

SAMPLE Questions

CS 367 - Computer Systems and Programming

Spring Semester, 2023

Exercises For Final Exam

FULL Post-Midterm Topics

The answers will be provided just before the Final Exam. Use these sample questions as exercises to help you study!

Remember, the Final is Cumulative, so continue to study from the Sample Midterm too.

Post Week 8 (Mar 20 - Mar 24)

- Added Questions on Signals and Unix I/O

Post Week 9 (Mar 27 - Apr 1)

- Chapter 3.1-3.5 (Basic Assembly Expressions) Questions Added

Post Week 10 (Apr 3 - Apr 7)

- Chapter 3.1-3.5 (Full Assembly Expressions) Questions Added
- Chapter 3.6 Control Codes (beginning of Assembly Control Flow) Questions Added

Post Week 11 (Apr 10 - Apr 14)

- Chapter 3.6 Assembly Control Flow Questions Added

Post Week 12 (Apr 16 - Apr 21)

- Chapter 3.7 Assembly Procedures Questions Added

Post Week 13 (Apr 24 - Apr 28)

- Chapter 6 Caching Questions Added

Post Week 14 (May 1 - May 5)

- Chapter 9 Virtual Memory Questions Added
- Chapter 4 Architecture Questions Added
- Chapter 7 Linking Questions Added

1 Data Representation and Processes

Q1: 0 points

This is a reminder that the Final will be cumulative.

Questions for Before the Midterm are not in this sample.

- They are in the previous Midterm Sample Guides.
- Only Questions for Assembly and beyond are in this Sample Guide.
- Below are all topics that may be on the Final from the entire course.

Textbook References The chapter references listed in the topic list below are not exclusive, but are simply the starting point in the textbook where you can review the topic.

Topics Topics in this section *may* include:

Bitwise and Logical Operations in C

- (Ch 2.1.1) Base Conversion, between: Binary, Decimal, and Hexadecimal
- (Ch 2.1.3) Addressing and Byte Ordering
- (Ch 2.1.7) Bit-Level Operations in C, to include: |, &, ^, ~
- (Ch 2.1.8) Logical Operations in C, to include: ||, &&, !
- (Ch 2.1.9) Shift Operations in C, to include: <<, >>

Signed and Unsigned Integers

- (Ch 2.2.1) Basic Integer Data Types
- (Ch 2.2.3) Two's Complement (Signed) Representation
- (Ch 2.2.3) Value Ranges on Unsigned Integers: UMIN, UMAX
- (Ch 2.2.3) Value Ranges on Signed Integers: TMIN, TMAX
- (Ch 2.2.4) Converting between Signed and Unsigned Representations
- (Ch 2.2.5) Signed and Unsigned in C
- (Ch 2.2.6) Expanding Signed and Unsigned Bit Representations

Integer Arithmetic

- (Ch 2.3.1) Unsigned Integer Addition
- (Ch 2.3.4) Unsigned Integer Multiplication
- (Ch 2.3.2) Two's Complement Integer Addition
- (Ch 2.3.5) Two's Complement Integer Multiplication
- (Ch 2.3.6) Multiplication and Division via Shifting

IEEE 754 Floating Point

- (Ch 2.4.2-2.4.3) IEEE 754 Representation
- (Ch 2.4.4) Rounding

Arrays

- (Ch 3.8.1) Array Principles
- (Ch 3.8.2) Pointer Arithmetic
- (Ch 3.8.3) Nested Arrays (Two-Dimensional Arrays)

Data Structures

(Ch 3.9.1) Structs

(Ch 3.9.3) Alignment

(Ch 8.1.0) Idea of Exceptions

(Ch 8.1.1) Interrupt Vector (a.k.a. Exception Table)

(Ch 8.1.2) Types of Exceptions

Processes

(Ch 8.2.1) Logical Control Flow

(Ch 8.2.2) Concurrent Flows

(Ch 8.2.5) Context Switching

(Ch 8.4.2) Process States, fork, and exit

(Ch 8.4.3) Reaping Child Processes

(Ch 8.4.5) Loading Programs (execv/execl)

Signals

(Ch 8.5.0) Big List of Signals (REFERENCES for Signal Numbers WILL Be Given)

