UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 1

Taylor Johnson

Outline

- Administration
- Course Objectives
- Computer Organization Overview

Administration

- Overview
  - CSE2312 (Section 002)
  - Topic: Computer Organization and Assembly Language Programming
  - Time: T/R 2:00pm~3:20pm
  - Location: NH109
  - Website: http://www.taylortjohnson.com/class/cse2312/f14/
  - Instructor: Taylor Johnson
    - Office: ERB 559
    - Office Hours: 3:30pm~4:30pm and by appointment (email me to schedule taylor.johnson@uta.edu)
    - Background: Electrical/Computer Engineering (BSc, MSc, PhD)
    - Research: ensuring computer systems that interact with the physical world do what they're supposed to do (i.e., avoiding bugs)
  - GTA: Nathan Harvey
    - Office: TBA
    - Office Hours: TBA

Prerequisites and Materials

- Required courses / course credit
  - CSE1320: Intermediate Programming
  - CSE1310: Introduction to Computers and Programming
  - You should know:
    - How to program in at least one language
    - How to compile, execute, and debug programs
    - Elementary discrete math (binary, Boolean operations, etc.) and programs/algorithms
- Materials
  - Textbook: David A. Patterson and John L. Hennessy, Computer Organization and Design, Fifth Edition: The Hardware/Software Interface, Morgan Kaufmann, September 2013
  - More references on website

Syllabus Overview

- See website:
  http://www.taylortjohnson.com/class/cse2312/f14/
- Homework, programming assignments, slides, and other updates will appear on the website, so please ***CHECK OFTEN***

Expectations

- From course, instructor, and GTA:
  - Cover key issues and concepts in class
  - Homework
  - Programming assignments (projects)
    - May add more homeworks and remove some programming assignments
    - May have in class quizzes
  - Mid-term exam and final exam
- From you:
  - Come to class and to office hours if you need help
  - Read the textbook
  - Work through problems in textbook and homeworks
  - Do the programming assignments, start early
  - Ask questions (***ESSENTIAL***)

Administration Questions?

?

Outline

- Administration
- Course Objectives
- Computer Organization Overview

What is this Course About?

- This course is about one fundamental question in computer science and engineering
- You probably do not yet know the answer
- ***How do computers compute?***
- What does the computer actually do when you ask it to do something (i.e., run a program you've written)?

Topic of this Course

- Structured Computer Organization
  - Different levels of abstraction at which we can conceptualize a computer
  - Each level is useful for specific tasks, hiding useless details of lower levels

- Understanding some lower levels (hardware level, assembly instruction level)
  - In many applications, understanding these levels is necessary for writing effective code

Why Computer Organization

- At this point, you know how to program
- This course will teach you how your programs actually get executed
- There are several levels of translation between your code and the actual machine execution level
- Understanding these levels can help you write better code:
  - Example: understanding the role of memory cache

Why Computer Organization

- Understanding computer architecture will help you write code for different systems
- Right now, probably all your programs run on a desktop or laptop
- There are vast numbers of computers that are not desktops or laptops
    - Examples?

Why Computer Organization

- Understanding computer architecture will help you write code for different systems
- Right now, probably all your programs run on a desktop or laptop
- There are vast numbers of computers that are not desktops or laptops
  - Examples:
    - RFIDs
    - microcontrollers on radios, clocks, cars
    - cell phones/smart phones
    - music players

Course Objectives

- Answer the question: ***how do computer compute?***
- ***Understand*** computer components and levels (structure)
- Be able to write assembly programs to ***solve problems***
  - Many problems that need assembly are for getting systems started (bootloading), interacting with hardware (drivers), or performance
  - Write code for ARM processors and virtual machines (QEMU)
- Even if you don't think you'll ever do this again, it's important conceptual knowledge that you need to know

Assembly Language

- Assembly language is a programming language that is very low-level
  - Hard/painful for humans to use
  - Closer to the machine execution level
- Contains only simple instructions, that closely match the instructions that the hardware can execute in a single step

Why Assembly

- At the end of the day, all our code is converted to machine code

- Who does this conversion?

Why Assembly

- At the end of the day, all our code is converted to machine code
- Who does this conversion?
  - Compilers (and assemblers)
- While you will not learn how to write compilers in this class, understanding assembly is a prerequisite for being able to write a compiler

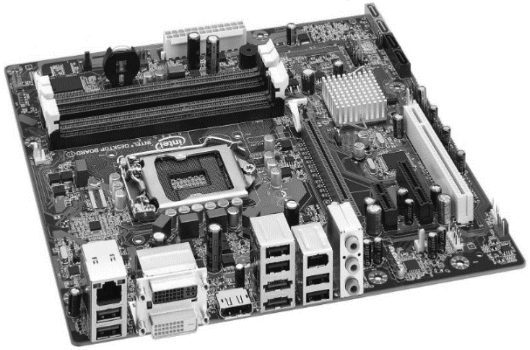
Why Assembly

- Understanding of assembly language can be useful for:
    - Optimizing code (when compiler optimization is not available or sufficient)
    - Designing virtual machines, that simulate hardware using software
    - Designing and programming various computer devices and peripherals

