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 (§9).

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 bits wide (0–63). Certain portions of it have specific names. We do not have separate registers for 64-bit %rax, 32-bit %eax, etc. Instead, there is a single 64-bit %rax register, and its lower 32 bits are referred to as %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 by using other parts of these registers, such as `%edi, %esi, ...` or `%dil, %sil, ...`, depending on the size of the value. If there are more than six arguments, the additional arguments are stored on the stack.
See an example here: `https://godbolt.org/z/Yesv1Mo9d`

2.2 Calling Conventions - Callee and Caller saved registers

The ABI [2] 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 that the value of the register must be restored; in other words, it must have 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 the call to `fun_uses_rbp()`.
On the other hand, caller-saved (or **not** callee-saved) registers may have different values before and after a function call. The callee function is not responsible for saving or restoring their values; instead, the caller function must save their values if they are needed after the 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 for restoring its previous value. As a result, `%rax` may hold a different value after the function call.

2.3 Calling Conventions - More information

Register `%rax` holds the return value.
For example, for the following C code:

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

the following assembly code has been 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`.)
Hint: Choose the suffixes that are compatible with the smaller-sized register. You can extract one byte from an 8-byte register, but you cannot extract 8 bytes from a 1-byte register.

```
mov__   %eax, (%rsp)
mov__   (%rax), %dx
mov__   $0xFF, %bl
mov__   (%rsp,%rdx,4), %dl
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	$R \leftarrow \text{SignExtend}(S)$	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 ← SignExtend(%eax)	Sign-extend %eax to %rax
	Zero-Extending Instructions	
`movz` S, R	$R \leftarrow \text{ZeroExtend}(S)$	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.

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

That is why `movl $-1, %eax` moves the value to %eax and automatically `zero-extends` (zeros out the upper 4 bytes) due to the conventions mentioned above. You may also 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 simply used to move very large numbers. When a 64-bit number can be represented as a 32-bit value, the compiler uses `mov`. If it cannot, then it uses `movabsq`. `movabsq` can have a register as its destination, but it cannot write directly to memory. See an example here: `https://godbolt.org/z/bsvjGsf3G`

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 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	(Do some arithmetic)	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 Arithmetic right shift puts the SIGN BIT (0 or 1) to the beginning. Additionally, all of them set the condition flags except `lea`.

As you notice, an operation is performed, and the result is written to the destination. Remember that all of them take `D` as the 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:

Scaled Indexed Addressing Mode

Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

$$\text{Memory}\quad Imm(r_b, r_i, s) = M[Imm + R[r_b] + R[r_i] \cdot s]$$

In Related Problem 6, we will apply our equation to determine the memory address and then *dereference* that address to access the value stored at it. The LEA instruction uses the same equation for the calculation, but it **does not access memory** or **dereference the pointer**.

For example, imagine that register %rax holds the value `0x10`. Additionally, the memory address `0x14` contains 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`.

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 perform arithmetic operations. For example, Related Problem 7 demonstrates a case where the use of `lea` is unrelated to memory addresses but instead serves purely 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`: As you notice, `cmp` and `test` use *subtraction* and *and*, respectively.

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.

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): **After Execution (cmpq):**

```
# %rax = 5                     # %rax = 5

# %rbx = -3                    # %rbx = 2 (unchanged)

# SF = 1 (negative result)     # 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-like 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`

Note:
Please consider the above as a "C-like expression" rather than actual C code. The authors of the CSAPP book [1] may have mistakenly applied a similar logic to C code. For more details, please refer to `Lec04.pdf`, slide number 83, and Solution 2.30 from the book of the mentioned edition in the Acknowledgment §9.

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 C-like expressions for `add a b` for the given flags:

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

Then, determine the value of the flags after the operation `add a, b` 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 approach may be easier to implement. You can apply the same concept to the above Related Problems (Related Problem 8 and Related Problem 9). Since the numbers in those problems are 4-bit values, you can extend them to 8-bit and perform the comparison accordingly.

