CPSC 2310 Exam 3 Study Guide

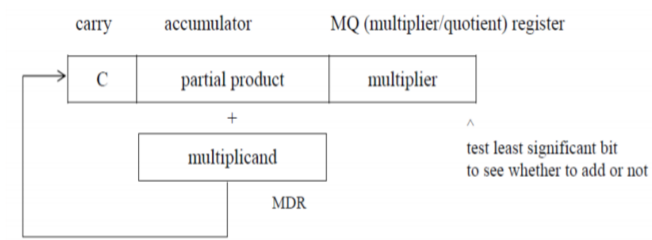
Brendan McGuire (bmmcgui@clemson.edu)

Good luck all on the exam! Go crush it!

Binary Multiplication

In order to multiply numbers together in registers more efficiently, a specific hardware configuration is used. There are a number of components to this system, as shown below. Since multiplication is just repeated addition, we will just add one number (the multiplicand) to itself a number (the multiplier) times.

Binary Multiplication is more complicated than Binary Addition, often requiring 8-10 steps to complete. Modern chips have been able to reduce that to 3-5 steps.



Carry — *Single bit flag that indicates a carry needs to be performed. Initially 0.*

Accumulator — *Register that holds the intermediate result of the computation. Initially set to 0.*

Multiplier — *Also known as Q, the number that indicates the number of additions that are still required.*

Multiplicand — *Also known as M, the number that gets repeatedly added to determine the computation.*

Algorithm

Set the carry flag and the accumulator to 0

Repeat this for each bit in the multiplier:

If the Least Significant Bit of Q is one, then add M to the accumulator

Shift ACC + Multiplier right 1 place

The value in the accumulator is the result of the multiplication

Worked Example

Let's consider the example for 11 (1011) multiplied by 15 (1111). We will set 11 as the multiplier and 15 as the multiplicand.

```
Step 1:
C Accumulator Multiplier
0        0000       1011
     +   1111
0        1111
   >>
0        0111       1101
```

The least significant bit is 1, so add the multiplicand to the accumulator

Right shift the carry, accumulator and multiplier by 1. Notice how the rightmost bit in the accumulator shifts into the multiplier.

```
========================

Step 2:
C Accumulator Multiplier
0        0111       1101
     +   1111
1        0110
  >>
0        1011       0110
```

The least significant bit is 1, so add the multiplicand to the accumulator. The addition overflows, so set the carry bit

Right shift everything! Notice how the carry bit shifts into the accumulator, and the zero at the end of the accumulator shifts into the multiplier

```
========================

Step 3:
C Accumulator Multiplier
0        1011       0110
     +   0000
0        1011
  >>
0        0101       1011
```

The least significant bit is zero, so do not add the multiplicand.

Right shift everything!

```
========================

Step 4:
C Accumulator Multiplier
0        0101       1011
     +   1111
1        0100
  >>
0        1010       0101

Result: 1010 0101
Verify:
128 + 32 + 4 + 1 = 165
         11 * 15 = 165
```

The least significant bit is 1, so add the multiplicand to the accumulator. The addition overflows, so set the carry bit

Right shift everything! We now have our result, the accumulator and the multiplier in succession. Verify the results are correct!

Integer Division

When dividing integers, remember that it is not possible to store a fractional result in an integer. This means that the results will get rounded, by a number of different rules. Remember that there are different types of rounding, as illustrated below. Note that *Round Down* and *Round Away From Zero* are only different for negative integers. Similarly, *Round Up* and *Round Away From Zero* are the same for positive numbers, and different for negative integers

Original Value	**Round Down** (towards -∞)	**Round Up** (towards ∞)	**Round Towards Zero**	**Round Away From Zero**
+23.67	+23	+24	+23	+24
+23.00	+23	+23	+23	+23
0	0	0	0	0
-23.00	-23	-23	-23	-23
-23.67	-24	-23	-23	-24

Division By A Power of Two

You can use the right shift operator to do this. If the number is signed, use an arithmetic right shift, and if the number is unsigned, use a logical right shift. Division is even more difficult for the CPU, often requiring 30+ CPU steps.

