1 What is this document?

This document contains theory and related practice problems to help students prepare for Midterm Exam 1 of the CS-4400 Spring 2025 course, taught by Prof. Dr. John Regehr at the University of Utah. It covers topics that students have asked about most frequently. Additional information is provided through links to Compiler Explorer, which include useful comments, assembly code, and even visualized stack diagrams in the comments. Please refer to them as you read.

For errors and additional information, please refer to the Acknowledgement section (§6).

2 Register info

Bit Ranges				Description
63	31	15	7	
%rax	%eax	%ax	%al	Return value
%rbx	%ebx	%bx	%bl	Callee saved
%rcx	%ecx	%cx	%cl	4th argument
%rdx	%edx	%dx	%dl	3rd argument
%rsi	%esi	%si	%sil	2nd argument
%rdi	%edi	%di	%dil	1st argument
%rbp	%ebp	%bp	%bpl	Callee saved
%rsp	%esp	%sp	%spl	Stack pointer
%r8	%r8d	%r8w	%r8b	5th argument
%r9	%r9d	%r9w	%r9b	6th argument
%r10	%r10d	%r10w	%r10b	Caller saved
%r11	%r11d	%r11w	%r11b	Caller saved
%r12	%r12d	%r12w	%r12b	Callee saved
%r13	%r13d	%r13w	%r13b	Callee saved
%r14	%r14d	%r14w	%r14b	Callee saved
%r15	%r15d	%r15w	%r15b	Callee saved

Table 1: Breakdown of x86-64 General Purpose Registers. Remember the register itself is 64-bit wide (0 - 63) . Certain parts of it has specific names. We do not have 64-bit %rax, then additional 32-bit %eax, etc. We have one 64-bit %rax and the lower 4 bytes of it (32-bit) is called %eax.

2.1 Calling Conventions - How parameters are passed to a function

The arguments are passed to a function using the registers:

`%rdi, %rsi, %rdx, %rcx, %r8, %r9`

or using other parts of the registers like using `%edi, %esi, ...`
or `%dil, %sil, ...`, etc. based on the size of the value. If there are more than 6 arguments, the additional arguments are saved in stack.

See an example here: `https://godbolt.org/z/Yesv1Mo9d`

2.2 Calling Conventions - Callee and Caller saved registers

The ABI defines callee-saved registers as:

A function can use one of these registers if it saves it first. The function must restore the register's original value before exiting.

This means the value of the register must be restored; in other words, it has the same value before and after a function call.
For example:

```
mov $0, %rbp    # %rbp has value 0
call fun_uses_rbp()
cmpq $0, %rbp   # %rbp still has the same value 0.
```

Since `%rbp` is callee-saved, the function `fun_uses_rbp()` can use `%rbp` but it must restore the previous value before exiting. That is why, `%rbp` has the same value 0 before and after `fun_uses_rbp()`.

On the other hand, caller-saved (or **not** callee-saved registers) registers' values can be different before and after a function call. Callee function is not responsible to save or restore the values; caller functions must save the values of them if they will need after function call.
For example:

```
mov $0, %rax    # %rax has value 0
call fun_uses_rax()
cmpq $0, %rax   # %rax can have a different value
```

Since `%rax` is not callee-saved, the function `fun_uses_rax()` is not responsible to restore the previous value of `%rax` so, `%rax` may have a different result.

2.3 Calling Conventions - More information

Register `%rax` holds the return value. For example: The C code:

```
long multiply(long a, long b) {
    long return_value = a * b;
    return return_value;
}
```

The assembly code generated:

```
multiply:
        movq    %rdi, %rax
        imulq   %rsi, %rax
        ret
```

As you can see, `%rax` is used as the register that is returned in the end.

See the code here: `https://godbolt.org/z/EjsPf5r7o`

Register `%rsp` is used as the stack pointer, a pointer to the topmost element in the stack.

3 Simple moves

3.1 Move with different sizes

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
char/unsigned char	Byte	b	1
short/unsigned short	Word	w	2
int/unsigned int	Double word	l	4
long/unsigned long	Quad word	q	8
char* (or any pointer)	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

Table 2: Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.

