Quiz 1

- Introduction, Number Systems
- Fixed / Floating Point Numbers
- Memory, CPU & Program Execution
- Instructions, Assembly Language and Machine Code
- High and low level; Compilation and Interpretation

Q1.1	Which of the following statements are true?
(a)	**A linker combines object files into an executable or library**
(b)	Compilation and interpretation are mutually exclusive processes
(c)	A Java compiler produces native machine code
(d)	**High-level languages are typically more readable and user-friendly**

Feedback:

(a) A linker is a computer program that takes one or more **object** files (generated by a compiler or an assembler) and combines them into a single executable file, library file, or another "object" file.

(b) Compilation might be a part of interpretation when it happens just-in-time. These are not mutually exclusive.

(c) The Java compiler (javac) produces bytecode for JVM, which is an intermediate representation.

(d) High-level languages are usually designed to conveniently express a particular domain, e.g. machine learning, so they are intended to be more readable for humans.

Q1.2	Which of the following statements are true?
(a)	A linker combines source files into an executable or library
(b)	**JVM is a stack-based virtual machine**
(c)	Interpretation translates source code to machine code
(d)	**Linking resolves external references to functions and variables**

Feedback:

(a) A linker is a computer program that takes one or more **object** files (generated by a compiler or an assembler) and combines them into a single executable file, library file, or another "object" file.

(b) JVM is a stack-based machine.

(c) An interpreter ‘simulates’ the source code execution.

(d) Source files might contain labels and symbols defined in other source files. The linker’s job is to resolve them and produce a single binary file once these source files are compiled to machine code.

Q1.3	Which of the following statements are true?
(a)	A compiler always translates a program into native machine code
(b)	Programs written in high-level programming languages generally execute faster than programs written in low-level languages
(c)	**A just-in-time compiler generates executable code at runtime**
(d)	**The JVM stack stores local variables and return addresses for function calls**

Feedback:

(a) A compiler may produce a program in some intermediate representation, e.g. java bytecode.

(b) High-level languages contain abstractions that help the language to be more human-readable and user-friendly, but these abstractions can come at the cost of slower execution.

(c) During interpretation it might be beneficial to compile certain pieces of a program into native machine code to gain more efficiency. This happens at runtime and is commonly known as JIT compilation.

(d) The JVM stack stores local variables and return addresses for function calls.

Q2.1	Which of the following statements are true?
(a)	High-level languages are always interpreted
(b)	**Assembly language is Instruction Set Architecture (ISA) specific**
(c)	**A program compiled to an intermediate representation (e.g. Java bytecode) could be executed on different platforms without recompilation**
(d)	Interpretation is the process of converting high-level code to bytecode.

Feedback:

(a) High-level languages can also be compiled. For example, Java is first compiled to an intermediate representation.

(b) Assembly language is ISA specific. You have different instructions available for different architectures. For example, RISC-V and x86 have different instructions.

(c) You can run a program in such intermediate representation on any platform that has a virtual machine for this representation. For example, you can run Java programs on any platform that has JVM implementation.

(d) An interpreter ‘simulates’ the source code execution.

Q2.2	Which of the following statements are true?
(a)	High-level languages are always interpreted
(b)	**Assembly language is Instruction Set Architecture (ISA) specific**
(c)	**Some part of linking may happen at executable loading time**
(d)	Interpretation is the process of converting high-level code to bytecode.

Feedback:

(a) High-level languages can also be compiled. For example, Java is first compiled to an intermediate representation.

(b) Assembly language is ISA specific. You have different instructions available for different architectures. For example, RISC-V and x86 have different instructions.

(c) When linking with shared libraries the resolution happens at loading time (this is so-called dynamic linking).

(d) An interpreter 'simulates' the source code execution.

Q2.3	Which of the following statements are true?
(a)	High-level languages are always compiled
(b)	**Assembly language is Instruction Set Architecture (ISA) specific**
(c)	**Some part of linking may happen at executable loading time**
(d)	**A program compiled to an intermediate representation (e.g. Java bytecode) could be executed on different platforms without recompilation**

Feedback:

(a) High-level languages can also be compiled. For example, Java is first compiled to an intermediate representation.