<u>**Unsigned Division**</u>

To divide c / 2^k perform a logical right shift on c, k times. **The result is rounded towards zero!**

```
uint8_t c = 0b01010001; // 81
uint8_t k = 3; // 2^3 = 8

// 01010001 >> 3 = 00001010
// 81 / 8 = 10.12500 → 10
uint8_t result = c >> k; // 10
```

<u>**Signed Division**</u>

To divide c / 2^k perform an arithmetic right shift on c, k times. **The result is rounded down!** This can have unexpected consequences for negatives

```
uint8_t c = 0b01010001; // 81
uint8_t k = 3; // 2^3 = 8

// 01010001 >> 3 = 00001010
// 81 / 8 = 10.12500 → 10
uint8_t result = c >> k; // 10

int8_t c = 0b10101111; // -81
int8_t k = 3; // 2^3 = 8

// 10101111 >> 3 = 11110101
// -81 / 8 = -10.12500 → -11
int8_t result = c >> k; // -11
```

<u>**Signed Division Bias**</u>

In order to reduce rounding errors when doing two's complement division, we can *bias* the value by adding `(1<<k) - 1` before performing the right shift (where k is exponent of the power of 2). **This will cause the signed division to round towards zero!** <u>That is helpful for negative numbers only!</u>

Floating Point Numbers

Of course, computers need to do more than just handle integers. Being able to represent fractional numbers is also incredibly important!

History

1970s — Floating point numbers were all handled differently by each manufacturer
1976 — Intel hires Dr. William Kahan to develop the floating point standard. Kahan worked with IEEE to develop IEEE 754, which is now the universal standard for floating point representation.

Fractional Binary Numbers

The basis of IEEE 754 is fractional binary numbers. Have all numbers to the right of the binary point (equivalent to the decimal place) be negative powers of two

2^6	2^5	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
0	1	0	1	0	1	0	.	1	0	1	0	0	1

```
2^5  + 2^3   + 2^1 + 2^-1 + 2^-3 + 2^-6 =
32   + 8     + 2   + 1/2  + 1/8  + 1/64 =
42 41/64 = 42.640625
```

Limitations of Fractional Binary Numbers

In the same way that fractional decimal numbers cannot perfectly represent all fractions, fractional binary numbers can't either. They can only represent numbers **that are multiples of a power of 2**. All other numbers must be approximated.

We can get arbitrarily close to the actual number by increasing the number of bits we use. For example, lets try to represent 0.20 in binary.

Binary Representation	Fractional Equivalent	Decimal Equivalent
0.0	0/2	0.0
0.01	1/4	0.25
0.0011	3/16	0.1875
0.001101	13/64	0.203125
0.00110011	51/256	0.19921875

As we increase the number bits, we are getting closer and closer, but we will never reach the value exactly!

IEEE 754 Standard

The standard builds on the idea of fractional binary numbers, but instead of actually representing all of those, it uses **scientific notation**, storing a *sign bit, mantissa, and an exponent* in the form

```
-1^(Sign) + Mantissa * 2^Exponent
```

Single Precision VS Double Precision

As we saw above, more bits allow us to represent numbers more exactly. So the IEEE 754 standard has two floating-point numbers: single-precision (32-bit) and double-precision (64-bit) to allow us to make this tradeoff.

Single Precision (32-Bit)	*Double Precision (64-Bit)*
1 sign bit 8 exponent bits 23 mantissa bits Bias of 127	1 sign bit 11 exponent bits 52 mantissa bits Bias of 1023
`float` in C	`double` in C

Converting to IEEE 754

The process of converting a decimal number into IEEE 754 is fairly simple, but there are a few gotchas you should keep in mind. An example is shown below, using single precision (32-bits)

Convert -187.69 to binary.