Outline

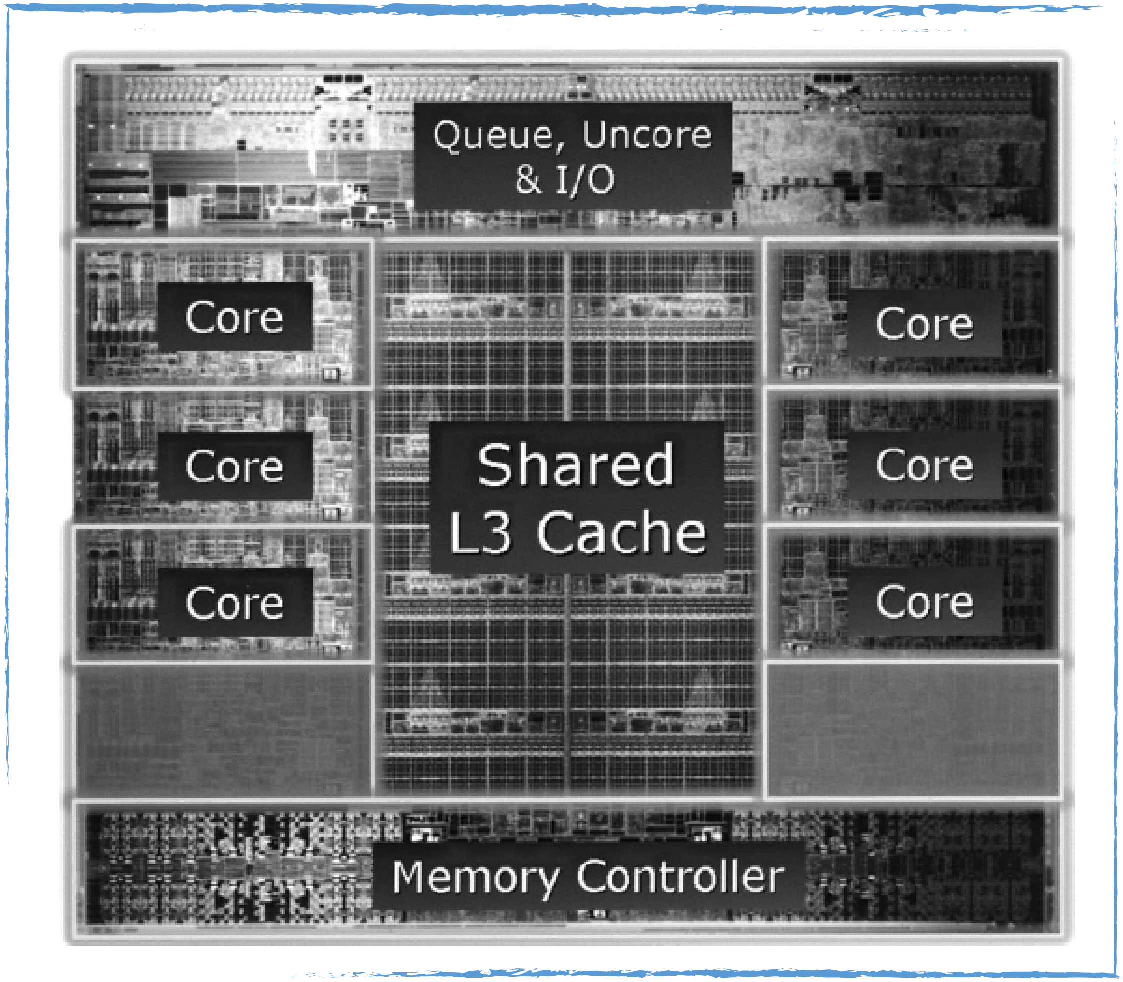
- Administration
- Course Objectives
- Computer Organization Overview

Digital Computers



- Machine for carrying out instructions
  - Program = sequence of instructions
- Instructions = primitive operations
  - Add numbers
  - Check if a number is zero
  - Copy data between different memory locations (addresses)
  - Represented as machine language (binary numbers of a certain length)
    - Example: $\overbrace{00}^{opcode}\ \overbrace{10}^{dest}\ \overbrace{01}^{src0}\ \overbrace{00}^{src1}$ on an 8-bit computer may mean:
      - Take numbers in registers 0 and 1 (special memory locations inside the processor) and add them together, putting their sum into register 2
      - That is, to this computer, 00100100 means $r2 = r1 + r0$
      - In assembly, this could be written: `add r2 r1 r0`
    - Question: for this same computer, what does 00000000 mean?
      - `add r0 r0 r0`, that is: $r0 = r0 + r0$

Computer Examples: Our PC/Laptop

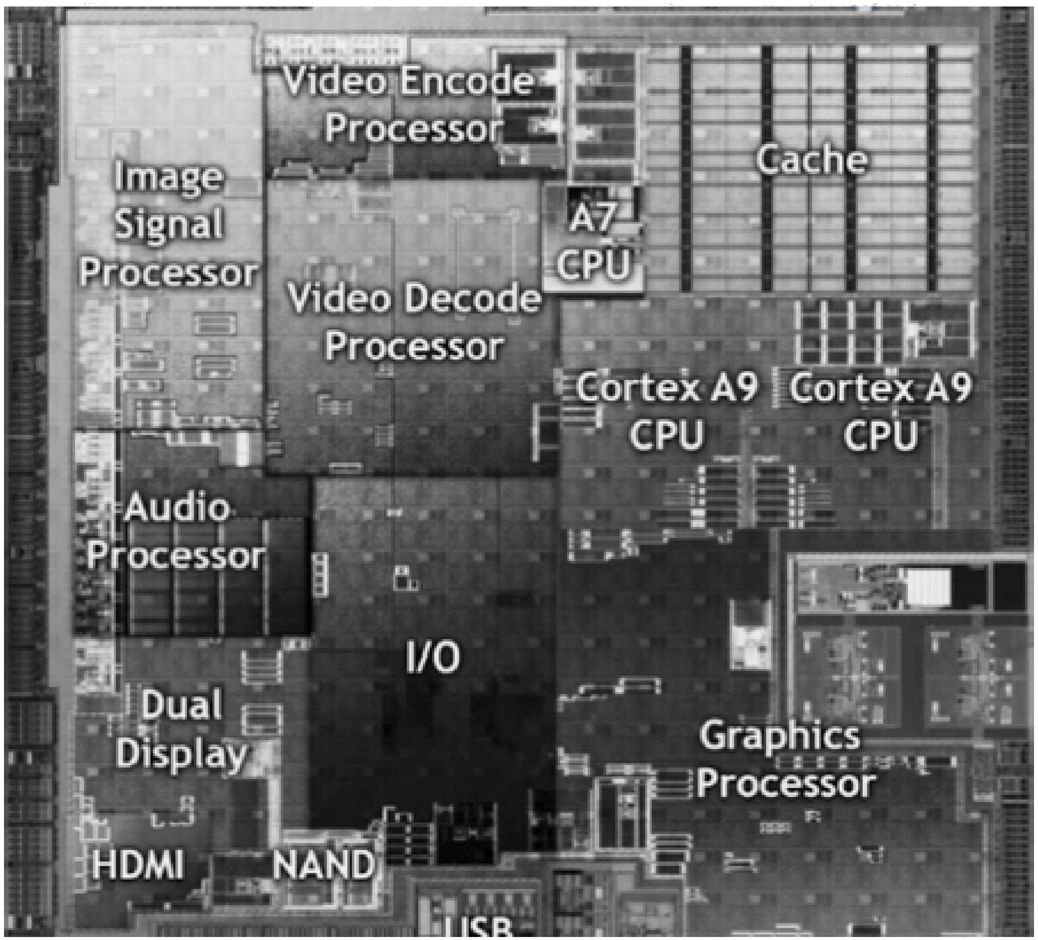


Intel Core i7-3960X die

The die is 21 by 21 mm and has 2.27 billion transistors

ISA: x86-64

Computer Examples: Our Phone



Nvidia Tegra 2 system on a chip (SoC)