Related Problem 1 (Updated from Practice Problem 3.2)

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands. (For example, `mov` can be rewritten as `movb`, `movw`, `movl`, or `movq`.)

```
mov__   %eax, (%rsp)
mov__   (%rax), %dx
mov__   $0xFF, %bl
mov__   (%rsp,%rdx,4), %d1
mov__   (%rdx), %rax
mov__   %dx, (%rax)
```

Related Problem 2 (Updated from Practice Problem 3.3)

Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line and write the corrected version.

```
movb  $0xF, (%ebx)     | Correct version:
movl  %rax, (%rsp)     | Correct version:
movw  (%rax), 4(%rsp)  | Correct version:
movb  %al, %sl         | Correct version:
movq  %rax, $0x123     | Correct version:
movl  %eax, %rdx       | Correct version:
movb  %si, 8(%rbp)     | Correct version:
```

Instruction	Effect	Description
	Sign-Extending Instructions	
`movs` S, R	$R \leftarrow$ SignExtend(S)	Move with sign extension
`movsbw`		Move sign-extended byte to word
`movsbl`		Move sign-extended byte to double word
`movswl`		Move sign-extended word to double word
`movsbq`		Move sign-extended byte to quad word
`movswq`		Move sign-extended word to quad word
`movslq`		Move sign-extended double word to quad word
`cltq`	%rax $\leftarrow$ SignExtend(%eax)	Sign-extend %eax to %rax
	Zero-Extending Instructions	
`movz` S, R	$R \leftarrow$ ZeroExtend(S)	Move with zero extension
`movzbw`		Move zero-extended byte to word
`movzbl`		Move zero-extended byte to double word
`movzwl`		Move zero-extended word to double word
`movzbq`		Move zero-extended byte to quad word
`movzwq`		Move zero-extended word to quad word
`movzlq (not exist)`		**!!! Why do we not have this?**

Table 3: Move with extensions. As you notice, we do not have `movzlq` because it would be unnecessary based on the following conventions.

3.2 Move with Extensions

Register Updates Based on Data Movement Instructions

As there are two conventions for data movement, the remaining bytes in the register are affected differently:

- Instructions generating **1- or 2-byte** quantities leave the remaining bytes unchanged.
- Instructions generating **4-byte** quantities set the upper 4 bytes of the register to zero.

```
1 movabsq $0x0011223344556677, %rax     %rax = 0011223344556677
2 movb    $-1, %al                      %rax = 00112233445566FF
3 movw    $-1, %ax                      %rax = 001122334455FFFF
4 movl    $-1, %eax                     %rax = 00000000FFFFFFFF
5 movq    $-1, %rax                     %rax = FFFFFFFFFFFFFFFF
```

That is why, `movl $-1, %eax` moves the value to %eax and already `zero extends` (zeros out the upper 4 bytes) because of the conventions above. You can see `movl %eax, %eax`, which is a simple trick to zero out the upper bytes.

FWIW: Do not worry about `movabsq $0x0011223344556677, %rax`; it is just used to move really big numbers. When a 64-bit number can be represented as a 32-bit value, the compiler uses `mov`. If it cannot, then it actually uses `movabsq`. `movabsq` can have a register as a destination. It cannot write to the memory directly.

Most of the time you will see `xorl %eax, %eax` that is used to zero out the `%rax`. Based on the Table 7 xoring the same value with itself will result in zero. Since zeroing `%eax` also zeros out the upper 4 bytes, compiler chooses `xorl %eax, %eax` which is a cheaper than `movq $0, %rax`

Related Problem 3 (Updated from Practice Problem 3.5)

You are given the following information. A function with the prototype:

```
void decode(long *xp, long *yp, long *zp);
```

is compiled into assembly code, yielding the following:

```
decode:
    movq    (%rdi), %r8
    movq    (%rsi), %rcx
    movq    (%rdx), %rax
    movq    %r8, (%rsi)
    movq    %rcx, (%rdx)
    movq    %rax, (%rdi)
    ret
```

Parameters `xp`, `yp`, and `zp` are stored in registers ____, ____, and ____, respectively (Remember calling conventions from§2.1.)