Convert the whole number component to binary (ignore the sign). Take note of the number of bits required to store the whole number component, and use only what you need to. This means the MSB should be a 1!

```
187 = 128 + 32 + 16 + 8 + 2 + 1 = 1011 1011
```

Convert the fractional part to binary. Repeatedly multiply the number by 2, but only preserve the fractional part. Because the whole number component took up 8 bits, we can stop after 23 - 8 + 1 = 16 operations.

```
0.69 * 2 = 1.38
0.38 * 2 = 0.76
0.76 * 2 = 1.52
0.52 * 2 = 1.04
0.04 * 2 = 0.08
0.08 * 2 = 0.16
0.16 * 2 = 0.32
0.32 * 2 = 0.64
0.64 * 2 = 1.28
```

```
0.28 * 2 = 0.56
0.56 * 2 = 1.12
0.12 * 2 = 0.24
0.24 * 2 = 0.48
0.48 * 2 = 0.96
0.96 * 2 = 1.92
0.92 * 2 = 1.84
```

Read the whole number of the resultant numbers down to get the fractional binary number

```
0.69 ≈ 0.1011000010100011
```

Combine this with the whole number we found above to get the entire binary representation

```
187.69 ≈ 1011 1011.1011 0000 1010 0011
```

Convert to scientific notation. Shift the binary point all the way to the front, and count the number of moves. This will be the exponent.

$1011\ 1011.1011\ 0000\ 1010\ 0011 = 1.011\ 1011\ 1011\ 0000\ 1010\ 0011 * 2^{7}$

Convert this number to IEEE 754 representation. The sign bit should be 1 if the number is negative. The exponent should be the number we got above, plus the bias (for single precision, this is 127). The mantissa should be the fractional component of the scientific notation number we got earlier.

```
Sign: 1
Exponent: 7 + 127 = 134 = 1000 0110
Mantissa: 011 1011 1011 0000 1010 0011
```

Combine that representation to form the binary representation of the number.

```
S   Exponent  Mantissa
1  1000 0110  011 1011 1011 0000 1010 0011
```

So *-187.69 in binary is* `1100 0011 0011 1011 1011 0000 1010 0011`

Converting from IEEE 754
This largely follows the same practice, but in reverse.

Convert `1100 0011 0011 1011 1011 0000 1010 0011` to decimal.

Split up the binary number into its components. We know the first bit is the sign bit, followed by the 8 bits for the exponent, followed by the 23 bits of the mantissa.

```
Sign: 1
Exponent: 1000 0110
Mantissa: 011 1011 1011 0000 1010 0011
```

Remove the bias from the exponent

```
1000 0110 - 0111 1111 = 0000 0111
      134 -       127 =         7
```

Form the number into scientific notation using the below formula

$(-1)^{sign} * (1 + mantissa) * 2^{exponent}$

$-1^{1} * (1.011\ 1011\ 1011\ 0000\ 1010\ 0011) * 2^{7}$

Shift the number to remove the exponent

$(1.011\ 1011\ 1011\ 0000\ 1010\ 0011) * 2^{7} = 1011\ 1011.1011\ 0000\ 1010\ 0011$

Evaluate the bits at each position, and apply the sign bit

```
1011 1011.1011 0000 1010 0011
```

$2^{7} + 2^{5} + 2^{4} + 2^{3} + 2^{1} + 2^{0} + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-9} + 2^{-11} + 2^{-15} + 2^{-16}$
$= -187.6899871826171875$

Notice this is very very close, but not exactly the value we had.

Special Cases
In addition to the standard floating point numbers, there are a few different special cases of IEEE 754.

1. **Normalized Numbers** are just the numbers we saw above. Normalized numbers have an exponent which is not 0 AND not 255 (basically, not all zeros, and not all 1s)
2. **Denormalized Numbers** have an exponent value of all zeros. The effective exponent is equal to 1 - bias. They allow us to represent +0, -0; they allow us to represent values very close to zero by using a property called **gradual underflow**
3. **Not A Number Values** have exponent values of all 1s. Infinity is represented by a mantissa field of all zeros (can be signed as well!). All other values are NaN, and usually indicate uninitialized data.

Assembly

Before the invention of higher-level programming languages, programmers need to write code in assembly. While there isn't much use in this anymore, it can still be important to understand how assembly works, and understand the assembly language generated by compilers.