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	$D \leftarrow (SF\hat{\ }OF) \vert ZF$	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	$D \leftarrow CF \vert ZF$	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 $(SF\hat{\ }OF) \vert ZF$	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 $CF \vert ZF$	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 $(SF\hat{\ }OF) \vert ZF$	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 $CF \vert ZF$	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 are no explicit loop instructions at the assembly level:

1. Compare the condition (check loop condition).
2. If the condition is not met, jump back to the loop body.
3. If the condition is met, do not jump back and execute the instructions after the jump (execute code after the 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 15
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 Switch case and Jump tables

Compilers translate `switch` statements into **jump tables** when cases are dense. The best way is to look at an example.
For the given example below, instead of multiple comparisons, an index (`n - 100`) maps directly to an entry in `.L4`. For example, `case 104:` lacks a `break`, so it **falls through** to `case 106:`, meaning both use the same `.L3` label in assembly. The **default case** is `.L8`, where out-of-range values jump, setting `x = 0`. **Bias correction** is needed because `switch` cases may not start at 0; the compiler subtracts 100 (`subq $100, %rsi`) to normalize indexing, ensuring efficient lookups in the jump table; so the index may start from zero not from 100.

Please carefully examine the following example, click the *Compiler Explorer link* (`https://godbolt.org/z/EoPx6zooT`) and play with the source code to see how it compiles to a jump table.

```
void switch_case(long x, long n, long *dest)
{
    switch (n) {
        case 100: //  .L7
            x += 1;
            break;
        // Missing case 101: goes to default .L8
        case 102: // quad .L6
            x += 2;
            break;
        case 103: // .L5
            x += 3;
            break;
        case 104:
            /* Fall through  to case 106: which is .L3 */
        // Missing case 105: gos to default .L8
        case 106: // .L3
            x += 4;
            break;
        default:
            x = 0;
    }
    *dest = x;
}
```

The code above compiles to:

```
 switch_case:
        subq    $100, %rsi    # Add bias so it starts from 0:
        cmpq    $6, %rsi      # Max case number 106 now is 6;
                              # 6 = 106- 100 because of above bias
        ja      .L8           # If more than 6 jump to default case .L8
        jmp     *.L4(,%rsi,8) # Otherwise jump to the table to select
.L4:
        .quad   .L7 # Case 100 jumps to .L7
        .quad   .L8 # Case 101 is missing; jmp to default
        .quad   .L6 # Case 102 jumps to .L6
        .quad   .L5 # Case 103 jumps to .L5
        .quad   .L3 # Case 104 falls through to case 6
        .quad   .L8 # Case 105 is missing; jmp to default
        .quad   .L3 # Case 106 jumps to .L3 like case 104 did because of fall through
.L7:
        addq    $1, %rdi     # x+= 1;
        jmp     .L2
.L6:
        addq    $2, %rdi     # x+= 2;
        jmp     .L2
.L5:
        addq    $3, %rdi     # x+= 3;
        jmp     .L2
.L3:
        addq    $4, %rdi     # x+= 4;
.L2:
        movq    %rdi, (%rdx) # *dest = x;
        ret
.L8:
        movl    $0, %edi     # x = 0;
        jmp     .L2
```

See the source code here: `https://godbolt.org/z/EoPx6zooT`

Related Problem 16
The `switch_case` example above specifies that any value less than zero must go to the default case. However, there is no explicit handling of negative values in the assembly code.

Explain how the assembly code handles these cases.

Related Problem 17 (Updated from Practice Problem 3.31)
Complete the following C code based on the assembly code given.
The given C code

```
    void switcher(long a, long b, long c, long *dest)
{
    long val;
    switch(a) {
    case ___:
        c = ___;
        break;
    case ___:
        val = ___;
        break;
    case ___:
        /* Fall through */
    case ___:
        val = ___;
        break;
    case ___:
        val = ___;
        break;
    default:
        val = ___;
    }
    *dest = val;
}
```

compiles to the following assembly code:

```
    switcher:
        cmpq    $7, %rdi
        ja      .L5
        jmp     *.L4(,%rdi,8)
.L4:
        .quad   .L7
        .quad   .L5
        .quad   .L3
        .quad   .L5
        .quad   .L6
        .quad   .L8
        .quad   .L5
        .quad   .L3
.L6:
        movq    %rdi, %rsi
        jmp     .L5
.L7:
        leaq    112(%rdx), %rsi
        jmp     .L5
.L3:
        addq    %rdx, %rsi
        salq    $2, %rsi
.L5:
        movq    %rsi, (%rcx)
        ret
.L8:
        movl    $0, %esi
        jmp     .L5
```

