Ceng 111 – Fall 2015
Week 6a

The world of programming

Credit: Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.
The Reliability of Binary Representation

- Electronic devices are most reliable in a bistable environment
- Bistable environment
  - Distinguishing only two electronic states
    - Current flowing or not, or
    - Direction of flow
- Computers are bistable: hence binary representations
The Future
A SIMD Parallel Processing System

Previously on CENG 111!
Model of MIMD Parallel Processing
New Trend: Optical Computing

- Currently available for only data transfer.
- Work in progress towards “photonic logic” for designing circuits which use photons:
  - There are a lot of challenges and disadvantages

http://en.wikipedia.org/wiki/Optical_computing
There are several problems:

- Decoherence: The more the qubits, the more their effect on the environment.


Memory Types

- Dynamic Memory
  - The voltages stored in capacitors die away with time.
  - Solution?
    - Refresh the memory frequently; i.e., re-write the contents of the memory.

- Static Memory
  - A coupled transistor system stores the information.
  - One of the couples triggers the other.

- DRAM is cheaper and more widely used.

Memory Types

- **Volatile Memory:**
  - The stored values are lost when power is off.
  - DRAM, SRAM

- **Non-volatile:**
  - Read-only Memory (ROM), flash memory

Booting the Computer

- BIOS (Basic Input-Output System)
  - BIOS is a Read-Only Memory (ROM)
  - When loaded into the CPU, BIOS first initiates self-check of hardware (Power-on self test – POST)
  - Then, it goes over non-volatile storage devices to find something bootable.

- Boot Device & MBR
  - BIOS loads MBR from the boot device
  - MBR is not OS specific
  - MBR checks for a partition to boot
  - Loads OS (starts OS kernel)
Master Boot Record

- First 512-bytes of a partitioned data storage device; e.g., a disk.

- Responsibilities:
  - Holding the partition table
  - Helps booting of operating systems
  - Stores the ID of the disk (not used)

- A part of this 512-bytes is a code that searches the “primary” partitions and finds the one that is “active”. The “volume boot record” of the active partition boots the OS.
Interrupts

- When you type a character at the keyboard, the keyboard controller transmits the character to the CPU.
- However, the CPU is busy all the time and the keyboard controller has a limited storage; usually, only for one character.
- How to get the CPU’s attention?
- Polling vs interrupt.
  - Polling: a phone without a bell.

http://www.atarimagazines.com/compute/issue149/60_Interrupts_made_easy.php
Interrupts

- I/O, time management, power-off signals, traps, ...

When an interrupt is received,

- CPU stops the current task/program.
- It saves the current “context”
- It executes the code necessary for the received interrupt.
Timer Interrupt

- A system timer generates interrupts on every “tic”
- Called 18.2 times every second; once every 55ms.
  - Programmable
- Why is it needed?
  - Memory refreshing
  - OS time management
    - OS reads the initial time when booted
    - It increments the initial value with every time interrupt
<table>
<thead>
<tr>
<th>Interrupt vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>CPU: Executed after an attempt to divide by zero or when the quotient does not fit in the destination.</td>
</tr>
<tr>
<td>01h</td>
<td>CPU: Executed after every instruction while the trace flag is set</td>
</tr>
<tr>
<td>02h</td>
<td>CPU: NMI, used e.g. by POST for memory errors</td>
</tr>
<tr>
<td>03h</td>
<td>CPU: The lowest non-reserved interrupt, it is used exclusively for debugging, and the INT 03 handler is always implemented by a debugging program</td>
</tr>
<tr>
<td>04h</td>
<td>CPU: Numeric Overflow. Usually caused by the INTO instruction when the overflow flag is set.</td>
</tr>
<tr>
<td>05h</td>
<td>Executed when Shift-Print screen is pressed, as well as when the BOUND instruction detects a bound failure.</td>
</tr>
<tr>
<td>06h</td>
<td>CPU: Called when the Undefined Opcode (invalid instruction) exception occurs. Usually installed by the operating system.</td>
</tr>
<tr>
<td>07h</td>
<td>CPU: Called when an attempt was made to execute a floating-point instruction and no numeric coprocessor was available.</td>
</tr>
<tr>
<td>08h</td>
<td>IRQ0: Implemented by the system timing component; called 18.2 times per second (once every 55 ms) by the PIC</td>
</tr>
<tr>
<td>09h</td>
<td>IRQ1: Called after every key press and release (as well as during the time when a key is being held)</td>
</tr>
<tr>
<td>0Bh</td>
<td>IRQ3: Called by serial ports 2 and 4 (COM2/4) when in need of attention</td>
</tr>
<tr>
<td>0Ch</td>
<td>IRQ4: Called by serial ports 1 and 3 (COM1/3) when in need of attention</td>
</tr>
<tr>
<td>0Dh</td>
<td>IRQ5: Called by hard disk controller (PC/XT) or 2nd parallel port LPT2 (AT) when in need of attention</td>
</tr>
<tr>
<td>0Eh</td>
<td>IRQ6: Called by floppy disk controller when in need of attention</td>
</tr>
<tr>
<td>0Fh</td>
<td>IRQ7: Called by 1st parallel port LPT1 (printer) when in need of attention</td>
</tr>
</tbody>
</table>

Masking Interrupts

- Disabling interrupts for a limited time.