Abridged History

Processor	Year	Number of Transistors	Description
8086	1978	29,000	One of the first single-chip, 16bit microprocessors
i386	1985	275,000	Expanded to 32-bit architecture, first model to fully support UNIX
Pentium	1993	3,100,000	Improved performance without many changes to the instruction set
Pentium/MMX	1997	4,500,000	Add support for operations on vectors of integers
Pentium 4E	2004	125,000,000	Added support for hyperthreading, a means of executing multiple programs at once
Core i7 Haswell	2013	1,400,000,000	Improved performance and increased capabilities, specifically to do with 256 bit vectors

As technology progressed, it became possible to place successively more transistors into a smaller package, allowing for greater performance. Additionally, new features were added to the x86 architecture to make it more efficient and computing common problems.

Machine-Level Abstractions

While assembly is closer to the hardware, the are still a number of important abstractions that are important at this level:

1. The *Instruction Set Architecture* defines the possible operations a processor can perform, the format, and the effect each will have
2. Memory Addresses in programs are virtual addresses and not representative of the "actual" hardware address.

Producing Readable Assembly

Compilers will apply a lot of optimizations to make your code run more efficiently, but this can be very hard for humans to understand.

- ➔ Use **-Og** to disable the optimizations to those that preserve the structure of the program
- ➔ Use **-S** to stop just before the assembler, outputting an assembly program
- ➔ Use **-c** to stop just before the linker (.s), producing an object file (.o)

Diassassemblers

To go the other way, and produce assembly from a compiled object file, we can use a disassembler, like objdump, to translate the machine code into assembly.

To see this in action, let's compile the following simple C program, using `-O0` to remove compiler optimizations, and `-c` to output the object file

```
#include <stdio.h>

int main() {

    int side = 16;
    int area = side * side;
    int weight = 96;
    int density = area / weight;
    printf("density: %d", density);
};
```

The output of objdump -d density.o is shown below. *Notice how instructions range from 1-15 bytes in size.*

```
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 10             sub    $0x10,%rsp
   c:   c7 45 f0 10 00 00 00    movl   $0x10,-0x10(%rbp)
  13:   8b 45 f0                mov    -0x10(%rbp),%eax
  16:   0f af c0                imul   %eax,%eax
  19:   89 45 f4                mov    %eax,-0xc(%rbp)
  1c:   c7 45 f8 60 00 00 00    movl   $0x60,-0x8(%rbp)
  23:   8b 45 f4                mov    -0xc(%rbp),%eax
  26:   99                      cltd
  27:   f7 7d f8                idivl  -0x8(%rbp)
  2a:   89 45 fc                mov    %eax,-0x4(%rbp)
  2d:   8b 45 fc                mov    -0x4(%rbp),%eax
  30:   89 c6                   mov    %eax,%esi
  32:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 39 <main+0x39>
  39:   b8 00 00 00 00          mov    $0x0,%eax
  3e:   e8 00 00 00 00          callq  43 <main+0x43>
  43:   b8 00 00 00 00          mov    $0x0,%eax
  48:   c9                      leaveq
  49:   c3                      retq
```

Registers

➔ There are 16 registers in total, each storing up to 64-bit values

➔ We can store less in each register, and copy smaller amounts if we choose (by using a different register label for the same name)

➔ Used to hold temporary data

➔ Note that Assembly doesn't really have the concept of types for register values, so it makes no distinction between different unsigned/signed integers, or even pointers and integers

Examples of register names. All of these labels refer to the same register, but they allow us to access the lower portions of the same 64 bit register

```
%rax (64)
%eax (32)
%ax  (16)
%al  ( 8)
```

Figure 3.2 Integer registers. The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

Here we are using "word" to represent 16-bits. When processors became 32-bit, it allowed for "double word." 64 bit processors have the concept of a "quad word"

Special Purpose Registers

In addition to the general purpose registers, there are a few registers that point to specific values that may be interesting to us.

Name	32-bit Label	16-bit Label	8-bit Label	Description
%rsp	%esp	%sp	%spl	**Stack Pointer**. Keeps track of the call stack.
%rbp	%ebp	%bp	%bpl	**Base Pointer.** Points to the bottom of the current function's stack frame. Local variables are usually referenced as an offset from the base pointer.
%rip	%eip	%ip		**Instruction Pointer**. Also known as the program counter, tracks the address of the next instructions
%rflags	%eflags	%flags		**CPU flag.** Some operations will occasionally set flags, such as a warning about division by zero.