(Ch 8.5.1) Idea of Signals

(Ch 8.5.2) Sending Signals

(Ch 8.5.3) Receiving Signals and Signal Handlers

Unix I/O

(Ch 10.1 - 10.2) Files in Unix

(Ch 10.3) Opening and Closing Files (open)

(Ch 10.9) I/O Redirection (dup2)

2 x86-64 Assembly

Basic Expressions and Arithmetic in x86-64 Assembly

- (Ch 3.1-3.3) Basics of x86-64 and Data Format Sizes
- (Ch 3.4) General Purpose Registers
 - (Ch 3.4.1) Operand Formats
 - (Ch 3.4.2) Moving Data with `movq`
 - (Ch 3.4.2) Zero and Sign Extending Data Movement
- (Ch 3.5-3.5.2) Basic Arithmetic and Logical Operations (and `imulq`)
 - (Ch 3.5.1) Load Effective Address for Arithmetic
 - (Ch 3.5.3) Shift Operations

Accessing Information in x86-64

- (Ch 3.3) Data Types in x86-64
- (Ch 3.4.0) Register Names
- (Ch 3.4.1) Operand Specifier Formats
- (Ch 3.4.2) Moving Data (Registers and Memory)
- (Ch 3.4.4) Pushing and Popping to the Stack

Arithmetic Operations

- (Ch 3.5.1) Using LEA for Arithmetic
- (Ch 3.5.2) Basic Arithmetic and Logical Operations
- (Ch 3.5.3) Bitwise Shifting

Control Flow

- (Ch 3.6.1) Condition Codes
- (Ch 3.6.3) Jump Instructions
- (Ch 3.6.5) Conditional Instructions
- (Ch 3.6.6) Conditional Move
- (Ch 3.6.7) Loops (Do-While, While, For)
 - (While using Jump-to-Middle or Guarded-Do)
- (Ch 3.6.8) Switch Statements

Procedures

- (Ch 3.7.1) The Stack
- (Ch 3.4.4) Pushing and Popping
- (Ch 3.7.2) Calling and Returning
- (Ch 3.7.3) Argument Passing
- (Ch 3.7.4) Saving Values on the Stack
- (Ch 3.7.5) Saving Values in Registers (Callee/Caller Save)
- (Ch 3.7.6) Recursion

3 Architecture, Linking, and Memory Topics

Architecture, Linking, and Memory

Processor Architecture

(Ch 4.1) Assembly to Machine Code Translation (Overview Only)

(Ch 4.2.1) Basic Logic Gates and Circuits

(Ch 4.3-4.4) CPU Stages and Pipelining (High-Level Concepts)

Linking

(Ch 7.2) Static Linking (Symbol Resolution and Relocation Concepts)

(Ch 7.3-7.4) Object Files (Idea of Sections in Relocatable Object Files)

(Ch 7.5) Global and Local Symbols and Symbol Tables

(Ch 7.6) Symbol Resolution

(Ch 7.6.2) Static Library Linking

Memories and Memory Hierarchies

(Ch 6.2) Locality

(Ch 6.3.1) General Idea of Caching

(Ch 6.4.1) Cache Memory Organization

(Ch 6.4.2) Direct-Mapped Caches

(Ch 6.4.3) Set Associative Caches

Virtual Memory

(Ch 9.1) Idea of Physical and Virtual Addresses

(Ch 9.2) Idea of Address Spaces

(Ch 9.3) Page Tables

(Ch 9.6) Address Translation

(Ch 9.6.2) Translation Lookaside Buffer (TLB)

(Ch 9.6.4) Looking at Address Translation - Big Example

Dynamic Memory (Heap)

