture

CSCI 2050U - Computer Architecture

Randy J. Fortier
@randy_fortier

OntarioTech UNIVERSITY

Outline

- Assembly language programming
  - Development tools
  - Registers
  - Variables - the data section
  - A basic program

Development Tools

CSCI 2050U - Computer Architecture

OntarioTech UNIVERSITY

Language Translators

- *Interpreters* translate each high-level language source code line every time it is needed for execution
  - e.g. JVM, Python
- *Compilers* translate HLL source code to object code that is almost ready for the CPU to execute
  - e.g. C++
- *Assemblers* translate assembly language – a low level language – to object code
  - Two popular assemblers for Linux: `nasm` and `yasm`

Linker

- Object code files produced by a compiler or assembler are not quite ready for execution
- A linker combines object code files and prepares them to be loaded into memory for execution
- Two popular linkers for Linux: `ld` and `gcc`'s linker

Debugger

- Allows the programmer to control execution of a program
  - Step through instructions one at a time
  - Stop at a preset breakpoint
- Lets you look at memory or register contents
  - Helps find programming errors
  - Helps understand how the computer works

Integrated Development Environments

- Single interface provides access to text editor, compiler or assembler, linker, and debugger
- IDEs for developing assembly language:
  - Microsoft Visual Studio
  - Code::Blocks
  - Eclipse
  - SASM