ISA: ARM

Computer Examples: Our Server

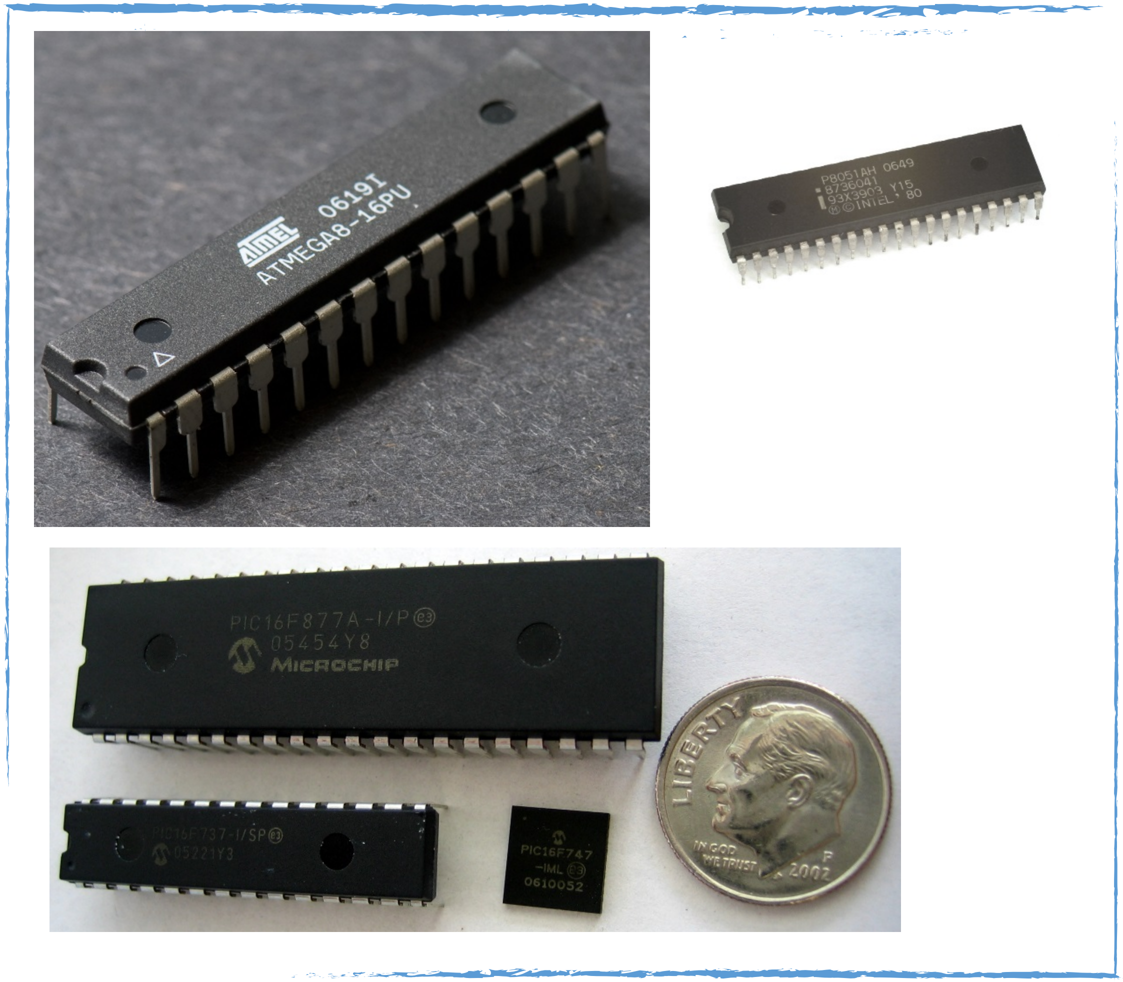
Oracle (Sun) SPARC T4

ISA: SPARC

Intel Xeon 7500

ISA: x86-64

Computer Examples: Our Car



Atmel ATmega

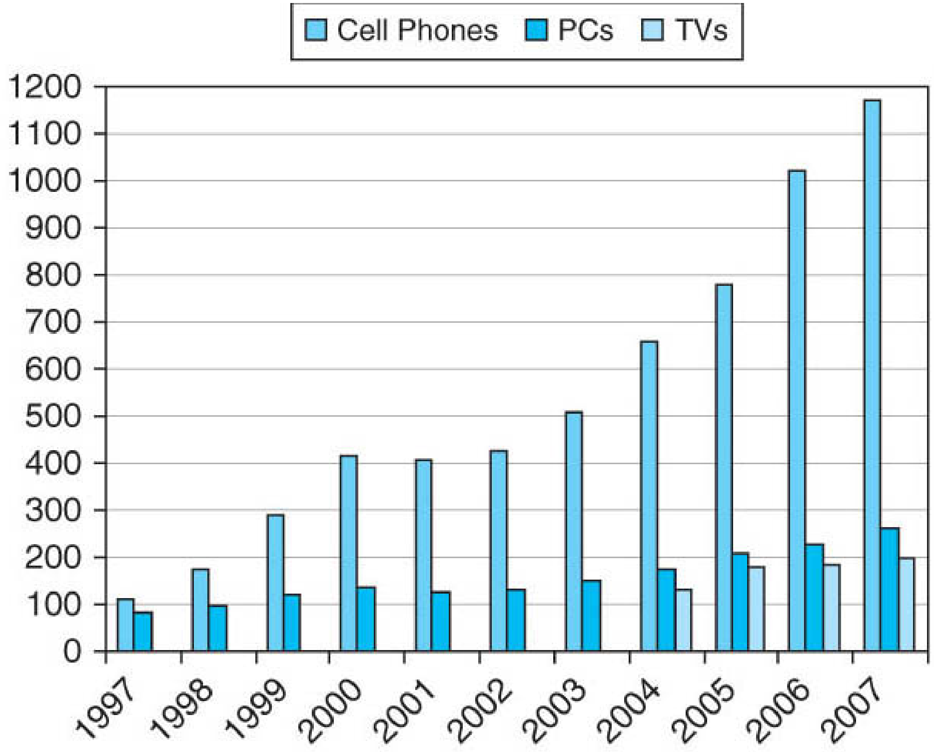
Microchip PIC

Intel 8051

Many others from TI, Cypress, etc.