Answer: `https://godbolt.org/z/cnaGd5no3`

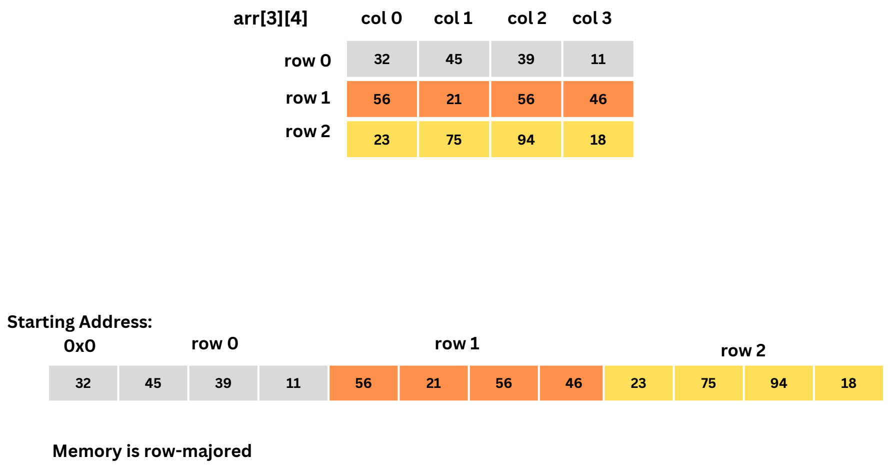

Figure 1: Nested Array Diagram. In C, arrays are saved in *row-major* order.

7 Nested Arrays

The array elements are ordered in memory in row-major order, meaning all elements of row 0, which can be written A[0], followed by all elements of row 1 (A[1]), and so on. In other words, since the memory is one dimensional, nested arrays in C programming language are saved in a *row-major* order in the memory. Figure 1 demonstrates how 2-D array `arr[3][4]` is saved in the memory in a row-major order.

If an array is declared as:

$$\text{Type } A[R][C];$$

where *Type* can be `char`, `int`, `long`, etc., the **memory address** of an element $A[i][j]$ in a row-major order layout is calculated as:

$$A[i][j] = A + \text{sizeof}(\text{Type}) \times (C \times i + j) \tag{1}$$

Let us consider the array in Figure 1 as an example: Given the declaration:

```
char arr[3][4];
```

where `char` has sizeof(`char`) = 1 byte, the memory address of `arr[1][2]` is:

$$\begin{aligned}\text{arr}[2][3] &= \text{arr} + \text{sizeof}(\text{Type}) \times (C \times i + j)\\ &= \text{arr} + 1 \times (4 \times 1 + 2)\\ &= \text{arr} + 1 \times (4 + 2)\\ &= \text{arr} + 6\end{aligned}$$

Thus, `arr[1][2]` is located 6 bytes after the base address *arr*. Since the array starts at address `0x0` from the figure, we need the element at address 6 which is 56.
Let us consider another example,
Given the declaration:

```
int arr[4][5];
```

where `int` has sizeof(`int`) = 4 bytes, we want to find the equivalent `arr[i][j]` corresponding to `((int *)arr)[17]`.
`((int *)arr)[17]` means that we cast our 2-D array, to one dimensional integer array. The memory offset of the `arr[17]` is `17 * sizeof(int)` which is 68. Remember, the equation above is used for the **memory address** of an element $A[i][j]$ in a row-major order layout.
Now we can apply our equation:

$$\begin{aligned}arr[i][j] &= arr + \text{sizeof}(int) \times (C \times i + j)\\ &= arr + 4 \times (5 \times i + j)\\ &= arr + 20 \times i + 4 \times j\end{aligned}$$