Write C code for `decode` that will have an effect equivalent to the assembly code shown.

```
// Complete the function below
void decode(long *xp, long *yp, long *zp) {




}
```

Answer: `https://godbolt.org/z/9G4696P6j`

Related Problem 4 (Updated from Practice Problem 3.4)

```
void cast(TYPE_1 *sp, TYPE_2 *dp) {
    *dp = (TYPE_2) *sp;
}
```

You are given this function that casts a source value (`*sp`) and writes it to a destination (`*dp`). Write the instructions necessary to perform the same operation with different `TYPE_1` and `TYPE_2`. An example is provided for you.

Remember:

- **You cannot move from memory to memory.**
- When performing a cast that involves both a **size change** and a **change of signedness** in C, the operation should change the size first.

TYPE_1	**TYPE_2**	**Instruction**
long	long	`movq (%rdi), %rax` `movq %rax, (%rsi)`
char	int	
char	unsigned	
unsigned char	long	
int	char	
unsigned	unsigned char	
char	short	

Answer: `https://godbolt.org/z/ePox4Y7sv`

4 Arithmetic and Logical Operations

4.1 Basic operations

Instruction	Effect	Description
`leaq` S, D	$D \leftarrow \&S$	Load effective address
`INC` D	$D \leftarrow D + 1$	Increment
`DEC` D	$D \leftarrow D - 1$	Decrement
`NEG` D	$D \leftarrow -D$	Negate
`NOT` D	$D \leftarrow\sim D$	Complement
`ADD` S, D	$D \leftarrow D + S$	Add
`SUB` S, D	$D \leftarrow D - S$	Subtract
`IMUL` S, D	$D \leftarrow D * S$	Multiply
`XOR` S, D	$D \leftarrow D\hat{}S$	Exclusive-or
`OR` S, D	$D \leftarrow D\|S$	Or
`AND` S, D	$D \leftarrow D\&S$	And
`SAL` k, D	$D \leftarrow D << k$	Left shift
`SHL` k, D	$D \leftarrow D << k$	Left shift (same as `SAL`)
`SAR` k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
`SHR` k, D	$D \leftarrow D >>_L k$	Logical right shift

Table 4: Common arithmetic and logical instructions in x86-64. The `leaq` (Load Effective Address) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation `>>` for right shifts, with signed values using *arithmetic right shift* ('SAR') and unsigned values using *logical right shift* ('SHR'). Logical right shift puts ZEROS to the beginning while Atithmetic right shift puts the SIGN BIT (0 or 1) to the beginning. Additionally, all of them sets the condition flags except `lea`

As you notice, we do operation and write it in destination. Since sub operation subtract source from destination remember in a way that all of them takes D as first argument:
add S D: D = D + S (D comes first)
sub S D: D = D - S (D comes first)

Related Problem 5 (Updated from Practice Problem 3.10)

```
short calc(short a, short b, short c) {

    short d = ______;
    short e = ______;
    short f = ______;
```

```
    short g = ______;
    return g;

}
```

The generated assembly code implementing these expressions is as follows:

```
calc:
    movl    %esi, %eax
    orl     %esi, %edi
    sarw    $11, %di
    notl    %edi
    subl    %edi, %eax
    ret
```

Based on this assembly code, fill in the missing portions of the C code above.

Answer: `https://godbolt.org/z/aq9h3G8h8`

4.2 LEA (Load effective address)

First, let us remember the equation that is used to access certain parts of memory:

> Memory $\quad Imm(r_b, r_i, s) = M[Imm + R[r_b] + R[r_i] \cdot s]$
>
> Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

In Related Problem 6, we will apply our equation to determine the memory address, then *dereference* that address to access the value stored at it.

LEA instruction work the similar way, but it **DOES NOT DEREFERENCE** the memory. For example, imagine register %rax has the value `0x10`. Additionally, the memory address `0x14` points to the value `0x20`. When we execute the instruction:

```
movq 4(%rax), %rsi
```

First, the `mov` instruction takes the value in %rax, which is `0x10`. Then, based on the equation above, it adds `4` to this value: `0x10 + 4 = 0x14`. After that, it dereferences the memory address `0x14`, which holds the value `0x20`, and moves this value into the %rsi register. If we use the `LEA` instruction instead:

```
leaq 4(%rax), %rsi
```

The instruction computes `0x10 + 4 = 0x14` and moves this computed address directly into the %rsi register without dereferencing the memory at `0x14`. As

the name implies, `LEA` (Load Effective Address) loads the address, not the value at that address.