(Ch 9.9.1) Allocating with malloc/free

(Ch 9.9.4) Fragmentation

(Ch 9.9.6) Implicit Free Lists

(Ch 9.9.8) Splitting Free Blocks

(Ch 9.9.9) Coalescing Free Blocks

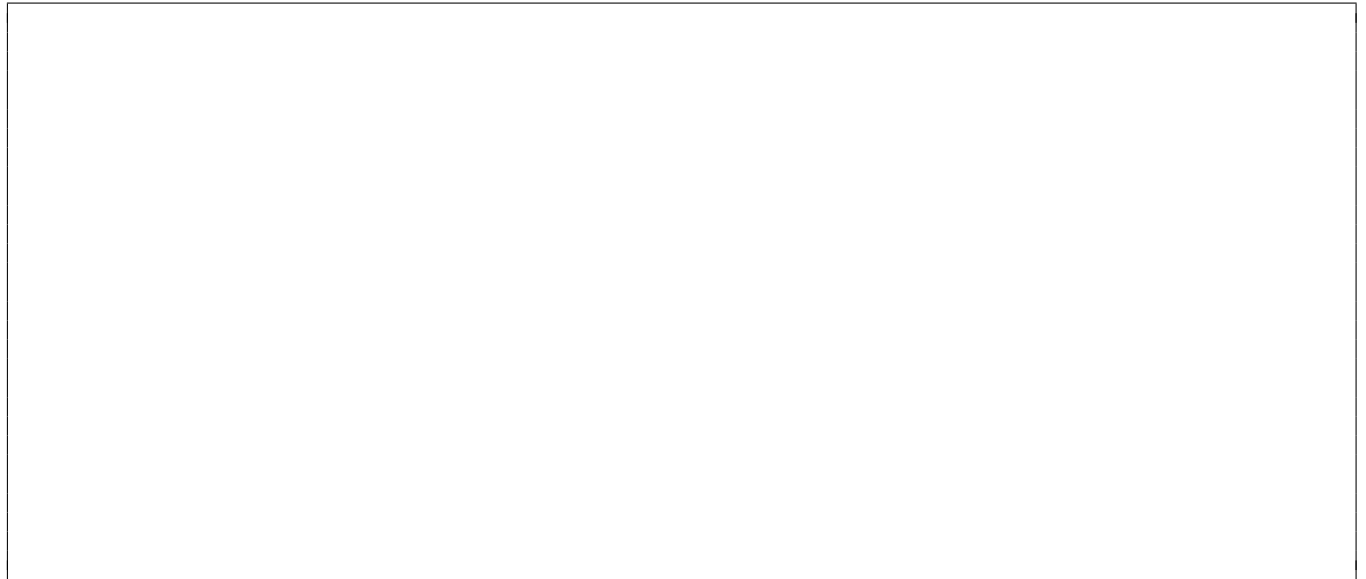
Rest of the Sample The remainder of this sample final consists of sections representing select sample questions of the topics from after the midterm.

Q2: 0 points

Processes: Finish this C program that will create a child that uses **execv** to run "cal" program from /usr/bin with "2077" as the argument. You can hardcode the strings for the argument array (argv) for this question. You are only writing the code for the child process.

- You can hardcode the size of argv to be exactly what is needed for this question.
- If execv fails, exit with code 1.
- You do not have to write in any includes in this answer.

```
int main() {  
    pid_t pid = fork();  
    if(pid == 0) {
```



```
    }  
    int status = 0;  
    waitpid(pid, &status, 0);  
    if(WIFEXITED(status)) {  
        return WEXITSTATUS(status);  
    }  
    return 0;  
}
```

Q3: 0 points

Signals: Write a C program that will print out "Hello World!" over and over again forever, until a user performs Control-C on the keyboard. When Control-C is pressed, your program should print out "Bye Bye!" and then exit immediately with exit code 0.

- The Control-C Associated Signal defined constant is SIGINT.
- You do not have to write in any **includes** in this answer.

Q4: 0 points

Unix I/O: Finish a C program that will create a child process that will execute the `/usr/bin/ls` command with no arguments. This process should redirect its output to "out.txt", which must be opened for writing (`O_WRONLY`, `O_CREAT`, `O_TRUNC`) flags with `0644` as the permissions mode. The parent should wait on the child to finish. (You are only finishing the child code)

```
int main() {
    pid_t child = fork();
    if(child == 0) {
        char *argv[2] = {"ls", NULL};
```

```
        execv("/usr/bin/ls", argv);
        exit(1);
    }
    else {
        waitpid(child, NULL, 0);
    }
}
```