(b) Assembly language is ISA specific. You have different instructions available for different architectures. For example, RISC-V and x86 have different instructions.

(c) When linking with shared libraries the resolution happens at loading time (this is so-called dynamic linking).

(d) You can run a program in such intermediate representation on any platform that has a virtual machine for this representation. For example, you can run Java programs on any platform that has JVM implementation.

Q3.1	Which of the following statements are true?
(a)	Main memory (RAM) is larger and faster than cache memory
(b)	Registers are faster than cache memory
(c)	Registers are slower to access than main memory
(d)	Main memory is larger than cache memory and slower to access

Feedback:

The number of registers is small. L1 cache is larger. L2 cache will be larger still. Main memory (RAM) will be significantly larger than each of these.

Speed of access can be expressed as: registers - L1 cache - L2 cache - RAM

Q3.2	Which of the following statements are true?
(a)	**Data stored in a register can be accessed faster than data in RAM**
(b)	**Main memory is slower to access than cache memory**
(c)	Access to data in a register will take longer than accessing data in RAM
(d)	The size of registers is greater than the size of RAM

Feedback:
The number of registers is small. L1 cache is larger. L2 cache will be larger still. Main memory (RAM) will be significantly larger than each of these.
Speed of access can be expressed as: registers - L1 cache - L2 cache - RAM

Q3.3	Which of the following statements are true?
(a)	**Registers are slower to access compared to RAM**
(b)	**Cache memory is faster to access than RAM**
(c)	RAM is much larger than cache memory
(d)	**The amount of data that can be held in registers is less than can be held in cache**

Feedback:
The number of registers is small. L1 cache is larger. L2 cache will be larger still. Main memory (RAM) will be significantly larger than each of these.
Speed of access can be expressed as: registers - L1 cache - L2 cache - RAM

Q4.1	Which of the following statements are true?
(a)	**An instruction set architecture (ISA) can have multiple, different hardware implementations**
(b)	**The memory of a von Neumann architecture machine is shared between instructions and data**
(c)	The Data Fetch phase occurs before the Instruction Decode phase
(d)	Data is transferred on the Control bus
(e)	**Conditional branch instructions can reduce the effectiveness of the CPU pipeline**

Feedback:
An instruction set architecture is a formal definition of a computer architecture. It can be implemented in different ways with different compromises, for different purposes. (for instance
The von Neumann architecture only has one memory system. This is used to store both data and instructions.
At its simplest the execution of an instruction has the following phases: Instruction Fetch (IF), Instruction Decode (ID), Data / Operand Fetch (DF/OF), Instruction Execution (EX), Result Return / Store (RR/ST)
Data is transferred on the data bus - the control bus sends control instructions
A pipeline relies on knowing the order in which instructions will be executed. One hazard to this is conditional branch instructions which can change the order of program execution.

Q4.2	Which of the following statements are true?
(a)	**The Harvard architecture has separate memory and pathways for instructions and data**
(b)	**The data bus is accessed by memory, processor and peripherals**
(c)	**Dependencies between consecutive instructions affects the benefit from pipelining**
(d)	A stack based architecture requires a large number of registers
(e)	Instructions are fetched from memory in the data fetch phase of instruction execution

Feedback:

The von Neumann architecture has a single memory and pathway. However, the Harvard architecture has separate memory and pathways for data and instructions.

The data bus is shared by all components of the system.

If an instruction's operation is dependent upon the result of a previous instruction, then this creates a hazard which may affect the throughput of the pipeline.

A stack based architecture does not usually require registers to store intermediate results - operations will be upon the stack.

Instructions are fetched from memory in the Instruction Fetch phase.