Related Problem 6 (Updated from Practice Problem 3.1)
Assume the following values are stored at the indicated memory addresses and registers:

Address	**Value**	**Register**	**Value**
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Fill in the following values for the indicated operands (Use Equation 4.2):

Operand	**Value (not used LEA)**	**Value (used LEA)**
%rax	_____	_____
0x104	_____	_____
$0x108	_____	_____
(%rax)	_____	_____
4(%rax)	_____	_____
9(%rax,%rdx)	_____	_____
260(%rcx,%rdx)	_____	_____
0xFC(,%rcx,4)	_____	_____
(%rax,%rdx,4)	_____	_____

Compilers take advantage of `lea` to do arithmetic operations. For example, Related Problem 7 is an example where usage of the lea has nothing to do with the addresses but for arithmetic calculations:

Related Problem 7

Consider the following code, in which we have omitted the expression being computed:

```
long calc(long a, long b, long c) {
    long result = ____________;
    return result;
}
```

Compiling the actual function with `gcc` yields the following assembly code:

```
calc:
    leaq    (%rdi,%rdi,2), %rax
    leaq    (%rsi,%rax,4), %rax
    leaq    3(%rax,%rdx), %rax
```

```
ret
```

Fill in the missing expression in the C code above.

Answer: `https://godbolt.org/z/rT5cYaY6z`

5 Control

5.1 Setting flags

In addition to the integer registers, the CPU maintains a set of single-bit *condition code* registers describing attributes of the most recent arithmetic or logical operation. The `leaq` instruction does not alter any condition codes, since it is intended to be used in address computations. Otherwise, all of the instructions listed in Table 4 cause the condition codes to be set. In addition to the instructions in the Table 4 there are two instruction classes (having 8-, 16-, 32-, and 64-bit forms) that set condition codes without altering any other registers: `cmp` and `test`:

Instruction	Based on		Description
`CMP`	S_1, S_2	$S_2 - S_1$	Compare
`cmpb`			Compare byte
`cmpw`			Compare word
`cmpl`			Compare double word
`cmpq`			Compare quad word
`TEST`	S_1, S_2	$S_1 \& S_2$	Test
`testb`			Test byte
`testw`			Test word
`testl`			Test double word
`testq`			Test quad word

Table 5: Comparison and test instructions. These instructions set the condition codes without updating any other registers.

As you notice, `cmp` and `test` use *subtraction* and *and*, respectively. But what makes them different from the corresponding *sub* and *and* instructions in Table 4 is that `cmp` and `test` **DO NOT UPDATE THE VALUE IN THE DESTINATION REGISTER**; they just set the flags.
See the following example.

```
movq $5, %rax                    movq $5, %rax
movq $2, %rbx                    movq $2, %rbx
subq %rax, %rbx                  cmpq %rax, %rbx
# %rbx = %rbx - %rax             # %rbx - %rax, only sets flags
                                 # not assigns val to %rbx
```

After Execution (subq):

```
# %rax = 5
# %rbx = -3
# SF = 1 (negative result)
```

After Execution (cmpq):

```
# %rax = 5
# %rbx = 2 (unchanged)
# SF = 1 (negative result)
```

This demonstrates that while `subq` updates the destination register (%rbx), `cmpq` performs the same subtraction but only modifies the processor flags without changing the register contents. The same idea holds true for the `test` instruction, too.

For example, suppose the `cmpl a b` instruction to perform that does `b - a`, where variables `a`, `b` are integers. Then the condition codes would be set according to the following C expressions:

```
CF  (unsigned) b < (unsigned) a                Unsigned overflow
ZF  (b == a)                                   Zero
SF  ((b - a) < 0)                              Negative
OF  (a < 0 && b > 0 && (b - a) < 0) ||
    (a > 0 && b < 0 && (b - a) > 0)            Signed overflow
```

Related Problem 8

It is easier to see a concept in smaller numbers, so we first consider 4-bit values for condition codes. The concept is the same, since 4-bit range is smaller it makes things easier.