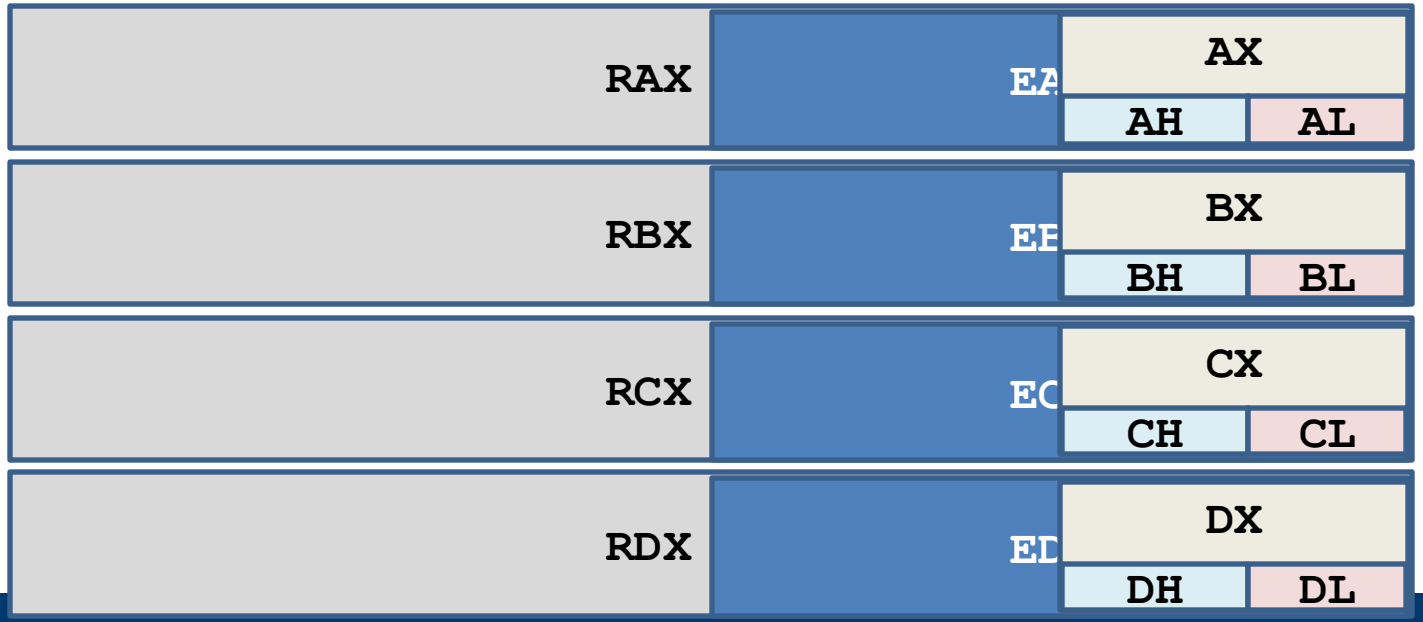
Registers

CSCI 2050U - Computer Architecture

OntarioTech UNIVERSITY

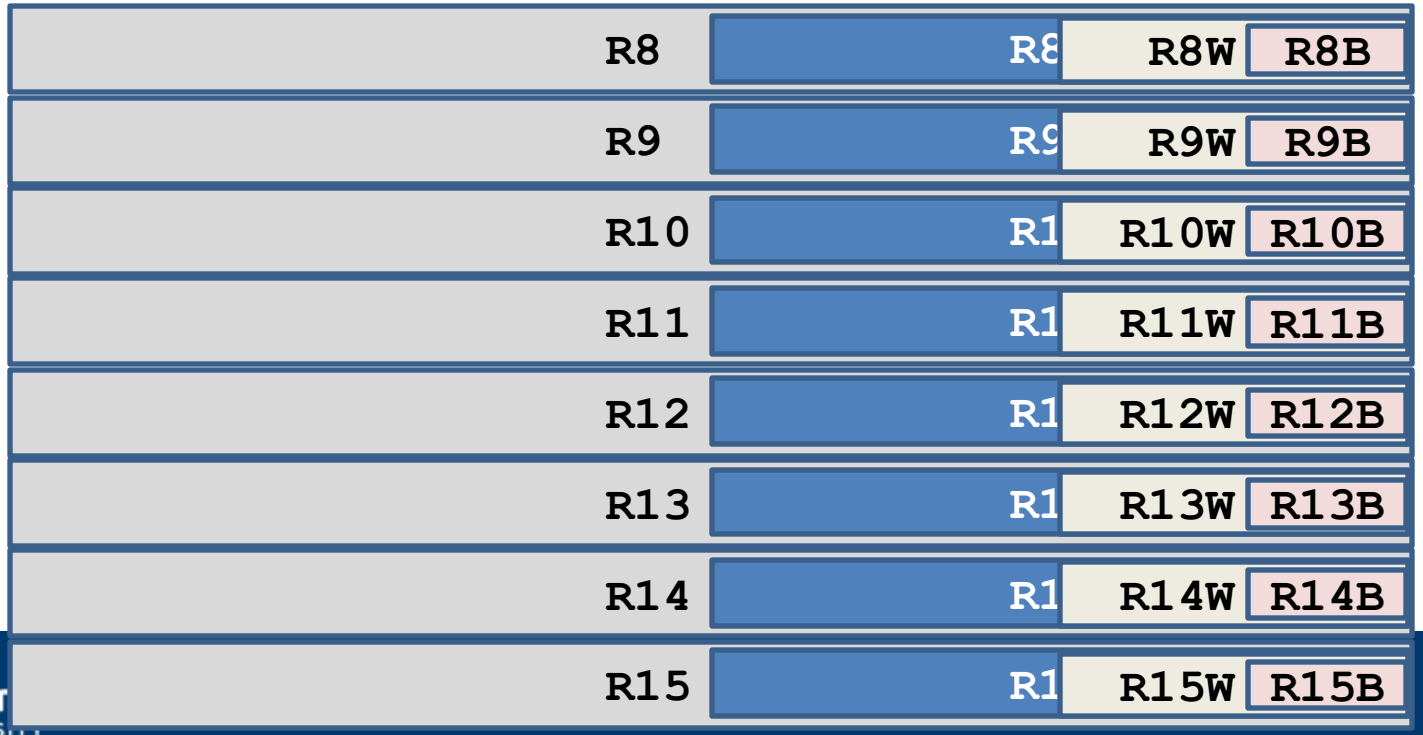
General Purpose Registers

RAX, **RBX**, **RCX**, **RDX**, each 64 bits long (quadword)