Q4.3	Which of the following statements are true?
(a)	**In a von Neumann architecture there is no distinction between instruction and data memory**
(b)	In a stack-based architecture (e.g. JVM), instructions can only access the stack
(c)	**The aim of a pipeline is that consecutive instructions can concurrently be at different phases of execution.**
(d)	**A conditional branch instruction can affect the throughput of the pipeline**
(e)	The control bus is used for the transfer of data between memory and the CPU

Feedback:

The von Neumann architecture has a single memory system and data pathway. This is used for both data and instructions.

Whilst some instructions in a stack-based architecture only operate on the stack (e.g. IADD) others also need to access memory (e.g. ISTORE).

The aim of a pipeline is that consecutive instructions can concurrently be at different phases of execution (e.g. DF,ID, DF etc.)

Data is transferred across the data bus. The control bus is used to send control signals)

Q5.1	Which numbers could the byte $(10001010)_2$ represent on an eight-bit computer, using unsigned, sign-and-magnitude and two's complement representations?
(a)	**138**
(b)	**-10**
(c)	**-118**
(d)	-138
(e)	10
(f)	118
(g)	142
(h)	-142

Feedback:

When considered as an unsigned binary number, 10001010 represents the decimal number 2^7 + 2^3 + 2^1 = 138.

When considered as a signed binary number, 10001010 represents the decimal number -(2^3 + 2^1) = -10.

When considered as a two's complement binary number, 10001010 represents the decimal number -2^7 + 2^3 + 2^1 = -118.

Q5.2	Which numbers could the byte $(11000001)_2$ represent on an eight-bit computer, using unsigned, sign-and-magnitude and two's complement representations?
(a)	**193**
(b)	**-65**
(c)	**-63**
(d)	-193
(e)	65
(f)	63
(g)	187
(h)	-187

Feedback:

When considered as an unsigned binary number, 11000001 represents the decimal number 2^7 + 2^6 + 2^0 = 193.

When considered as a signed binary number, 11000001 represents the decimal number -(2^6 + 2^0) = -65.

When considered as a two's complement binary number, 11000001 represents the decimal number -2^7 + 2^6 + 2^0 = -63.

Q5.3	Which numbers could the byte $(10011000)_2$ represent on an eight-bit computer, using unsigned, sign-and-magnitude and two's complement representations?
(a)	**152**
(b)	**-24**
(c)	**-104**
(d)	-152
(e)	24
(f)	104
(g)	128
(h)	-128

Feedback:

When considered as an unsigned binary number, 10011000 represents the decimal number 2^7 + 2^4 + 2^3 = 152.

When considered as a signed binary number, 10011000 represents the decimal number -(2^4 + 2^3) = -24.

When considered as a two's complement binary number, 10011000 represents the decimal number -2^7 + 2^4 + 2^3 = -104.

Q6.1	Calculate the subtraction of the following two's complement fixed-point binary numbers: $(11001.001)_2$ - $(00001.100)_2$
(a)	**10111.101**
(b)	11010.101
(c)	10111.100
(d)	01000.011
(e)	01000.010

Feedback:

First we flip every bit and add 1 to the LSB of the second argument of the subtraction.

Second argument:	`00001.100`
Flip:	`11110.011`
Add 1 to LSB:	`11110.100`

Then we simply add this and the first argument:

```
   11001.001
 + 11110.100
------------
(1)10111.101
```

Q6.2	Calculate the subtraction of the following two's complement fixed-point binary numbers: $(01101.010)_2$ - $(01000.001)_2$
(a)	**00101.001**
(b)	10101.011
(c)	00101.000
(d)	11010.111
(e)	11010.110

Feedback:

First we flip every bit and add 1 to the LSB of the second argument of the subtraction.