Q5: 0 points

Unix I/O: Finish a C program that will create a child process that will execute the `/usr/bin/wc` command, with `"-l"` as an argument, with `execv`. The child process should redirect its input from "foo.txt", which must be opened for reading (`O_RDONLY`) flags. The parent should wait on the child to finish. (You are only finishing the child code)

```
int main() {
    pid_t child = fork();
    if(child == 0) {
        char *argv[3] = {"wc", "-l", NULL};
```

```
        execv("/usr/bin/wc", argv);
        exit(1);
    }
    else {
        waitpid(child, NULL, 0);
    }
}
```


Q6: 0 points

Machine Level Programming Basics

(a) How many general purpose registers are there in x86-64? (a) _____

(b) What is the name of the lower 8-bits of the RAX Register? (b) _____

(c) What is the name of the lower 32-bits of the RDX Register? (c) _____

(d) What does the RIP register contain? (d) _____

(e) What are the four byte-sizes for Integer data types in Assembly? (e) _____

(f) What is the operation suffix for operating on 16-bits of data? (f) _____

(g) What is the operation suffix for operating on 8-bits of data? (g) _____

(h) What symbol is used to represent an Immediate value as an operand? (h) _____

(i) List the 4 stages of C compiling: (i) _____

Q7: 0 points

Assembly Reading (movq and Arithmetic)

- Below are memory and register values for the following question.
- The values in the tables are in either **hex** or **decimal**.
- **None of these values in either table will change as the result of any operations.**

Table 1: **Memory**

Address	Value
0x10F	1
0x10E	2
0x10D	0x42
0x10C	101
0x10B	113
0x10A	58
0x109	204
0x108	0xAA
0x107	32
0x106	5
0x105	226
0x104	89
0x103	1
0x102	17
0x101	14
0x100	6

Table 2: **Registers**

Register	Value
%rax	50
%rbx	0x42
%rcx	0x104
%rdx	0x100
%rsi	0x2
%rdi	2

For each instruction, write down the value that will be written in to the Destination Register
Treat Each Part as a Separate Question. No Changes Made Affect Future Questions
Remember to use the 0x prefix for Hex values.

- (a) `movq %rdx, %r8` (a) _____
- (b) `addq $2, %rdi` (b) _____
- (c) `shlq $4, %rsi` (c) _____
- (d) `xorq %rax, %rax` (d) _____
- (e) `imulq %rax, %rdi` (e) _____
- (f) `incq %rcx` (f) _____
- (g) `movq $0x42, %r8` (g) _____

Q8: 0 points

Assembly (movq and leaq)

- Below are memory and register values for the following question.
- The values in the tables are in either **hex** or **decimal**.
- **None of these values in either table will change as the result of any operations.**

Table 3: **Memory**

Address	Value
0x10F	1
0x10E	2
0x10D	0x42
0x10C	101
0x10B	113
0x10A	58
0x109	204
0x108	0xAA
0x107	32
0x106	5
0x105	226
0x104	89
0x103	1
0x102	17
0x101	14
0x100	6

Table 4: **Registers**

Register	Value
%rax	50
%rbx	0x42
%rcx	0x104
%rdx	0x100
%rsi	0x2
%rdi	2

For each instruction, write down the value that will be written in to %r8

Treat Each Part as a Separate Question. No Changes Made Affect Future Questions

Remember to use the 0x prefix for Hex values.

- | | |
|--------------------------------|-----------|
| (a) movq %rdx, %r8 | (a) _____ |
| (b) movq (%rdx), %r8 | (b) _____ |
| (c) movq (%rdx, %rdi), %r8 | (c) _____ |
| (d) movq 1(%rdx, %rdi), %r8 | (d) _____ |
| (e) movq (%rdx, %rdi, 2), %r8 | (e) _____ |
| (f) movq 4(%rdx, %rdi, 2), %r8 | (f) _____ |
| (g) movq \$0x42, %r8 | (g) _____ |
| (h) leaq (%rdx), %r8 | (h) _____ |
| (i) leaq (%rdx, %rdi), %r8 | (i) _____ |
| (j) leaq (%rdx, %rdi, 2), %r8 | (j) _____ |
| (k) leaq 4(%rdx, %rdi, 2), %r8 | (k) _____ |

Q9: 0 points

Write the assembly code to compute the following expressions, putting the results into `%rax`
Feel free to use the tables for your own work. Only your code will be graded, not the tables.