Computer Examples: Others?

- What other examples can you come up with?
  - Moral: everywhere and in everything



Multilevel Computers

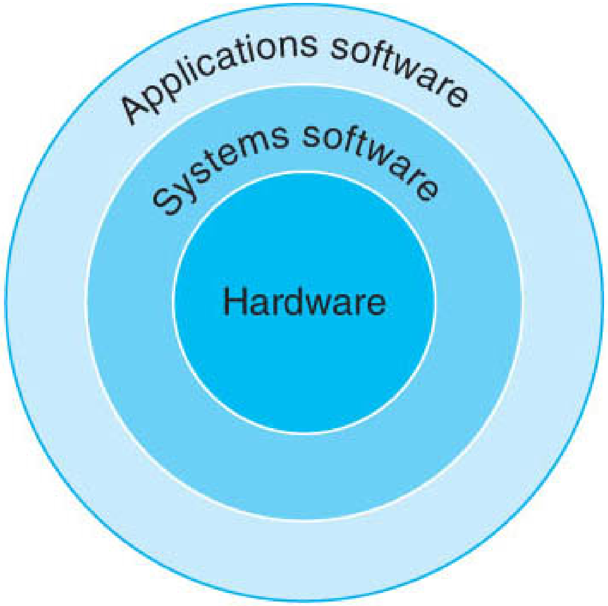
Level 3	Java Virtual Machine (Java Bytecode)
Level 2	QEMU (ARM)
Level 1	VMWare Virtual Machine (x86)
Level 0	Hardware (x86)

Multilevel Architectures

Level 4	Operating System Level	C / …
Level 3	Instruction Set Architecture (ISA) Level	Assembly / Machine Language
Level 2	Microarchitecture Level	n/a / Microcode
Level 1	Digital Logic Level	VHDL / Verilog
Level 0	Physical Device Level (Electronics)	n/a / Physics

Multiple Ways To Defining Levels

- Decomposition into these specific levels describes nicely some standard and widely used abstractions
- As you look closely, you may find that a specific level itself can be decomposed into more levels
- Examples:
  - A virtual machine
  - A compiler may go through two or three translation steps to produce the final compiled program. We can think of each of these steps as an intermediate level
  - A more sophisticated operating system (e.g., Windows) may run on top of a more simple operating system (e.g., DOS), adding extra capabilities (e.g., window management and graphical interfaces)



Virtual Machines (1)

- Some virtual machines only get implemented in software:
  - Why?
  - Examples?

Virtual Machines (1)

- Some virtual machines only get implemented in software:
  - Why? Because a hardware implementation is not cost-effective.
  - Examples? The virtual machines defined by C, Java, Python. They would be way more complex and expensive than typical hardware.
- Some virtual machines get first implemented in software, and then in hardware.
  - Why?

Virtual Machines (1)

- Some virtual machines only get implemented in software:
    - Why? Because a hardware implementation is not cost-effective.
    - Examples? The virtual machines defined by C, Java, Python. They would be way more complex and expensive than typical hardware.
- Some virtual machines get first implemented in software, and then in hardware.
    - Why? Because software implementations are much cheaper to make, and also much easier to test, debug, and modify.

Virtual Machines (2)

- Some virtual machines are used by the public in both hardware and software versions.
    - Why?

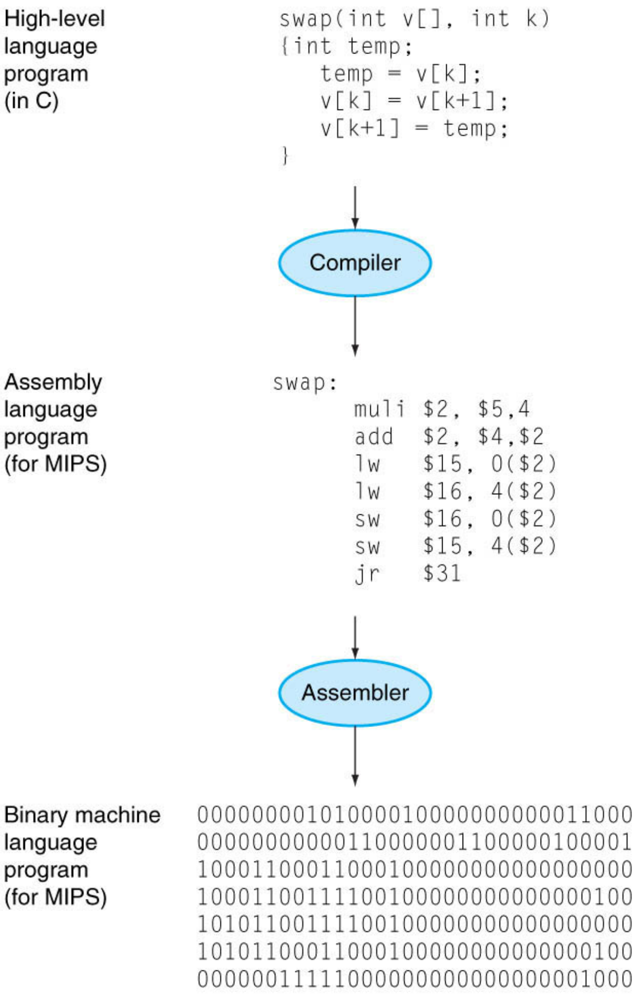
Virtual Machines (2)

- Some virtual machines are used by the public in both hardware and software versions.
    - Why? We may have a Macintosh computer, on which we want to run Windows software, or the other way around.