Second argument: 01000.001

Flip: 10111.110

Add 1 to LSB: 10111.111

Then we simply add this and the first argument:

```
   01101.010
 + 10111.111
------------
(1)00101.001
```

Q6.3	Calculate the subtraction of the following two's complement fixed-point binary numbers: $(11010.001)_2$ - $(10000.010)_2$
(a)	**01001.111**
(b)	01010.011 (adds them)
(c)	01001.110 (doesn't add 1 to LSB when converting to 2C)
(d)	10110.001 (flips the wrong one)
(e)	10110.000 (flips the wrong one and doesn't add 1 to LSB)

Feedback:

First we flip every bit and add 1 to the LSB of the second argument of the subtraction.

Second argument: 10000.010

Flip: 01111.101

Add 1 to LSB: 01111.110

Then we simply add this and the first argument:

```
   11010.001
 + 01111.110
------------
(1)01001.111
```

Q7.1	Consider the following MIPS assembly code:
	
	`lw $1, &a`
	`lw $2, &b`
	`L1:`
	`bne $1, $2, L2`
	`sw $1, &a`
	`j exit`
	`L2:`
	`bgt $1, $2, L3`
	`sub $2, $2, $1`
	`j L1`
	`L3:`
	`sub $1, $1, $2`
	`j L1`
	`exit:`
	
	Assuming initial values of a = 24 and b = 18, what will be the final values of a & b when the code reaches exit?

(a)	a = 24, b = 18
(b)	**a = 6, b = 18**
(c)	a = 18, b = 6
(d)	a = 0, b = 18

Feedback:

The above code implements the GCD algorithms, as given below:

```
int gcd(int a, int b)
{
      while (a != b)
            if (a > b)
                  a = a - b;
            else
                  b = b - a;
      return a;
}
```

When a=24 and b = 18, the code gives the final values of a = 6 and b = 18, where a is the GCD of the two input values. The table below shows how the values will be updated:

$1	24	6	6	**6**
$2	18	18	12	6

We only write back the value for $1 to **a**, therefore **b** still has the original value of 18.

Q7.2	Consider the following MIPS assembly code:
	
	`lw $1, &a`
	`lw $2, &b`
	`L1:`
	`  bne $1, $2, L2`
	`  sw $1, &a`
	`  j exit`
	`L2:`
	`  bgt $1, $2, L3`
	`  sub $2, $2, $1`
	`  j L1`
	`L3:`
	`  sub $1, $1, $2`
	`  j L1`
	`exit:`
	
	Assuming initial values of a = 32 and b = 12, what will be the final values of a & b when the code reaches exit?

(a)	a = 32, b = 12
(b)	**a = 4, b = 12**
(c)	a = 12, b = 4
(d)	a = 0, b = 12

Feedback:

The above code implements the GCD algorithms, as given below:

```
int gcd(int a, int b)
{
      while (a != b)
            if (a > b)
                  a = a - b;
            else
                  b = b - a;
      return a;
}
```

When a=32 and b = 12, the code gives the final values of a = 4 and b = 12, where a is the GCD of the two input values. The table below shows how the values will be updated:

$1	32	20	8	8	**4**
$2	12	12	12	4	4

We only write back the value for $1 to **a**, therefore **b** still has the original value of 12.

Q7.3	Consider the following MIPS assembly code:
	
	`lw $1, &a`
	`lw $2, &b`
	`L1:`
	`bne $1, $2, L2`
	`sw $1, &a`
	`j exit`
	`L2:`
	`bgt $1, $2, L3`
	`sub $2, $2, $1`
	`j L1`
	`L3:`
	`sub $1, $1, $2`
	`j L1`
	`exit:`
	
	Assuming initial values of a = 21 and b = 35, what will be the final values of a & b when the code reaches exit?

(a)	a = 21, b = 35
(b)	**a = 7, b = 35**
(c)	a = 35, b = 7
(d)	a = 0, b = 35

Feedback:

The above code implements the GCD algorithms, as given below:

```
int gcd(int a, int b)
{
      while (a != b)
            if (a > b)
                  a = a - b;
            else
                  b = b - a;
      return a;
}
```

When a=32 and b = 12, the code gives the final values of a = 4 and b = 12, where a is the GCD of the two input values. The table below shows how the values will be updated:

$1	21	21	7	**7**
$2	35	14	14	7

We only write back the value for $1 to **a**, therefore **b** still has the original value of 35.