Note: In some special cases, %rsp and %rbp can serve as general purpose registers. This will happen when the compiler decides that all local variables can be stored in registers, freeing up %rbp to be used

Calling Convention

Defines how functions on a particular OS and architecture interact. *This allows functions from different languages to interoperate, and let OS run code from different programming languages.* Note that assembly itself doesn't have the concept of functions. Instead calling convention defines how compilers will reason about functions to make them more interoperable.

Entry Sequence

In order to call a function (the callee), the caller function must perform a number of steps, known as the entry sequence:

1. Copy the first 6 arguments into registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9 respectively)
2. Store additional or large arguments on the stack frame, in increasing order (the 7th argument is closer to %rbp)
3. Save any of the caller-saved registers (these are registers that are not preserved across function calls)
4. Execute the callq assembly instruction to pass control to the callee

Exit Sequence

After a function is complete, a number of clean up steps should be performed so control can return successfully to the caller:

1. The callee should place the return value into %rax
2. Restore the stack pointer to it's value at the beginning of function execution with entry %rsp
3. Execute the retq to pass control flow back to the caller. This removes the top return address from the stack and jumps to that address
4. The caller now cleans up argument registers, and restores caller-saved registers

Operand Specifiers

When executing an assembly instruction, there are a number of possible operands (think arguments) you can call with it.

<u>Immediate Values</u>

These are numeric literals that can be passed directly to instructions. They are designated with a $

Form	Operand Value
$*Imm*	*Imm*

Imm in this context can mean

<u>Register Values</u>

By passing a register reference (designated with a %), you are using the value of that register.

Form	Operand Value
$\%r_a$	R[r_a]

Note that R[] refers to the value of the given register.

<u>Memory Values</u>

Use the value at a computed memory address for the instruction, usually designated with parentheses. ***Literal values without $ however are absolute memory addresses!***

Note that M[] refers to the value of the given computed address

The scalar value (shown as s on the right) must be either 1, 2, 4, 8

Form	Operand Value
Imm	M[*Imm*]
($\%r_a$)	M[R[r_a]]
Imm($\%r_a$)	M[*Imm* + R[r_a]]
($\%r_a$,$\%r_a$)	M[R[r_a] + R[r_b]]
Imm($\%r_a$,$\%r_a$)	M[*Imm* + R[r_a] + R[r_b]]
(,$\%r_a$,s)	M[R[r_a] * s]
Imm(,$\%r_a$,s)	M[*Imm* + R[r_a] * s]
($\%r_a$,$\%r_b$,s)	M[R[r_a] + R[r_b] * s]
Imm($\%r_a$,$\%r_b$,s)	M[*Imm* + R[r_a] + R[r_b] * s]

Examples

Given the following register and memory values, compute the result of the operand specifier given.

Memory	
Address	Value
0x100	0xFF
0x104	0x32
0x108	0x2A
0x208	0xFE
0x304	0x69
0x408	0x45

Register	
Address	Value
%rax	0x104
%rbx	0x345
%rcx	0x200
%rdx	0x100

Value	Computation	Result
`$0x104`		`0x104`
`0x104`	*M[0x104] =*	`0x32`
`%rax`	*R[%rax] =*	`0x104`
`(%rax)`	*M[R[%rax]] = M[0x104] =*	`0x32`
`4(%rax)`	*M[4 + R[%rax]] = M[4 + 0x104] = M[0x108] =*	`0x2A`
`(%rax,%rcx)`	*M[R[%rax] + R[%rcx]] = M[0x104 + 0x200] = M[0x304] =*	`0x69`
`(%rcx, %rax, 2)`	*M[R[%rcx] + R[%rax] * 2] = M[0x200 + 2 * 0x104] = M[0x408] =*	`0x45`
`264(%rcx, %rdx, 1)`	*264 = 0x108* *M[0x108 + R[%rcx] + R[%rdx] * 1] = M[0x108 + 0x200 + 0x100] = M[0x408]*	`0x45`
`(,%rax, 2)`	*M[R[%rax] * 2] = M[0x104 * 2] = M[0x208] =*	`0xFE`

Move Instructions

Some of the most commonly used instructions, used to copy data from one place to another. Valid moves are shown below; note how **memory** → **memory** is not possible on x86. Additionally note that an immediate cannot be a destination.