Compilation vs. Interpretation

- Compilation:
  - your n-level program is translated into a program at a lower level
  - the program at the lower level is stored in memory, and executed
  - while running, the lower-level program controls the computer
- Interpretation:
  - An interpreter, implemented at a lower level, executes your n-level program line-by-line
  - The interpreter translates each line into lower-level code, and executes that code
  - The interpreter is the program that is running, not your code

C program compiled into assembly language and then assembled into binary machine language



High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Assembly language program (for MIPS)

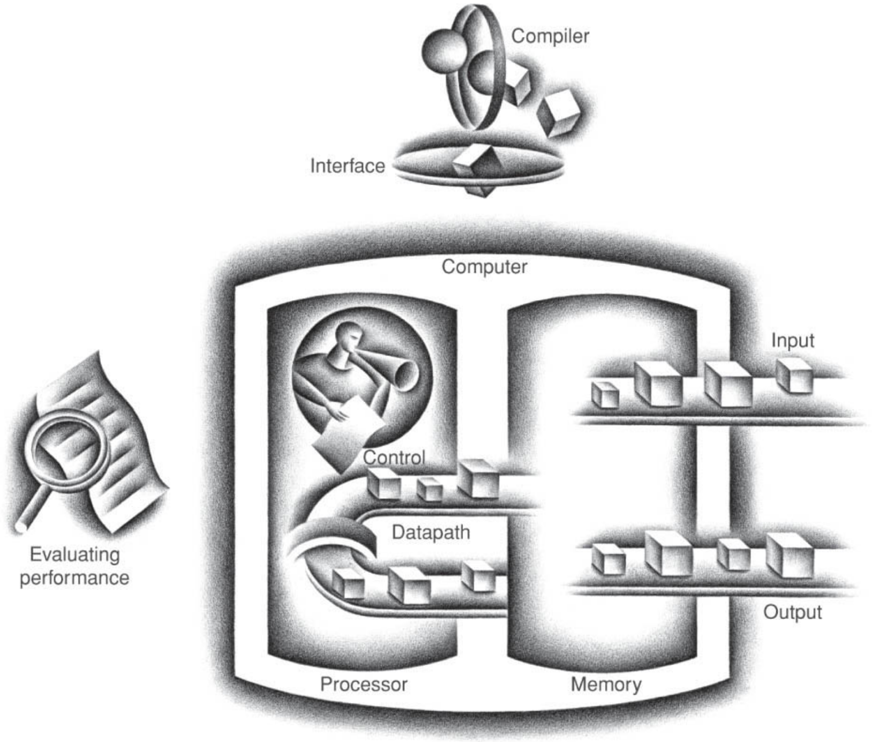
```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

System Overview

- CPU
  - Executes instructions
- Memory
  - Stores programs and data
- Buses
  - Transfers data
- Storage
  - Permanent
- I/O devices
  - Input: keypad, mouse, touch
  - Output: printer, screen
  - Both (input and output), such as:
    - USB, network, Wifi, touch screen, hard drive

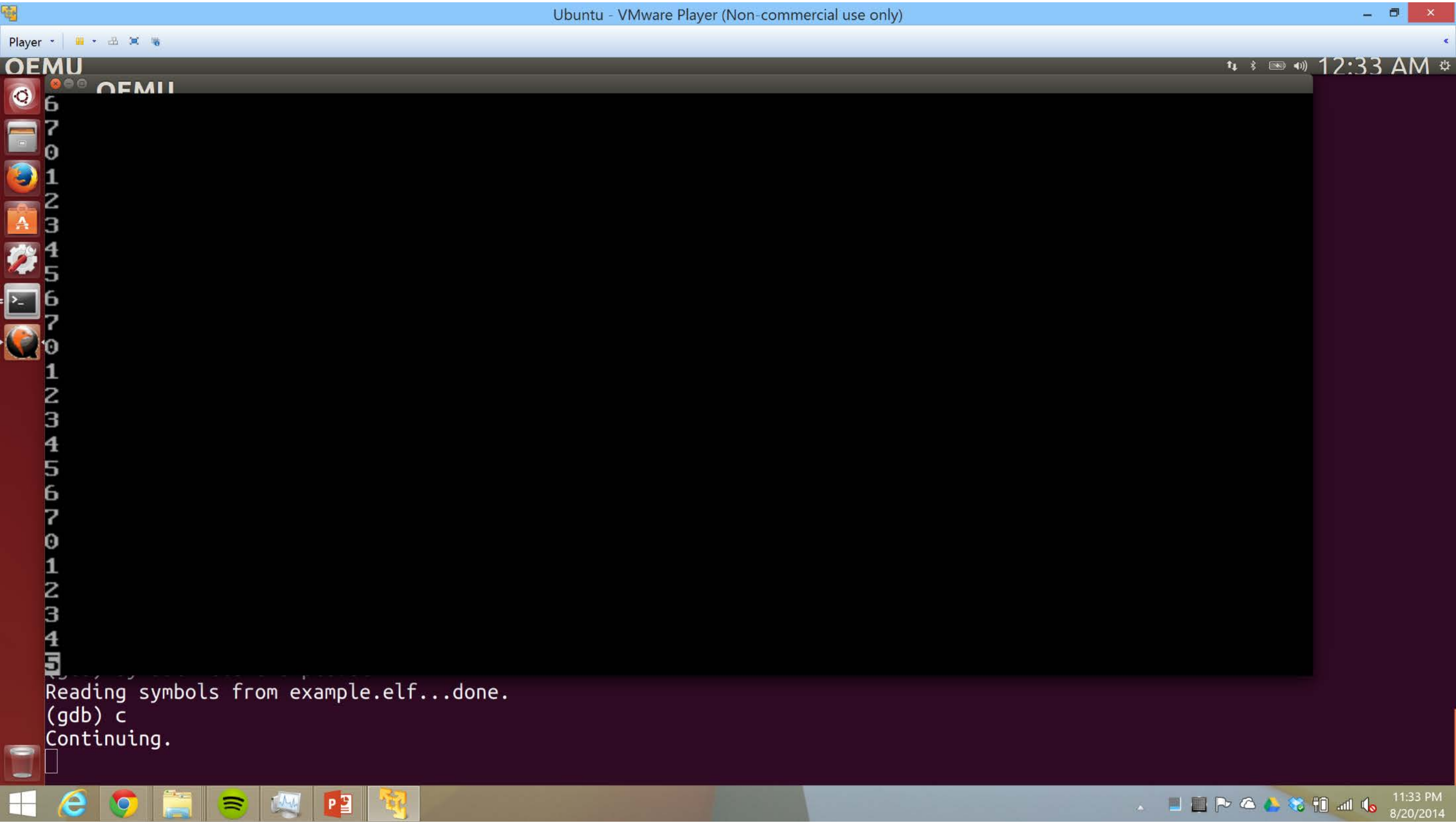


Instruction Set Architectures

- Interface between software and hardware
- High-level language to computer instructions
  - How do we translate from a high-level language (e.g., C, Python, Java) to instructions the computer can understand?
    - Compilation (translation before execution)
    - Interpretation (translation-on-the-fly during execution)
  - What are examples of each of these?

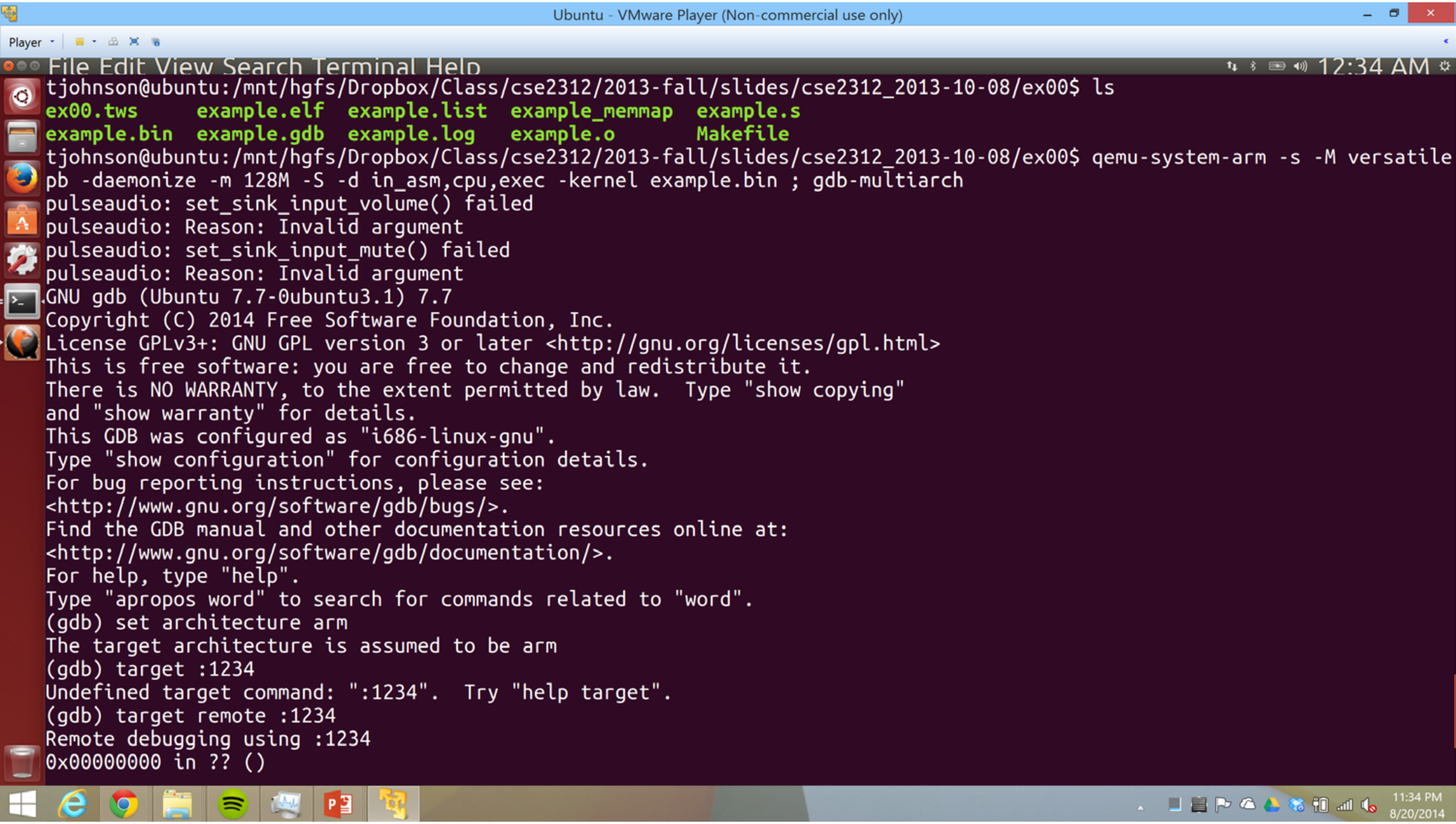
Demonstration

- VMWare, QEMU, and ARM ISA and gdb
- We will use QEMU and ARM later in this course
  - Particularly for programming assignments
- ARM versus x86
  - ARM is prevalent in embedded systems and handheld devices, many of which have more limited resources than your x86/x86-64 PC
  - Limited resources sometimes requires being very efficient (in space/memory or time/processing complexity)
  - Potentially greater need to interface with hardware

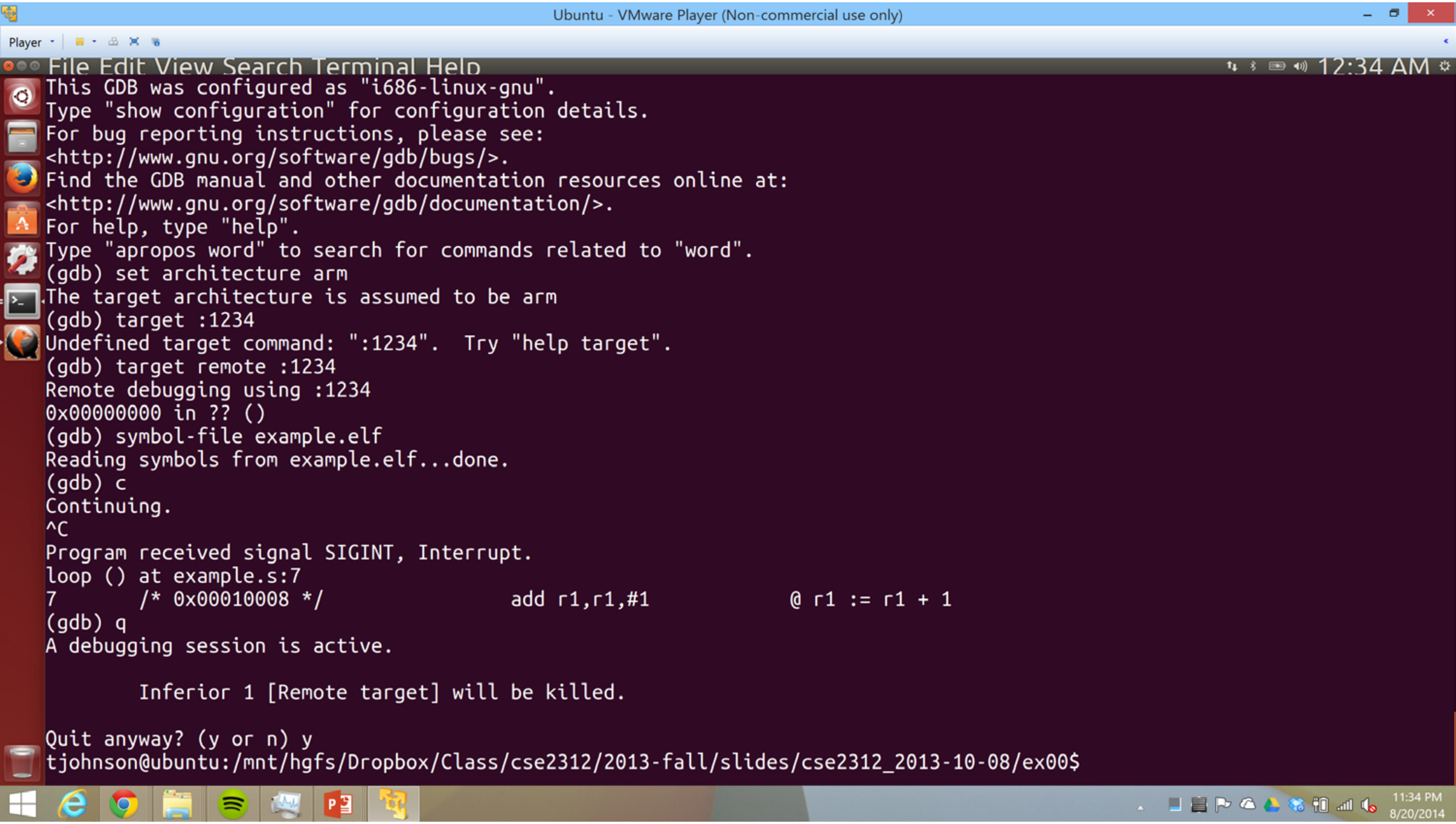