(a) $(3x + 5z) * (y + 2z)$

Register	Initial	Final
<code>%rdi</code>	x	
<code>%rsi</code>	y	
<code>%rdx</code>	z	
<code>%rcx</code>		
<code>%rax</code>		
<code>%rbx</code>		

(b) $(x * 2y) + (x + y + 3z)$

Register	Initial	Final
<code>%rdi</code>	x	
<code>%rsi</code>	y	
<code>%rdx</code>	z	
<code>%rcx</code>		
<code>%rax</code>		
<code>%rbx</code>		

Q10: 0 points

Write assembly to swap the values at 0x100 and 0x108 in memory, using the provided registers.

Feel free to use the tables for your own work. Only your code will be graded, not the tables.

Register	Initial	Final
%rcx		
%rdx		
%rsi	0x100	
%rdi	0x108	

Memory	Initial	Final
0x100	42	
0x108	12	

Q11: 0 points

(Condition Codes) After the **sequence** of instructions on the left are executed, list the final value of the four condition codes on the table to the right. Use 0 to indicate if the flag is not set and 1 to indicate the flag is set.

```
movb $0xFF, %a1
movb $0x01, %b1
addb %a1, %b1
```

Condition Code	Value
CF	
ZF	
SF	
OF	

Q12: 0 points

(Condition Codes) Given the set of condition codes on the left, write a small sequence of assembly instructions that will generate the condition codes listed. You may use any operation discussed in class or the textbook.

Condition Code	Value
CF	0
ZF	0
SF	1
OF	1

Q13: 0 points

(Control Flow) Convert the following C code into Assembly

(Assume x starts in %rdi and y starts in %rsi)

```
if(x < y) {  
    return x  
} else {  
    return y;  
}
```

Q14: 0 points

(Control Flow) Convert the following C code into Assembly

(Assume x starts in %rdi and y starts in %rsi)

```
while(x < y) {  
    x++;  
}  
return x;
```

Q15: 0 points

(Control Flow) Convert the following C code into Assembly

(Assume x starts in %rdi and y starts in %rsi)

```
do {  
    x++;  
} (while (x < y);  
return x;
```

Q16: 0 points

(Control Flow) Convert the following C code into Assembly

(Assume x starts in %rdi and y starts in %rsi)

```
int i;  
for(i = 0; i < x; i++) {  
    x+=y;  
}  
return x;
```

Q17: 0 points

(High-Level Interpretation) Convert the following two C functions into Assembly.

You must use Caller Save Registers.

```
long fun1(long x, long y) {
    int temp = x + y;
    temp = fun2(temp);
    return temp;
}
long fun2(long x) {
    if(x == 1) {
        return 1;
    }
    return x * fun2(x-1);
}
```

Q18: 0 points

(High-Level Interpretation) Convert the following two C functions into Assembly.

You must use Callee Save Registers.

```
long fun1(long x, long y) {
    int temp = x + y;
    temp = fun2(temp);
    return temp;
}
long fun2(long x) {
    if(x == 1) {
        return 1;
    }
    return x * fun2(x-1);
}
```


Q19: 0 points

(High-Level Interpretation) Convert the two Assembly functions (fun1 and fun2) into C

```
# Function fun1
fun1:  movq $5, %rdi
       call fun2
       ret

# Function fun2
fun2:  cmpq $2, %rdi
       jl  base
       pushq %rdi
       decq %rdi
       call fun2
       popq %rdi
       pushq %rax
       subq $2, %rdi
       call fun2
       popq %rcx
       addq %rcx, %rax
       jmp end
base:  movq %rdi, %rax
end:   ret
```

Q20: 0 points
(High-Level Interpretation) Use the C and Assembly code to reverse engineer the dimensions of the array. (The Assembly is generated from the C). The constant Y is defined in another file.

Use the code to figure out what value Y has in the array definition.