- When is it needed?
  - Busy, handling an important interrupt already
  - Being interrupted by the interrupt being handled.

Interrupts & Multi-tasking

- Most of the time, there are more than one programs/processes running.
- Each program runs for a fixed amount of time T.
  - T depends on the priority of the program/process.
- After T is finished, the CPU “interrupts”.

http://6004.csail.mit.edu/Spring98/Lectures/lect21/sld009.htm
Virtual Memory

- Memory management technique for “abstraction” over several RAM and disk storage devices.
- A program sees only “one RAM”; it does not care about the details.
- Not suitable for real-time systems.
- “Page”s are used in Virtual Memory.

http://en.wikipedia.org/wiki/Virtual_memory
Virtual Memory
- Paging -

- A page: at least 4KB of consecutive virtual memory addresses.
- An application program is stored in a set of pages.
- A page table maps the logical/virtual addresses of the pages with the physical addresses.

http://en.wikipedia.org/wiki/Virtual_memory
Direct Memory Access (DMA)

- All memory accesses go over the CPU.
  - CPU gets the virtual address,
  - It looks up the physical address,
  - It writes the data to the physical device (memory, in most cases)

- This slows down the CPU for simple and consecutive memory accesses.

- "Modern" computers have a channel for DMA.

- The CPU just initiates the DMA command, and a separate unit (DMA controller) handles the copy of data between a device and a memory.
Rotational Media as External Storage

Previously on CENG 41!
Rotational Drive Performance Characteristics

- For bigger disks, multiple layers & condensed layers are used.
- Access Time = Seek Time + Rotational Latency
  - Seek Time
    - “the time it takes the head to travel to the track of the disk where the data will be read or written”
    - around 9ms
  - Rotational Latency
    - “delay waiting for the rotation of the disk to bring the required disk sector under the read-write head.”
    - Depends on Rotations Per Minute (RPM)
- Data Transfer Rate
  - Covers the internal rate & the external rate
  - SATA: 3.0 Gbit/s

One Last Bit of Information Regarding Rotational Media

- Average Seek Time
  - x: track that we want to move the arm to.
  - y: current position of the arm.
  - How can we model it in terms of the number of tracks?
- Average Seek Time = average distance |
  $|x - y|$
  multiplied by the time to move the arm by one track.
- The average distance = $L/3$. How??

Average Distance =

$$\frac{1}{L^2} \int_{0}^{L} \int_{0}^{L} |x - y| \, dx \, dy = \frac{L}{3}$$
Transfer of Information In/On the Computer

- SATA – Serial Advanced Technology Attachment:
  - Bus interface between motherboard and storage devices, using high-speed cable.
  - Several versions (protocol & encoding is different):
    - SATA1: 1.5Gbit/s
    - SATA2: 3.0Gbit/s
    - SATA3: 6.0Gbit/s

- Alternatives:
  - IDE
  - PATA – Parallel Advanced Technology Attachment
Transfer of Information In/On the Computer

- Firewire – IEEE 1394 Interface
- High-Definition Audio-Video Network Alliance standard interface for audio/video communication.
- Several versions exist.
  - Around 800Mbit/s.
- Alternative:
  - Universal Serial Bus (i.e., USB)
Transfer of Information In/On the Computer

- Universal Serial Bus – USB
- Alternative to Firewire
  - Minor differences exist
- In addition to communication, USB bus can provide power to the device.
- Several versions exist: 1.0, 2.0 and 3.0.
  - 2.0 is the current widely used standard.
  - 3.0 is available but still not widely used.
Transfer of Information Between Computers

- **Ethernet**
  - For inter-net as well as intra-net.