Q8.1	On an eight-bit computer that cannot represent negative numbers (i.e. all numbers are positive integers), a calculation doubles the number 150. Given that this number is represented as an eight-bit binary word, what is the output of the calculation?
(a)	300
(b)	**44**
(c)	255
(d)	-212

Feedback:

The decimal number 150 would be represented as the 8-bit binary word $(10010110)_2$.

Doubling this gives the following calculation (i.e. the word is shifted one to the left):

```
   10010110
+  10010110
-----------
  100101100
```

As this is a nine-bit word, overflow occurs and the MSB is lost.

$(00101100)_2$ represents the decimal number 44.

Q8.2	On an eight-bit computer that cannot represent negative numbers (i.e. all numbers are positive integers), a calculation doubles the number 250. Given that this number is represented as an eight-bit binary word, what is the output of the calculation?
(a)	500
(b)	**244**
(c)	255
(d)	-12

Feedback:

The decimal number 250 would be represented as the 8-bit binary word $(11111010)_2$.

Doubling this gives the following calculation (i.e. the word is shifted one to the left):

```
   11111010
+  11111010
-----------
  111110100
```

As this is a nine-bit word, overflow occurs and the MSB is lost.

$(11110100)_2$ represents the decimal number 244.

Q8.3	On an eight-bit computer that cannot represent negative numbers (i.e. all numbers are positive integers), a calculation doubles the number 178. Given that this number is represented as an eight-bit binary word, what is the output of the calculation?
(a)	356
(b)	**100**
(c)	255
(d)	-100

Feedback:

The decimal number 178 would be represented as the 8-bit binary word $(10110010)_2$.

Doubling this gives the following calculation (i.e. the word is shifted one to the left):

```
   10110010
+  10110010
-----------
  101100100
```

As this is a nine-bit word, overflow occurs and the MSB is lost.

$(01100100)_2$ represents the decimal number 100.

Q9.1	In a hypothetical computer, real numbers are represented as 11-bit floating point binary numbers in the following way: ● The **first** bit is the sign bit of the number (i.e. 1 is negative) ● The next **three** bits are the exponent, which is represented in **two's complement** and **not as a biased/offset number** ● The final **seven** bits are the magnitude of the number On such a computer, how would the number **-3.328125** be represented?
(a)	**10011010101**
(b)	10011110101 *(junk)*
(c)	11101010100 *(assumed it was a fixed point number)*
(d)	10101101010 *(forgot about the hidden 1.)*

Feedback:

The decimal number -3.328125 would be represented in fixed-point binary as 11.010101.

We then normalise this number (meaning that it starts with "1."), which gives us 1.1010101 x 2^1.

This means the exponent is 1, which is represented in 3-bit two's complement as 001, and the magnitude is 11010101.

Finally, the sign bit is clearly 1 as the number is negative.

Therefore, the number is represented on the hypothetical computer as 1 001 1010101.

Q9.2	In a hypothetical computer, real numbers are represented as 10-bit floating point binary numbers in the following way: ● The **first** bit is the sign bit of the number (i.e. 1 is negative) ● The next **two** bits are the exponent, which is represented in **two's complement** and **not as a biased/offset number** ● The final **seven** bits are the magnitude of the number On such a computer, how would the number **0.408203125** be represented?
(a)	**0101010001**
(b)	0101010010
(c)	0011010001
(d)	0001101000
Feedback: The decimal number 0.408203125 would be represented in fixed-point binary as 0.011010001. We then normalise this number (meaning that it starts with "1."), which gives us 1.1010001 x 2^-2. This means the exponent is -2, which is represented in 2-bit two's complement as 10, and the magnitude is 1010001. Finally, the sign bit is clearly 0 as the number is non-negative. Therefore, the number is represented on the hypothetical computer as 0 10 1010001.	