Since the offset is 68 it means that

$$20 \times i + 4 \times j$$

must be equal to `68` we get `i = 3` and `j = 2` using through the following:

$$20 \times i + 4 \times j = 68$$

Solving for i:

$$i = \frac{68}{20} = 3 \quad \text{(integer division)}$$

Solving for j:

$$
\begin{aligned}
j = 68 \mod 20 &= 4 \times j \\
&=> 8 = 4 \times j \\
&=> j = 2
\end{aligned}
$$

Thus, the equivalent array access form is:

```
arr[3][2]
```

You may not need to memorize the equation. Once you understand how nested arrays are stored in *row-major* order, as shown in Figure 1, you can easily visualize the memory layout and determine the answer quickly.

Using the equation is fine as long as you correctly implement *pointer arithmetic* when necessary.

As examples of pointer arithmetic, consider the following declarations:

```
char    A[12];
char   *B[8];
int     C[6];
double *D[5];
```

These declarations will generate arrays with the following parameters:

Array	**Element size**	**Total size**	**Start address**	**Element i**
A	1	12	x_A	$x_A + i$
B	8	64	x_B	$x_B + 8i$
C	4	24	x_C	$x_C + 4i$
D	8	40	x_D	$x_D + 8i$

Related Problem 18 (Updated from Practice Problem 3.38)
Based on the provided assembly code, find the values of `M` and `N` for the given C code below:

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

The assembly code:

```
sum_element:
    leaq    (%rdi,%rdi,2), %rdx
    addq    %rsi, %rdx
    leaq    (%rsi,%rsi,4), %rax
    addq    %rdi, %rax
    movq    Q(,%rax,8), %rax
    addq    P(,%rdx,8), %rax
    ret
```

`M` is: ___

`N` is: ___

Answer: `https://godbolt.org/z/68b4fEc38`

8 Heterogeneous data structures

8.1 Struct

Different ways to declare the structure:

```
// Standard struct declaration
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
// Anonymous struct with typedef
typedef struct {
    int i;
    int j;
    int a[2];
    int *p;
} rec_anonymous;
```

```
// Using typedef with struct
typedef struct rec_t {
    int i;
    int j;
    int a[2];
    int *p;
} rec_t;
```

This structure contains four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-byte integer pointer, giving a total of 24 bytes:

Offset	0	4	8	16	24
Contents	`i`	`j`	`a[0]`	`a[1]`	`p`

8.2 Padding

Fields of a `struct` in C are stored in the order in which they are declared, and the C standard specifies that this order cannot be rearranged in memory. Additionally, many computer systems impose alignment restrictions on **primitive** data types, requiring that certain objects be placed at addresses that are multiples of a specific value K (typically 2, 4, or 8). These alignment constraints simplify the design of hardware that interfaces between the processor and the memory system. To satisfy these alignment requirements, padding may be inserted between `struct` members.

The easiest way to determine padding follows two rules:

1. Starting from address zero check if the offset where a variable starts is divisible by the size of its type:
   - If yes, place it without adding any padding.
   - If no, round up to the nearest multiple by adding padding.
2. After placing all fields, check the total size of the struct:
   - Is the total size divisible by the largest data type's size?
   - If not, round it up to the nearest multiple by adding padding.

Let us consider the following example:

```
typedef struct {
    short a;
    double b;
    int c;
} my_struct, *p_my_struct;
```

We have declared `my_struct` and `p_my_struct`, which is a pointer to `my_struct`. The size of `p_my_struct` is always 8 bytes, as it is a pointer. This is the same as declaring a variable of type `my_struct*`. Writing `p_my_struct a;` and `my_struct* a;` are equivalent. Now, let's determine the memory layout of `my_struct` using Rule 1.