Values: a has the value 0x6011E0

```
long a[10][Y];
void decr(long i, long j) {
    (a[i][j])--;
}
```

```
decr:  leaq (%rdi,%rdi,2),%rdx
       leaq (%rsi,%rdx,2),%rdx
       leaq 0x6011e0(,%rdx,8),%rdx
       decq (%rdx)
       retq
```

What is Y?

20. _____

Q21: 0 points
(Memory Dereferencing Exercise) Given the main function below in C, write the **extract** procedure in Assembly to return element i.

```
int main() {
    int A[15] = {1,2,3,4,5,6,7,6,5,4,3,2,1};
    srand(time(NULL));
    int i = rand()%15;
    return extract(A, i);
}

// This is the only function to write in Assembly
int extract(int *A, int i) {
    return A[i];
}
```

```
struct s1 {
    short a;
    long b;
    char c[2];
}
```

```
struct s2 {
    int a;
    short b;
    struct s1 *c;
    struct s2 *d;
}
```

Q22: 0 points

Assembly (Memory Dereferencing Exercise)

- (a) Translate the below C functions to the equivalent assembly code in the space provided below, using these above structure definitions (same structs from parts a and b in the last question).

```
long p1(struct s1 *x, long y) {
    return x->b;
}

int p2(struct s2 *x, long y) {
    return x->d->d->a;
}
```

- (b) Translate the below assembly functions to C, using the above structure definitions (from parts a and b). For each prompt, fill in the rest of the line at the prompt on the right.

```
p1:    movb 16(%rsi,%rdi,1), %al
        ret
p2:    movq 16(%rdi), %rdi
        movq 8(%rdi), %rdi
        movw (%rdi), %ax
        ret
```

```
char p1(struct s1 *x, long y) {
    return x->
}

```

(b) _____

```
short p2(struct s2 *x, long y) {
    return x->
}

```

(b) _____

4 Virtual Memory, Caches, and the Heap

Q23: 0 points

Caching: In one word, what concept makes caching effective? 23. _____

Q24: 0 points

Locality: For the code below...

```
int index;
for(index = 0; index < 10000; index++) {
    ary[index]++;
}
```

(a) Which *variable* benefits from Spatial Locality? (a) _____

(b) Which *variable* benefits from Temporal Locality? (b) _____

Q25: 0 points

L1 Cache Address Derivation: Suppose that you have a **different** System that features the following:

- RAM is byte-addressable.
- The L1 Cache is Direct Mapped.
- The Physical Address (PA) is 8 bits.
- There are 16 Cache Sets.
- The Cache Offset is 2 bits.

Use this information to answer the following questions:

(a) How many bytes are in RAM? (a) _____

(b) How many bits are Index portion of the PA? (b) _____

(c) How big (in bytes) is a Cache Block on this new system? (c) _____

Q26: 0 points

L1 Cache Address Derivation: Suppose that you have a **different** System that features the following:

- RAM is byte-addressable.
- The L1 Cache is Direct Mapped.
- There are 256 ($2 * 8$) bytes in RAM.
- The Cache Index is 2 bits.
- There are 32 bytes in each Cache Block.

Use this information to answer the following questions:

(a) How many bits is the Physical Address? (a) _____

(b) How many Sets are in the Cache? (b) _____

(c) How many bits are in the Cache Block Offset? (c) _____

Q27: 0 points

Cache Data Access: Below are the entries for a L1 Cache for a different system with the following properties:

- Physical Address has 11 bits, with 5 bits for the PPN and 6 bits for the PPO.
- Cache Tag (CT) is 5 bits, Cache Index (CI) is 4 bits, Cache Offset (CO) is 2 bits.
- Note: All entries and values in all tables are in HEX.
 - Take care with the number of bits in each section.