```
6
7
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
0
1
2
3
4
5
```

```
Reading symbols from example.elf...done.
(gdb) c
Continuing.
```



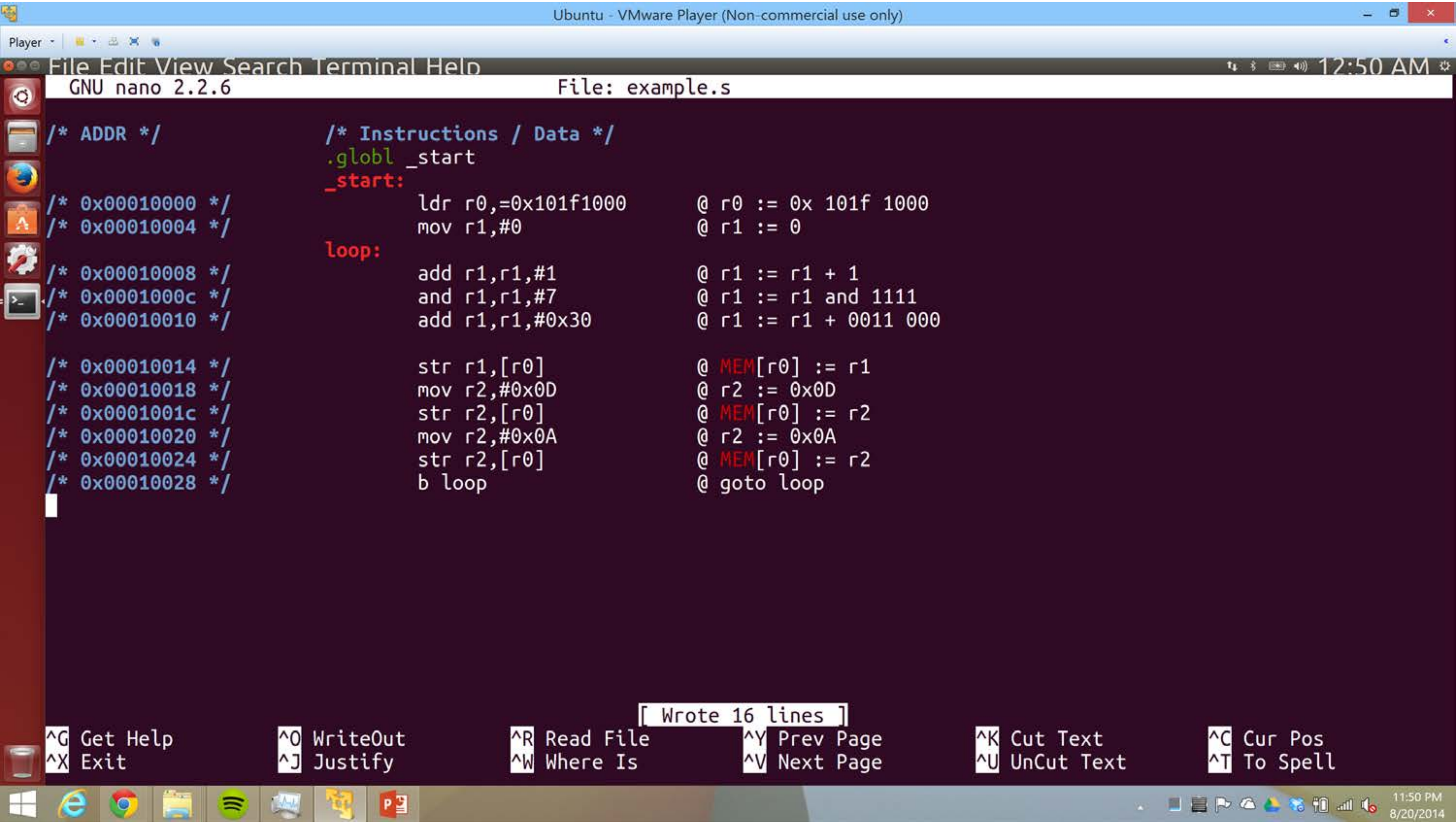
```
tjohnson@ubuntu:/mnt/hgfs/Dropbox/Class/cse2312/2013-fall/slides/cse2312_2013-10-08/ex00$ ls
ex00.tws      example.elf  example.list  example_memmap  example.s
example.bin   example.gdb  example.log   example.o       Makefile
tjohnson@ubuntu:/mnt/hgfs/Dropbox/Class/cse2312/2013-fall/slides/cse2312_2013-10-08/ex00$ qemu-system-arm -s -M versatile
pb -daemonize -m 128M -S -d in_asm,cpu,exec -kernel example.bin ; gdb-multiarch
pulseaudio: set_sink_input_volume() failed
pulseaudio: Reason: Invalid argument
pulseaudio: set_sink_input_mute() failed
pulseaudio: Reason: Invalid argument
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) set architecture arm
The target architecture is assumed to be arm
(gdb) target :1234
Undefined target command: ":1234".  Try "help target".
(gdb) target remote :1234
Remote debugging using :1234
0x00000000 in ?? ()
```



```
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) set architecture arm
The target architecture is assumed to be arm
(gdb) target :1234
Undefined target command: ":1234".  Try "help target".
(gdb) target remote :1234
Remote debugging using :1234
0x00000000 in ?? ()
(gdb) symbol-file example.elf
Reading symbols from example.elf...done.
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
loop () at example.s:7
7       /* 0x00010008 */                add r1,r1,#1            @ r1 := r1 + 1
(gdb) q
A debugging session is active.