Immediate → Register,
Immediate → Memory,
Register → Memory,
Memory → Register,
Register → Register

Basic Move

These move instructions simply copy the data from the source to the destination without any modifications.

Instruction	**Notes**
`movb <src>, <dest>`	Moves a single byte (8 bits) from the source to destination
`movw <src>, <dest>`	Moves a word (16 bits) from the source to the destination
`movl <src>, <dest>`	Moves a double word (32 bits) from the source to the destination
`movq <src>, <dest>`	Moves a quad word (64 bits) from the source to the destination
`movabsq <imm>, <reg>`	Typical move instructions only permit you to move immediates that are able to represented in 32-bit signed integers. movabsq lets you move a 64-bit immediate directly into a register.

Move Sizes

The specific move instruction to use is based on the sizes of your data. In general, your move should be the minimum of the size of your source and your destination. For example, if you are moving from memory to an 8-bit register, use movb.

Source	Dest	Instruction
$0x104 *(16 bit immediate)*	%rax *(64 bit register)*	movw $0x104 %rax
%ax *(16 bit register)*	$0x202 *(16 bit immediate)*	**Illegal Operation!** An immediate cannot be a destination
(%r8) *(memory location)*	%ebx *(32 bit register)*	movl (%r8) %ebx
$0xEAEAFE85FE85EAEA *(64 bit immediate)*	%rbx *(64 bit register)*	movabsq $0xEAEAFE85FE85EAEA, %rbx *movabsq is needed here because other move instructions can only have 32-bit immediates*
(%rax) *(memory location)*	0 *(memory location)*	**Illegal Operation!** You cannot move from memory location to memory location in x86.

Extension Moves

In addition to basic moves, we can also choose to sign or zero extend the data as we copy it from a source to destination. **Extension moves must have register or memory location as their source, and a register as their destination!**

Zero Extension Move

Instruction	Notes
movzbw <byte> <word>	Moves the byte to a word, zero extending to fill the empty space
movzbl <byte> <long>	Moves the byte to a double word (long), zero extending to fill the empty space
movzwl <word> <double>	Moves the word to a double word (long), zero extending to fill the empty space
movzbq <byte> <quad>	Moves the byte to a quad word, zero extending to fill the empty space
movzwq <word> <quad>	Moves the byte to a quad word, zero extending to fill the empty space

Signed Extension Move

movsbw <byte> <word>	Moves the byte to a word, sign extending to fill the empty space
movsbl <byte> <long>	Moves the byte to a double word (long), sign extending to fill the empty space
movswl <word> <double>	Moves the word to a double word (long), sign extending to fill the empty space
movsbq <byte> <quad>	Moves the byte to a quad word, sign extending to fill the empty space
movswq <word> <quad>	Moves the byte to a quad word, sign extending to fill the empty space
cltq	Sign extends %eax to %rax. This is a compact encoding for a common operation

Memory to Memory Moves

When doing memory to memory operations, it is necessary to use two instructions: one to move data from the source to a register, and one to move the data from the register to the destination. As we have seen previously, when numeric assignment happens between types, variables are either extended or truncated. Here are a few examples of *how* different C assignment instructions translate into assembly.

First Instruction

The first instruction should move the source data into a register *completely*, performing extensions if necessary.