L1 Cache						
Set	Tag	Valid	Off 0	Off 1	Off 2	Off 3
0	1B	0	EF	11	23	02
1	16	1	29	D3	AB	01
2	19	0	FE	00	3E	3F
3	0C	1	99	11	5A	FF
4	1A	0	99	D3	B3	23
5	0D	1	CA	3D	CA	00
6	1E	1	BE	67	23	03
7	0C	0	99	31	FE	02
8	02	1	12	34	56	02
9	11	0	99	11	A2	10
A	11	1	BA	E7	0B	11
B	09	1	01	16	02	23
C	0C	0	12	CA	23	C1
D	1A	0	BE	11	23	11
E	01	1	01	02	03	04
F	09	0	99	00	23	11

Translate from Physical Address (PA) to get the byte from Cache (if available)

Write in HIT or MISS-Invalid or MISS-Tag Mismatch for each of the Hit boxes and byte from Cache (or N/A)

(a) **PA = 0x26F** (0010 0110 1111 in binary)

Cache Hit?	Resulting Byte
	0x

(b) **PA = 0x6C2** (0110 1100 0010 in binary)

Cache Hit?	Resulting Byte
	0x

(c) **PA = 0x30C** (0011 0000 1100 in binary)

Cache Hit?	Resulting Byte
	0x

(d) **PA = 0x545** (0101 0100 0101 in binary)

Cache Hit?	Resulting Byte
	0x

Q28: 0 points

Virtual Address Translation: Suppose that you have a System that features the following:

- The Virtual Address (VA) is 10 bits.
- The number of Virtual Pages supported is 32.
- Pages are byte-addressable.
- The Virtual Page Offset is 5 bits.
- The TLB is Direct Mapped.
- The TLB Index is 4 bits.

Use this information to answer the following questions:

- (a) How many bits are in the Virtual Page Number? (a) _____
- (b) How big (in bytes) is one Page? (b) _____
- (c) How many sets are in the TLB? (c) _____

Q29: 0 points

Physical Address Translation:

Suppose that you have a **different** System that features the following:

- The Physical Address (PA) is 8 bits.
- The number of Physical Pages supported is 32.
- Pages are byte-addressable.
- The L1 Cache is Direct Mapped.

Use this information to answer the following questions:

- (a) How many bits are in the Physical Page Number (PPN)? (a) _____
- (b) How many bits are in the Physical Page Offset (PPO)? (b) _____
- (c) How big (in bytes) is a Physical Page on this new system? (c) _____

Q30: 0 points

Virtual Memory Data Access: Below are the entries for a TLB, Page Table, and L1 Cache for a different system with the following properties:

- Virtual Address has 14 bits, with 8 bits for the VPN and 6 bits for the VPO.
- The TLB Tag (TLBT) is 6 bits and the TLB Index (TLBI) is 2 bits in the 8-bit VPN.
- Physical Address has 11 bits, with 5 bits for the PPN and 6 bits for the PPO.
- Cache Tag (CT) is 5 bits, Cache Index (CI) is 4 bits, Cache Offset (CO) is 2 bits.
- Note: All entries and values in all tables are in HEX.
 - Take care with the number of bits in each section.
 - (eg. a TLBT with the value of 21 has only 6 bits (100001))

Translation Lookaside Buffer (TLB)			
Set	Tag	PPN	Valid
0	0A	12	1
1	13	17	1
2	02	1A	1
3	01	11	1

Page Table		
VPN	PPN	Valid
00	00	0
01	13	1
02	1E	1
03	08	0
04	02	1
05	1C	0
06	01	1
07	1A	1
08	00	0
09	10	0
0A	0A	1
0B	07	1
0C	10	0
0D	1F	1
0E	09	1
0F	16	1

L1 Cache						
Set	Tag	Valid	Off.0	Off.1	Off.2	Off.3
0	1B	0	EF	11	23	02
1	16	1	29	D3	AB	01
2	19	0	FE	00	3E	3F
3	3C	1	99	11	5A	FF
4	2A	0	99	D3	B3	23
5	0D	1	CA	3D	CA	00
6	1E	1	BE	67	23	03
7	3C	0	99	31	FE	02
8	02	1	12	34	56	02
9	11	0	99	11	A2	10
A	11	1	BA	E7	0B	11
B	09	1	01	16	02	23
C	0C	0	12	CA	23	C1
D	2A	0	BE	11	23	11
E	01	1	01	02	03	04
F	39	0	99	00	23	11

Translate from VA to PA, then get the byte from Cache (if available)

Write in yes, no, or skip for each of the Hit boxes, write in the PA, and byte (or N/A)

(a) **VA = 0xo1EA** (00000111101010 in binary)