Q9.3	In a hypothetical computer, real numbers are represented as 12-bit floating point binary numbers in the following way: ● The **first** bit is the sign bit of the number (i.e. 1 is negative) ● The next **three** bits are the exponent, which is represented in **two's complement** and **not as a biased/offset number** ● The final **eight** bits are the magnitude of the number On such a computer, how would the number **-7.078125** be represented?
(a)	**101011000101**
(b)	110110011010
(c)	111100010100
(d)	101111100010
Feedback: The decimal number -7.078125 would be represented in fixed-point binary as 111.000101. We then normalise this number (meaning that it starts with "1."), which gives us 1.11000101 x 2^2. This means the exponent is 2, which is represented in 3-bit two's complement as 010, and the magnitude is 11000101. Finally, the sign bit is clearly 1 as the number is negative. Therefore, the number is represented on the hypothetical computer as 1 010 11000101.	

Q10.1	A processor is operating at 50MHz. Each instruction takes a minimum of 8 cycles to execute. The processor has an 8 stage pipeline. If a program starts execution at time 0, what is the theoretical maximum number of instructions that will have completed their execution at the end of 10 milliseconds?
(a)	62,500 instructions
(b)	49,993 instructions
(c)	499,930 instructions
(d)	499,992 instructions
(e)	**499,993 instructions**

Feedback:

Speed = 50MHz = 50,000,000 cycles/second = 50,000 cycles/ms

The processor will execute 500,000 cycles in 10 milliseconds

With pipelining:

Number of clock cycles taken by the first instruction = 8 cycles.

After the first instruction has completely executed, one instruction comes out per cycle.

So, the number of cycles taken by each remaining instruction = 1 cycle.

Number of executed instructions = 1 + (500,000 cycles – 8 cycles)

= 1 + 499,992 = 499,993 instructions

In practice, there are factors that mean that this theoretical maximum is never reached.

Q10.2	A processor is operating at 60MHz. Each instruction takes a minimum of 5 cycles to execute. The processor has a 5 stage pipeline. If a program starts execution at time 0, what is the theoretical maximum number of instructions that will have completed their execution at the end of 10 milliseconds?
(a)	120,000 instructions
(b)	59,996 instructions
(c)	599,960 instructions
(d)	599,995 instructions
(e)	**599,996 instructions**

Feedback:

Speed = 60MHz = 60,000,000 cycles/second = 60,000 cycles/millisecond

The processor will execute 600,000 cycles in 10 milliseconds

With pipelining:

Number of clock cycles taken by the first instruction = 5 cycles.

After the first instruction has completely executed, one instruction comes out per cycle.

So, the number of cycles taken by each remaining instruction = 1 cycle.

Number of executed instructions = 1 + (600,000 cycles – 5 cycles)

= 1 + 599,995 = 599,996 instructions

In practice, there are factors that mean that this theoretical maximum is never reached.

Q10.3	A processor is operating at 45MHz. Each instruction takes a minimum of 9 cycles to execute. The processor has a 9 stage pipeline. If a program starts execution at time 0, what is the theoretical maximum number of instructions that will have completed their execution at the end of 10 milliseconds?
(a)	50,000 instructions (computes without pipelining)
(b)	44,992 instructions (computes for 1ms)
(c)	449,920 instructions (computes for 1ms then multiplied by 10)
(d)	449,991 instructions (forgets to add 1)
(e)	**449,992 instructions**

Feedback:

Speed = 45MHz = 45,000,000 cycles/second = 45,000 cycles/millisecond

The processor will execute 450,000 cycles in 10 milliseconds

With pipelining:

Number of clock cycles taken by the first instruction = 9 cycles.

After the first instruction has completely executed, one instruction comes out per cycle.

So, the number of cycles taken by each remaining instruction = 1 cycle.

Number of executed instructions = 1 + (450,000 cycles – 9 cycles)

= 1 + 449,991 = 449,992 instructions

In practice, there are factors that mean that this theoretical maximum is never reached.