1. The ISA instructions that a computer executes **consist of** a(n) **opcode**, and optionally, a(n) **operand**.

2. **What type of opcode** has a different number of bits in it depending on how many operands the instruction has?

   an Expanding Opcode

3. Does the **Pentium-4** have **fixed** or **variable-length** instructions?

   Variable-length (like most CISC architectures)

4. Does the **UltraSPARC** have **fixed** or **variable-length** instructions?

   Fixed-length (like most RISC architectures)

5. **How long** (how many bits) is an instruction in a computer system that has **64 registers** if the opcode is **12 bits long** and the instruction specifies **3 registers**?

   Registers take 6 bits each (2^6=64)
   (6 x 3) + 12 = 30 bits

6. Write the name of the addressing mode beside each of the descriptions below:

Register Indirect	The data is located in memory at the address held in a register specified by the instruction.
Register	The data is located in the register specified by the instruction.
Direct	The data is located in memory at the address specified in the instruction.
Indexed	The data is located in memory at the address which is computed by adding the contents of a register specified by the instruction to an offset specified in the instruction.
Immediate	The data is specified in the instruction.

7. What type of addressing uses PUSH and POP instructions:

Stack addressing

8. Compute the value of the following postfix expression:

26	15	10	+	5	4	–	+	/	8	x
					↓					
		↓		↓	4					
	↓	10		5	5	1			↓	
↓	15	15	25	25	25	25	26		8	
26	26	26	26	26	26	26	26	1	1	8

9. What are the **names** of these types of **branch addressing**:

a) The value in the instruction is **put into** the Program Counter

Absolute addressing

b) The value in the instruction is **added to** the Program Counter

Relative addressing

10. How does a **CALL** instruction **differ** from a **GOTO** instruction?

In addition to jumping to the target instruction, it also saves the address of the instruction after it, so that the RETURN knows where to go back to once the subroutine has finished.

11. **Where** is the return address (used by the **RETURN** instruction) stored in most Instruction Set Architectures?

The Stack

12. **What is wrong** with the following sequence of instructions:

```
        MOV     CX, FILE_SIZE       !Put file size into the CX register
        MOV     SI,BUFFER           !Point SI to the start of the file buffer
A10:
        MOV     AL,[SI]             !Put a byte from file into the AL register
        CALL    PROCESS_BYTE        !Process the byte in the AL register
        ADD     SI,1                !Point SI to the next byte
        SUB     CX,1                !Subtract 1 from no. of bytes remaining
        JNZ     A10                 !Loop for next byte if not end
```

**The loop won't do the right thing if the file contains 0 (zero) bytes.
Need to use a "test at the top" style of loop instead.**

13. **What is it called** when **every byte** of data read from or written to an I/O device is transferred by **instructions executed by the CPU**?

    Programmed I/O

14. One way to tell when an output device is ready to accept more data is to "**poll**" the device (request and check it's status **over and over again** until it's ready).
    What's a more efficient alternative?

    Have the device send an interrupt to the CPU when it is ready

15. **What event** causes an Interrupt Service Routine to start executing?

    An interrupt

16. What is the difference between a normal procedure and a **recursive** procedure?

    A recursive procedure calls itself (directly or indirectly) and can have multiple sets of variables for each call that was made.

17. Show the first 10 numbers that would be printed by the following subroutine when it's called with parameters of 0 and 1:

```
int  fib( int Param1, int Param2)
{
    print( Param1 );  // Prints value of Param1 followed by a space
    fib( Param2,  Param1+Param2 )
}
```

    0 1 1 2 3 5 8 13 21 34 (Fibonacci series)

18. **Why** would the above subroutine **fail with an error**?

    It will never stop calling itself over and over again, and will eventually run out of stack space to store parameters and call frames

19. **What is the difference** between a **trap** and an **interrupt**?

A trap is synchronous – it's a direct result of executing program instructions.

An interrupt is asynchronous – it originates outside of the CPU and is independent of what instructions are executing.

20. Is it possible for a **trap** to be triggered when there is **nothing** wrong with the program?

Yes – traps such as a "page fault" are a normal part of program execution in a virtual memory system.