For example:

- **For 4-bit integers:**
  - Unsigned values range from 0 to $2^4 - 1 = 15$.
  - Signed values range from $-2^3 = -8$ to $2^3 - 1 = 7$.
- The same logic applies to 32-bit integers, except that:
  - Unsigned values range from 0 to $2^{32} - 1$.
  - Signed values range from -2^{31} to $2^{31} - 1$.

Binary	Unsigned Value (0 to 15)	Signed Value (-8 to 7)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Table 6: 4-bit Unsigned and Signed Integer Representation

Given two **4-bit numbers** (Use Table 6 as a reference) a and b in two's complement representation:

1. $a = 5$, $b = 3$
2. $a = 7$, $b = -6$
3. $a = -8$, $b = 7$
4. $a = -3$, $b = -5$
5. $a = -8$, $b = -8$

For each case, compute the result of `cmp a b` and determine the values of the following flags: **CF, ZF, SF, OF**

Related Problem 9
From **Related Problem 8**, for each case, compute the result of `add a b` (Use Table 6 as a reference) and determine the values of the same flags (In the **Related Problem 8** you used the conditions for `cmp a b` now you are supposed to work with `add a b`.)
First, try to write condition codes for `add a b`:

`CF`	`Unsigned overflow`
`ZF`	`Zero`
`SF`	`Negative`
`OF`	`Signed overflow`

Then, determine the value of the flags for the following (Numbers are still **4-bit numbers**):

1. $a = 5$, $b = 3$
2. $a = 7$, $b = -6$
3. $a = -8$, $b = 7$
4. $a = -3$, $b = -5$
5. $a = -8$, $b = -8$

Note:
As discussed in the lecture, Prof. Dr. Regehr suggested a simpler approach to handling flags:

Cast register values to a signed 64-bit integer type, perform the arithmetic operation, and then compare the result against INT_MIN and INT_MAX. However, ensure that when casting to the larger type, the conversion correctly sign-extends rather than zero-extends.

This may be easier. You can use the same concept for the above Related Problems (Related Problem 8 and Related Problem 9). Since the numbers in those problems are 4-bits you can extend them to 8-bit and do comparison.

5.2 Usage of Flags

Now that we understand how flags are set, what do these flags actually signify? Their usage can be categorized into three main operations:

1. Setting a single byte to `0` or `1` based on a specific combination of condition codes (`SET` instructions are used).
2. Conditionally jumping to another part of the program (`JMP` instructions are used).
3. Conditionally transferring data (`CMOV` instructions are used.)

Please see Table 8 for the conditions of the operations listed above. As you can see, they use the same suffixes, which change based on whether the values are signed or unsigned. The suffixes **a** (above) and **b** (below) are used for unsigned values, while the suffixes **g** (greater) and **l** (less) are used for signed values. These suffixes determine how different flag combinations affect the result.

Additionally, please familiarize yourself with the truth tables for XOR, OR, NOT, and AND operations, as shown in Table 7, if you are not already familiar with them.

A	B	$A\&B$	$A\|B$	$A\hat{}B$	$\sim A$	$\sim B$
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Table 7: Truth table for AND (&), OR (|), XOR (ˆ), and NOT (∼).