- **Different versions:**
  - 100Mbit/s
  - 1Gbit/s

- **Ones that are not widely supported yet:**
  - 10Gbit/s
  - 100Gbit/s
Now

- System Software
- Assemblers and assembly language
- Operating systems
Types of System Software
The Spectrum of Programming Languages

<table>
<thead>
<tr>
<th>Machine language</th>
<th>Assembly language</th>
<th>Programming languages such as C++, Java</th>
<th>Pseudocode</th>
<th>English, Spanish, Japanese, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-level languages</td>
<td>(closely related to the hardware)</td>
<td>High-level languages</td>
<td>(more removed from details of the hardware)</td>
<td>Natural languages</td>
</tr>
</tbody>
</table>
Today

- The world of programming

Reminder:

- Tentative midterm date: 9 December at 17:40.
- THE1 Exam: 10 November at 17:40.
Now

- Program, Programming
- Programming Languages
- Programming Paradigms
- How to choose Programming Languages
- How to write programs
- How to test programs
- Finding errors in programs, i.e., debugging
Program, Programming

```c
int alice = 1;
int bob = 456;
int carol;
main(void)
{
    carol = alice * bob;
    printf("%d", carol);
}
```
Program, Programming

- **Program:**
  - “a series of steps to be carried out or goals to be accomplished”

- A recipe for cooking a certain dish is also a program (but not a computer program).
The Translation/Loading/Execution Process
Assembly Language (continued)

- Source program
  - An assembly language program

- Object program
  - A machine language program

- Assembler
  - Translates a source program into a corresponding object program
Assembly Language (continued)

Advantages of writing in assembly language rather than machine language

- Use of symbolic operation codes rather than numeric (binary) ones
- Use of symbolic memory addresses rather than numeric (binary) ones
- Pseudo-operations that provide useful user-oriented services such as data generation
.BEGIN    --This must be the first line of the program.

::    --Assembly language instructions like those in Figure 6.5.

HALT    --This instruction terminates execution of the program

::    --Data generation pseudo-ops such as 

::.DATA are placed here, after the HALT.

.END    --This must be the last line of the program.

Structure of a Typical Assembly Language Program
Examples of Assembly Language Code

- Arithmetic expression

\[ A = B + C - 7 \]

(Assume that B and C have already been assigned values)
Examples of Assembly Language Code (continued)

- Assembly language translation

```
LOAD B -- Put the value B into register R.
ADD C -- R now holds the sum (B + C).
SUBTRACT SEVEN -- R now holds the expression (B + C - 7).
STORE A -- Store the result into A.

:                     -- These data should be placed after the HALT.
:                     
A: .DATA 0
B: .DATA 0
C: .DATA 0
SEVEN: .DATA 7 -- The constant 7.
```
Examples of Assembly Language Code (continued)

Problem

- Read in a sequence of non-negative numbers, one number at a time, and compute their sum

- When you encounter a negative number, print out the sum of the non-negative values and stop
<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Set the value of Sum to 0</td>
</tr>
<tr>
<td>2</td>
<td>Input the first number $N$</td>
</tr>
<tr>
<td>3</td>
<td>While $N$ is not negative do</td>
</tr>
<tr>
<td>4</td>
<td>Add the value of $N$ to Sum</td>
</tr>
<tr>
<td>5</td>
<td>Input the next data value $N$</td>
</tr>
<tr>
<td>6</td>
<td>End of the loop</td>
</tr>
<tr>
<td>7</td>
<td>Print out Sum</td>
</tr>
<tr>
<td>8</td>
<td>Stop</td>
</tr>
</tbody>
</table>

Algorithm to Compute the Sum of Numbers
Assembly Language Program to Compute the Sum of Nonnegative Numbers

```
.BEGIN --This marks the start of the program.
CLEAR SUM --Set the running sum to 0 (line 1).
IN N --Input the first number N (line 2).

--The next three instructions test whether N is a negative number (line 3).
AGAIN: LOAD ZERO --Put 0 into register R.
COMPARE N --Compare N and 0.
JUMPLT NEG --Go to NEG if N < 0.

--We get here if N ≥ 0. We add N to the running sum (line 4).
LOAD SUM --Put SUM into R.
ADD N --Add N. R now holds (N + SUM).
STORE SUM --Put the result back into SUM.

--Get the next input value (line 5).
IN N

--Now go back and repeat the loop (line 6).
JUMP AGAIN

--We get to this section of the program only when we encounter a negative value.
NEG: OUT SUM --Print the sum (line 7)
HALT --and stop (line 8).

--Here are the data generation pseudo-ops
SUM: .DATA 0 --The running sum goes here.
N: .DATA 0 --The input data are placed here.
ZERO: .DATA 0 --The constant 0.

--Now we mark the end of the entire program.
.END
```
Translation and Loading

- Before a source program can be run, an assembler and a loader must be invoked

- Assembler
  - Translates a symbolic assembly language program into machine language