1. If the source is the same size as the destination, then you can directly copy the data into a register without performing an extension. For example, *int = int* should use *movl*
2. If the destination is smaller than the source (i.e. you are losing data), then copy the entire source into the register, you will truncate in the second step. *For example, char = int* should use *movl*
3. If the destination is larger than the source, you will need to perform an extension. If the source type is signed, perform a signed extension (using movs), and if the source type is unsigned, perform a zero extension (using movz). *For example, int = unsigned short* should use *movzwl* and *int = short* should use *movswl*

Second Instruction

The second instruction only needs to copy the data from the register into the destination address. *You should not perform any extension here, but you should perform truncation here*. For example, *int = unsigned short* should use movl, and *char = int* should use *movb*

For these examples, assume the address of the source a is held in `%rsi` and the destination address is held in `%rdi`. Additionally use the 1st register (`%rax %eax %ax %al`) to hold data.

C Assignment	Equivalent Assembly	Notes
`char a = 32;` `short b = a;`	`movsbw (%rsi), %ax` `movw %ax, (%rdi)`	*Signed extension here because char is a signed type*
`unsigned char a = 32;` `int b = a;`	`movzbl (%rsi), %eax` `movl %eax, (%rdi)`	*Zero extension here because char is an unsigned type*
`char a = 32;` `unsigned int b = a;`	`movsbl (%rsi), %eax` `movw %eax, (%rdi)`	*Even though b is unsigned, use a signed extension because a is signed.*
`int a = 4249;` `unsigned short b = a;`	`movl (%rsi), %eax` `movw %ax, (%rdi)`	*Truncation here, but copy the entire int into register*
`short b = -45;` `unsigned short a = b`	`movw (%rsi), %ax` `movw %ax, (%rdi)`	*No extension, direct copy, but data will be misinterpreted*

C Review

Covers topics discussed on March 26

Memory Manipulation Functions

These functions allow us to manipulate memory at a low level more easily. They are provided in the *string.h* library

memcpy

```
void *memcpy ( void *destination, const void *source, size_t num );
```

Copies a set number of bytes from the source buffer to the destination buffer. This is different from strcpy because it ignores the NULL byte (\0).

```
#include <string.h>
#include <stdio.h>

int main() {

        const char source[30] = "Hello World\0Extra String Pog!";
        char dest[30];
        char dest_b[30];

        // Copy exactly 30 bytes
        memcpy(dest, source, 30);

        // Copy until the NULL byte
        strcpy(dest_b, source);

        for (int i = 0; i < 30; i++) {
                // This will output Hello World\0Extra String Pog!
                printf("dest: %c\n", dest[i]);
                // This will output garbage beyond i = 10
                printf("dest_b: %c\n", dest_b[i]);
        }

}
```

memset

```
void *memset ( void *ptr, int value, size_t num );
```

Fills a buffer with a specific value, often zero.

```
#include <stdint.h>
#include <string.h>

int main() {
    unsigned int a = 0xFFFF0000;
    memset(&a, 0x0000, sizeof(uint64_t)); // a is now 0x000000000
}
```

memcmp

```
int memcmp ( const void * ptr1, const void * ptr2, size_t num );
```

Very similar to strcmp, except it does not stop copying upon reaching a NULL byte. Returns 0 if the values of the buffers are equal. If the return value is negative, the first byte that does not match in both memory blocks has a lower value in ptr1 than in ptr2 (if evaluated as unsigned char values). If the return value is positive, the first byte that does not match in both memory blocks has a greater value in ptr1 than in ptr2 (if evaluated as unsigned char values)

```
#include <string.h>
#include <stdio.h>

int main() {

        const char a[30] = "Hello World\0Extra String Pog!";
        const char b[30] = "Hello World\0Different Secret!";

        int mem = memcmp(a, b, sizeof(a));
        int str = strcmp(a, b, sizeof(a));

        // Won't get printed
        if (mem == 0) {
            printf("memcmp says a == b\n");
        }

        // Will get printed
        if (str == 0) {
            printf("strcmp says a == b\n");
        }

}
```

Processes

We can use the fork() system call to duplicate this process at the current execution point. The child process will have initially identical, but duplicated program state.

```
int fork();
```

The return value of the fork call indicates the status thereof. *Negative values indicate that forking was unsuccessful, Zero is returned to the new process, and Positive values are returned to the parent process.*

```
#include <stdio.h>
#include <unistd.h>
int main() {
   int result = fork();
   printf("fork = %d", result);
}
```