Instruction	Synonym	Effect	Condition
		Set Instructions	
`sete` D	`setz` D	$D \leftarrow ZF$	Equal / zero
`setne` D	`setnz` D	$D \leftarrow \sim ZF$	Not equal / not zero
`sets` D	–	$D \leftarrow SF$	Negative
`setns` D	–	$D \leftarrow \sim SF$	Nonnegative
`setg` D	`setnle` D	$D \leftarrow \sim (SF \hat{} OF) \& \sim ZF$	Greater (signed >)
`setge` D	`setnl` D	$D \leftarrow \sim (SF \hat{} OF)$	Greater or equal (signed $\geq$)
`setl` D	`setnge` D	$D \leftarrow SF \hat{} OF$	Less (signed <)
`setle` D	`setng` D	$D \leftarrow (SF \hat{} OF) \mid ZF$	Less or equal (signed $\leq$)
`seta` D	`setnbe` D	$D \leftarrow \sim CF \& \sim ZF$	Above (unsigned >)
`setae` D	`setnb` D	$D \leftarrow \sim CF$	Above or equal (unsigned $\geq$)
`setb` D	`setnae` D	$D \leftarrow CF$	Below (unsigned <)
`setbe` D	`setna` D	$D \leftarrow CF \mid ZF$	Below or equal (unsigned $\leq$)
		Jump Instructions	
`jmp` *Label*	–	Unconditional jump	–
`je` *Label*	`jz` *Label*	Jump if $ZF = 1$	Equal / zero
`jne` *Label*	`jnz` *Label*	Jump if $ZF = 0$	Not equal / not zero
`js` *Label*	–	Jump if $SF = 1$	Negative
`jns` *Label*	–	Jump if $SF = 0$	Nonnegative
`jg` *Label*	`jnle` *Label*	Jump if $\sim (SF \hat{} OF) \& \sim ZF$	Greater (signed >)
`jge` *Label*	`jnl` *Label*	Jump if $\sim (SF \hat{} OF)$	Greater or equal (signed $\geq$)
`jl` *Label*	`jnge` *Label*	Jump if $SF \hat{} OF$	Less (signed <)
`jle` *Label*	`jng` *Label*	Jump if $(SF \hat{} OF) \mid ZF$	Less or equal (signed $\leq$)
`ja` *Label*	`jnbe` *Label*	Jump if $\sim CF \& \sim ZF$	Above (unsigned >)
`jae` *Label*	`jnb` *Label*	Jump if $\sim CF$	Above or equal (unsigned $\geq$)
`jb` *Label*	`jnae` *Label*	Jump if CF	Below (unsigned <)
`jbe` *Label*	`jna` *Label*	Jump if $CF \mid ZF$	Below or equal (unsigned $\leq$)
		Cmove Instructions	
`cmove` S, R	`cmovz`	Move if $ZF = 1$	Equal / zero
`cmovne` S, R	`cmovnz`	Move if $ZF = 0$	Not equal / not zero
`cmovs` S, R	–	Move if $SF = 1$	Negative
`cmovns` S, R	–	Move if $SF = 0$	Nonnegative
`cmovg` S, R	`cmovnle`	Move if $\sim (SF \hat{} OF) \& \sim ZF$	Greater (signed >)
`cmovge` S, R	`cmovnl`	Move if $\sim (SF \hat{} OF)$	Greater or equal (signed $\geq$)
`cmovl` S, R	`cmovnge`	Move if $SF \hat{} OF$	Less (signed <)
`cmovle` S, R	`cmovng`	Move if $(SF \hat{} OF) \mid ZF$	Less or equal (signed $\leq$)
`cmova` S, R	`cmovnbe`	Move if $\sim CF \& \sim ZF$	Above (unsigned >)
`cmovae` S, R	`cmovnb`	Move if $\sim CF$	Above or equal (unsigned $\geq$)
`cmovb` S, R	`cmovnae`	Move if CF	Below (unsigned <)
`cmovbe` S, R	`cmovna`	Move if $CF \mid ZF$	Below or equal (unsigned $\leq$)

Table 8: Conditional set, move, and jump instructions in x86-64. These instructions depend on condition flags set by previous arithmetic or logical operations.

Time for the problems:

Related Problem 10
Given the C code:

```
int comp(TYPE a, TYPE b) {
    return a COMP b;
}
```

The code above shows a general comparison between arguments a and b where TYPE is the data type of the arguments.

For each of the following instruction sequences, determine which data types TYPE and which comparisons COMP could cause the compiler to generate this code. (There can be multiple correct answers; you should list them all.) Instruction Sequence:

```
cmpl    %esi, %edi
setl    %al
movzbl  %al, %eax
ret
---------------------------------
cmpw    %si, %di
setge   %al
movzbl  %al, %eax
ret
---------------------------------
cmpb    %dil, %sil
setnb   %al
movzbl  %al, %eax
ret
---------------------------------
cmpq    %rsi, %rdi
setne   %al
movzbl  %al, %eax
ret
```

Answer: `https://godbolt.org/z/ndrn6bqWY`

Related Problem 11
Given the C code:

```
int test(TYPE a) {
    return a COMP 0;
}
```

The code above shows a general comparison between arguments a and b where TYPE is the data type of the arguments.