- Loader
  - Reads instructions from the object file and stores them into memory for execution
Translation and Loading (continued)

- Assembler tasks
  - Convert symbolic op codes to binary
  - Convert symbolic addresses to binary
  - Perform assembler services requested by the pseudo-ops
  - Put translated instructions into a file for future use
Programming Languages

C code for the example problem:

```c
int alice = 123;
int bob = 456;
int carol;
main(void)
{
    carol = alice*bob;
}
```
Programming Languages

Assembly code for the example problem:

```assembly
main:
    pushq  %rbp
    movq   %rsp, %rbp
    movl   alice(%rip), %edx
    movl   bob(%rip), %eax
    imull  %edx, %eax
    movl   %eax, carol(%rip)
    movl   $0, %eax
    leave
    ret

 alice:
    .long  123

 bob:
    .long  456
```
Programming Languages

Machine code for a simple problem/program:

01010101 01001000 10001001 11100101 10001011 00010101 10110010 00000011
00100000 00000000 10001011 00000101 10110000 00000011 00100000 00000000
00001111 10101111 11000010 10001001 00000101 10111011 00000011 00100000
00000000 10111000 00000000 00000000 00000000 00000000 11001001 11000011
...
11001000 00000001 00000000 00000000 00000000 00000000
main:
    pushq  %rbp
    movq   %rsp, %rbp
    movl   alice(%rip), %edx
    movl   bob(%rip), %eax
    imull  %edx, %eax
    movl   %eax, carol(%rip)
    movl   $0, %eax
    leave
    ret

alice:
    .long  123

bob:
    .long  456

int alice = 123;
int bob = 456;
int carol;
main(void)
{
    carol = alice * bob;
}
There is a limit to how high a language can get.
Why can’t we write programs in our spoken language?
Programming Language Paradigms

Classification / Categorization of programming languages.

- Imperative Paradigm
- Functional Paradigm
- Logical-declarative Paradigm
- Object-oriented Paradigm
- Concurrent Paradigm
- Event-driven Paradigm
Imperative Paradigm

Statement_1
Statement_2
Statement_3
Statement_4
Statement_5

From C:

```c
int a = 2;
int b = a * 2;
int c;

c = -b - sqrt(b*b - 4*a*c) / (2*a);
```
Functional Paradigm

- Data environment is restricted.
- Functions receive their inputs and return their results to the data environment.
- Programmer’s task:
  - decompose the problem into a set of functions such that the composition of these functions produce the desired result.
Functional Paradigm

Imperative Version of Fibonacci Numbers

```python
# Fibonacci numbers, imperative style
N=10
first = 0  # seed value fibonacci(0)
second = 1  # seed value fibonacci(1)
fib_number = first + second  # calculate fibonacci
for position in range(N-2):  # iterate N-2 times
    first = second  # update the values
    second = fib_number
    fib_number = first + second  # update the result
print fib_number
```

Functional Version of Fibonacci Numbers

```python
# Fibonacci numbers, functional style
def fibonacci(N):  # Fibonacci number N (for N >= 0)
    if N <= 1: return N  # base cases
    else: return fibonacci(N-1) + fibonacci(N-2)

print fibonacci(10)
```
Logical-declarative Paradigm

- The data and the relations are states as rules, or facts.
- The problem is solved by writing new rules/facts.

From Prolog:

mother(matilda,ruth).
mother(trudi,paggy).
mother(eve,alice).
mother(zoe,sue).
mother(eve,trudi).
mother(matilda,eve).
mother(eve,carol).
grandma(X,Y) :- mother(X,Z), mother(Z,Y).
Object Oriented Paradigm

CAR Class

Data
- Wheels: Wheel_1, Wheel_2, ..., Wheel_N
- Gear: Manual, Automatic, Triptronic
- Steering: Hydraulic, Electrical, Mechanical
- Doors: Door_1, Door_2, ..., Door_M
- Size: Width, Height
- Fuel: Gasoline, Diesel, LPG
- Engine: K Valves, L Cylinders

Actions
- Steer
- Brake
- Speed up

CAR Objects - Instances of CAR Class
- Car Object 1
- Car Object 2
- Car Object 3
- Car Object 4
- Car Object 5
- Car Object 6
Object Oriented Paradigm

- Problem is decomposed into objects which hold data and the corresponding functions on the data.

- Objects can be defined using other objects as a basis; the new object inherits from the basis objects.

From C++:

```cpp
class Item {
    string Name;
    float Price;
    string Location;
    ...
};

class Book : Item {
    string Author;
    string Publisher;
    ...
};

class MusicCD : Item {
    string Artist;
    string Distributor;
    ...
};
```