        Inferior 1 [Remote target] will be killed.

Quit anyway? (y or n) y
tjohnson@ubuntu:/mnt/hgfs/Dropbox/Class/cse2312/2013-fall/slides/cse2312_2013-10-08/ex00$
```



```
  GNU nano 2.2.6                       File: example.s

/* ADDR */               /* Instructions / Data */
                         .globl _start
                         _start:
/* 0x00010000 */                ldr r0,=0x101f1000      @ r0 := 0x 101f 1000
/* 0x00010004 */                mov r1,#0               @ r1 := 0
                         loop:
/* 0x00010008 */                add r1,r1,#1            @ r1 := r1 + 1
/* 0x0001000c */                and r1,r1,#7            @ r1 := r1 and 1111
/* 0x00010010 */                add r1,r1,#0x30         @ r1 := r1 + 0011 000

/* 0x00010014 */                str r1,[r0]             @ MEM[r0] := r1
/* 0x00010018 */                mov r2,#0x0D            @ r2 := 0x0D
/* 0x0001001c */                str r2,[r0]             @ MEM[r0] := r2
/* 0x00010020 */                mov r2,#0x0A            @ r2 := 0x0A
/* 0x00010024 */                str r2,[r0]             @ MEM[r0] := r2
/* 0x00010028 */                b loop                  @ goto loop

                                             [ Wrote 16 lines ]
^G Get Help     ^O WriteOut     ^R Read File    ^Y Prev Page    ^K Cut Text     ^C Cur Pos
^X Exit         ^J Justify      ^W Where Is     ^V Next Page    ^U UnCut Text   ^T To Spell
```

Summary

- This course aims to answer the question: ***how do computers compute?***
- Complex and fundamental question
  - Organization of computer
  - Multilevel architectures

- Assembly programming
  - QEMU, ARM, gcc tools (as), and gdb (GNU debugger)

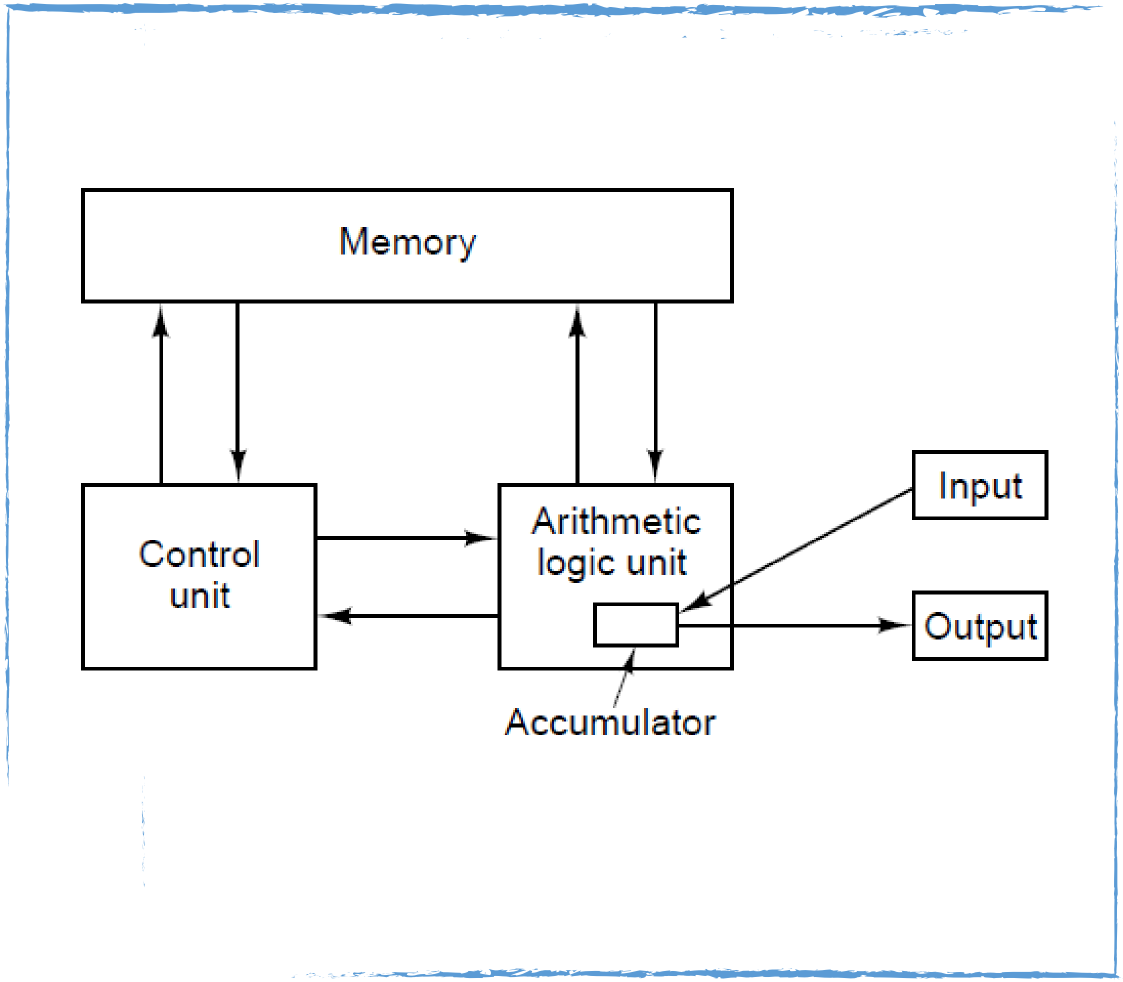
- Homework
  - Read chapter 1

  - Review binary arithmetic, Boolean operations, and representing numbers in binary

Questions?

?

Von Neumann Architecture



- Both data and program stored in memory
- Allows the computer to be “re-programmed”

Processor Overview