General Purpose Registers

- These registers are new to x64 (64-bit)



Index Registers

- **`RSI`** - source index
  - Source address in string moves
  - Array index
  - General purposes

- **`RDI`** - destination index
  - Destination address in string moves
  - Array index
  - General purposes

Stack Registers

- **`RSP`** - stack pointer
  - Holds address of top of stack frame

- **`RBP`** - base pointer
  - Used in procedure calls to hold address of reference point in the stack (i.e. bottom of stack frame)

Other Registers

- **`RIP`** - instruction pointer
  - Holds address of next instruction to be fetched for execution

- **`EFLAGS`** - flags
  - Collection of flags, or status bits
  - Records information about many operations
    - Carry Flag (CF) is bit 0
    - Zero Flag (ZF) is bit 6
    - Sign Flag (SF) is bit 7
    - Overflow Flag (OF) is bit 11

Variables

CSCI 2050U - Computer Architecture

Variables

- Variables are often declared in their own sections of the program
  - `.data` – initialized data
  - `.bss` – uninitialized data
  - `.rodata` – read only (initialized) data
- Variables can also be:
  - Stored on the calling stack (i.e. local variables)
  - Allocated on the heap (e.g. with malloc())

Data Sizes (Types)

- Assembly language does not have types per se

  e.g. If you want a number to be signed, you need to initialize it properly, and use instructions intended for signed numbers

- It does support sizes, however

  `db` – byte (8-bit)

  `dw` – word (16-bit)

  `dd` – double word (32-bit)

  `dq` – quad word (64-bit)

  `do` – octo word (128-bit)

The `data` Section

- Registers usually act like local variables
  - You tend to re-use the same registers for more calculations or operations later, however
- The data section is where you define your global variables

```
section .data
  promptFormat db "%s", 0
  prompt db "Enter a number: ", 0

  inputFormat db "%d", 0
  number dq 0           ; int number = 0;

  resultFormat db "The result is %d.", 0ah, 0dh, 0
```

The `data` Section

- Each declaration consists of a name, a type/size, and a value
- Define a single string (an array of `b`ytes) containing 'hello':

```
greeting db "hello", 0
```

- Define a single (`q`uadword) integer containing zero:

```
count dq 0
```

- Define an array of 10 (`q`uadword) integers containing one through ten:

```
list dq 1,2,3,4,5,6,7,8,9,10
```

The `bss` Section

- Each declaration consists of a name, a type/size, but no value
- Define an uninitialized string (an array of `b`ytes):

```
firstName resb 50
```

- Define a single (`q`uadword) uninitialized integer:

```
age resq 1
```

- Define an uninitialized array of 10 (`q`uadword) integers:

```
grades resq 10
```

A Basic Program

CSCI 2050U - Computer Architecture

OntarioTech UNIVERSITY

A Basic Assembly Program

directives

comments

instructions

comments

```
section .data
message db "This is a message from Linux assembly!", 0ah, 0dh

section .text
global _start

; more system calls are provided in:
;http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64

_start:
mov rax, 1          ; syscall number for sys_write()
mov rdi, 1          ; standard output
mov rsi, message    ; what to print
mov rdx, 40         ; how many characters to print
syscall

; exit
mov rax, 60         ; syscall number for sys_exit()
mov rdi, 0          ; exit code (0 means success)
syscall
```

Directives

- Provide instructions to the assembler
- Typically don’t cause code to be generated
- Examples
  - `.section text` - tells the assembler the where the instructions are to be found
  - `.section data` - tells the assembler the where the data is to be found
  - `db` - tells the assembler to reserve space for an 8-bit byte/character value

Wrap-Up

- Assembly language programming
  - Development tools
  - Registers
  - Variables - the data section
  - A basic program

What is Next?

- System V ABI calling convention
- Basic input and output
  - Using the c library (aka libc)