1. **Starting at address 0:** Is 0 divisible by the size of `short` (2 bytes)? Yes, because $0 \mod 2 = 0$. We place `a` at address 0, occupying 2 bytes.
2. **Next, adding `double b`:** The next available address is 2. Is 2 divisible by the size of `double` (8 bytes)? No, because $2 \mod 8 \neq 0$. To align `b` properly, we add 6 bytes of padding. Now, `b` starts at address 8 and occupies 8 bytes. The next available address is $8 + 8 = 16$.
3. **Adding `int c`:** Is 16 divisible by the size of `int` (4 bytes)? Yes, because $16 \mod 4 = 0$. We place `c` at address 16, occupying 4 bytes. The total struct size at this point is $16 + 4 = 20$ bytes.

Now, applying Rule 2:

- The largest data type in the struct is `double` (8 bytes).
- Is the total struct size (20 bytes) divisible by 8? No, because $20 \mod 8 \neq 0$.
- To satisfy alignment requirements, we add 4 bytes of padding at the end to make the total size 24, which is divisible by 8.

Final Struct Layout:

```
// | a | padding(6) | b       | c  | padding(4) |
// ^0  ^2          ^8        ^16  ^20          ^24
```

Thus, the final size of `my_struct` is **24 bytes**.
See this code and see the output for the same struct:

`https://godbolt.org/z/anrTEceE1`

Please carefully analyze the following examples and understand. They also include nested structs. You can run the examples below here: `https://godbolt.org/z/YzohaaoEz`

```
typedef struct {
    short a;
    int b;
    int* c;
    short* d;
} p1;


// | a | padding(2) | b | c | d |
//   ^2             ^4  ^8   ^16
// size = 16 + 8 (short* d) = 24



typedef struct {
    int a[2];
    char b[8];
    short c[4];
    long* d;
} p2;


// | a[0] | a[1] | b[0] | b[1] | ... | b[7] | c[0] | ... | c[3] | d
// ^0     ^4     ^8     ^9     ^10          ^16    ^18          ^24
// size = 24 + 8 (long* d) = 32



typedef struct {
    long a[2];
    int* b[2];
} p3;
```

```
// | a[0] | a[1] | b[0] | b[1]
// ^0     ^8     ^16    ^24
// size = 24 + 8(b[1]) = 32



typedef struct {
    char a[16];
    char* b[2];
} p4, *p_p4;


// | a[0] | a[1] | .... | a[15] | b[0] | b[1]
// ^0     ^1     ^2     ^15     ^16    ^24
// size = 24 + 8(b[1]) = 32



typedef struct {
    p4 a[4];
    p1 b;
} p5;


// | a[0] | a[1] | a[2] | a[3] | b
// ^0     ^32    ^64    ^96     ^128
// size = 128 + 24(b) = 152



typedef struct {
    int a;
    int b;
    long c;
    char d;
    char e;
} p6;


// | a | b | c | d | e |
// ^0  ^4  ^8  ^16  ^17
// size = 17 + 7 (!!! The total structure size must be a multiple
// of the largest alignment)
// In our case long is the largest member.
// Total size must be 24 (rounded up to the nearest multiple of 8).
```

Note:
As you may notice, alignment and our struct padding rules are based on primitive data types. This is why, in the case of `p5`, we did not apply our rule that checks whether 128 (the size of `p4 a[4]`) is divisible by 24 (the size of `p1 b`). Instead, we applied padding based on the alignment requirements of the individual fields. Additionally, we did not apply our second rule, which checks whether the total size is divisible by the size of the largest field. In this case, the largest field is `p4`, which has a size of 32 bytes, but the total struct size of 152 is not divisible by 32. Again, this is because we align based on primitive data types. Since structs do not exist at the assembly level, you can think of nested structs as being ”unrolled” into memory, where each nested struct is treated as a contiguous block before aligning the next field.

9 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`

References

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Pearson, 3rd global edition, 2016.

[2] H.J. Lu and Michael Matz and Milind Girkar and Jan Hubička and Andreas Jaeger and Mark Mitchell (Eds.). System V Application Binary Interface: AMD64 Architecture Processor Supplement, 2024. Accessed: Feb 1, 2025.