For each of the following instruction sequences, determine which data types TYPE and which comparisons COMP could cause the compiler to generate this code. (There can be multiple correct answers; you should list them all.)
Instruction Sequence:

```
testq   %rdi, %rdi
setle   %al
movzbl  %al, %eax
ret
---------------------------------
testw   %di, %di
sete    %al
movzbl  %al, %eax
ret
```

Answer: `https://godbolt.org/z/5s5ffGPz5`

Related Problem 12 (Updated from Practice Problem 3.18)

Complete the C code below based on the assembly code provided after the C code.

```
long test(long x, long y, long z) {
    long val = __________;
    if (__________) {
        if (__________)
            val = __________;
        else
            val = __________;
    } else if (__________)
        val = __________;
    return val;
}
```

GCC generates the following assembly code. Based on this assembly code complete the C code above.

```
test:
        leaq    (%rdx,%rsi), %rax
        movq    %rax, %rcx
        subq    %rdi, %rcx
        cmpq    $5, %rdx
        jle     .L2
        cmpq    $2, %rsi
        jle     .L3
        movq    %rdi, %rax
        subq    %rdx, %rax
```

```
        ret
.L3:
        movq    %rsi, %rax
        imulq   %rdi, %rax
        ret
.L2:
        cmpq    $2, %rdx
        jle     .L1
        movq    %rcx, %rax
.L1:
        ret
```

Answer: `https://godbolt.org/z/TWezeEM6d`

As we know, `jmp` instructions are used for loops since there is no explicit loop instructions at the assembly level:
1) Compare the condition (check loop condition)
2) If not met, jump back (to loop body)
3) If met, do not jump back and execute instuctions after jump (execute code after loop body).

Related Problem 13

```
For C code having the general form:

long loop(long a, long b)
{
    long result = __________;
    while (__________) {
        result = __________;
        b = __________;
    }
    return result;
}
```

```
loop:
        leaq    (%rsi,%rsi,2), %rax
        addq    %rdi, %rax
        jmp     .L2
.L3:
        addq    %rdi, %rax
        addq    $1, %rsi
.L2:
        cmpq    $10, %rsi
        jg      .L3
        ret
```

Answer: `https://godbolt.org/z/q7c1e45Mb`

Related Problem 14
For the following C code

```
    long modify_long(long a) {
    if (a > 6 || a < 0) {
            a = a + 3;
    } else {
        a = a + 5;
    }
    return a;
}
```

GCC generates the following assembly code with `-Og` flag:

```
    modify_long:
        cmpq    $6, %rdi
        jbe     .L2
        leaq    3(%rdi), %rax
        ret
.L2:
        leaq    5(%rdi), %rax
        ret
```

You can see it here: *https://godbolt.org/z/zdqTv8YWK*

As you can see, after `cmpq $6, %rdi` compiler uses *jbe .L2* instruction. However, as we know ftom Table 8 that `jbe` instruction is used for unsigned values while in our C code the type of `a` is `long` which is signed.

Explain why compiler uses `jbe` even if the value is signed.

Hint: Everything is just bits for the computer. See an example bit patterns for signed and unsigned values in the Table 6.

Related Problem 14
For the C code below:

```
    long cmove(long a, long b) {
        if (_) {
            return _____;
        } else {
            return _____;
        }
}
```

The compiler generates the following assembly code:

```
cmove:
    leaq    (%rdi,%rsi), %rax
    addq    $3, %rsi
    testq   %rdi, %rdi
    cmove   %rsi, %rax
    ret
```

Based on the assembly code, complete the given C code.

Answer: `https://godbolt.org/z/3GWE8bves`

6 Acknowledgement

.

This document has been created by TA Khagan Karimov for the **CS-4400 Spring 2025** course as a **Midterm Review 1** resource. The materials used are based on the textbook *Computer Systems: A Programmer's Perspective (Third Global Edition)* by Randal Bryant and David O'Hallaron, along with lecture slides and notes. Any errors or mistakes are more likely due to my interpretation. If you find an error, please open an issue or submit a pull request here: `https://github.com/khagankhan/cs4400-help`