TLB Hit?	PT Hit?	PA in Binary	Cache Hit?	Resulting Byte
				0x

(b) **VA = 0xo1B8** (00000110111000 in binary)

TLB Hit?	PT Hit?	PA in Binary	Cache Hit?	Resulting Byte
				0x

Q31: 0 points

Architecture: Answer each of the following with True or False:

- (a) True or False: The Arithmetic Logic Unit (ALU) executes instructions. (a) _____
- (b) True or False: Data hazards and control hazards are both limiting factors of the performance of a pipelined processor. (b) _____
- (c) True or False: Data hazards involve not knowing where to jump when fetching the next instruction in a pipeline. (c) _____
- (d) True or False: We solve Control hazards by adding NOP instructions. (d) _____

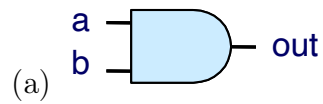
Q32: 0 points

Pipelining: Given a five-stage execution architecture (Fetch, Decode, Execute, Memory, Write-back), rewrite the following code snippet by adding NOP instructions in order to fix the data hazard in the following Assembly program

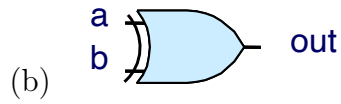
```
movq $1, %rax
incq %rcx
incq %rax
```


Q33: 0 points

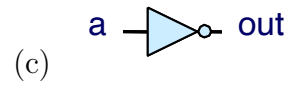
Digital Logic: Identify each of the following with a gate from this list: AND, OR, NOT, XOR



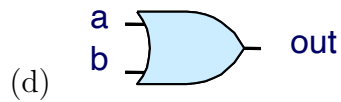
(a) _____



(b) _____



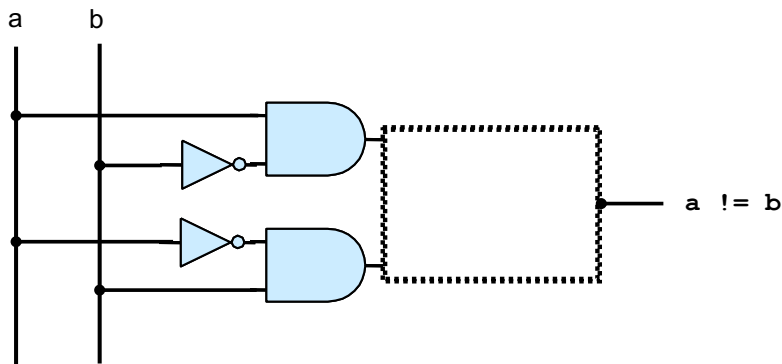
(c) _____



(d) _____

Q34: 0 points

Digital Logic: Select one of these gates to finish this circuit: AND, OR, NOT, XOR



34. _____

5 Linking

Q35: 0 points

Symbol Resolution: The following three C program files are used for this question.

Listing 1: A.c

```
int x;
int y = 4;

void f1() {
    x++;
    y++;
    f2();
}
```

Listing 2: B.c

```
int x = 3;
int y;
int z;

int main() {
    x = 0;
    f1();
    f2();
    printf("%d %d %d\n",
           x, y, z);
    return 0;
}
```

Listing 3: C.c

```
int x;
static int y = 2;
int z = 10;

void f2() {
    x++;
    z--;
    y = 4;
}
```

Complete the tables by listing each symbol in each module as either Weak or Strong. For the Ref field, write in the Module that will be used for any references from this Module. For example, entering C in Ref for f2 means any references to f2 in the current module would reference C's f2.

Mod A	Weak or Strong	Ref
x		
y		
z	N/A	

Mod B	Weak or Strong	Ref
x		
y		
z		

Mod C	Weak or Strong	Ref
x		
y	N/A	
z		

Q36: 0 points

Symbol Resolution:

(a) Which **best** describes the two key functions of a Linker?

- A. Symbol Resolution and Compiling
- B. Symbol Resolution and Relocation
- C. Symbol Resolution and Static Handling
- D. Symbol Resolution and Global Scoping

(b) How do you declare a local function variable to keep its value across function calls?

(c) What are the two types of global symbols that are considered **strong**?
