1. Think of a single line higher language statement which you think is too complex to be implemented directly in hardware.

   Answer:

   - **If entropy is less than the max, move the stars; draw the stars.**
   - `void (*(*f[])())()` **defines** `f` **as an array of unspecified size, of pointers, to functions that return pointers to functions that return** `void` **in C.**
   - `print("Hello, world!")` **in Python.**

2. In the expression `c = a + b`, what is the operation?

   (a) addition
   (b) assignment
   (c) instruction fetching

   Answer: (a)

3. In the C operation `c = a + b;`, what are the operands?

   (a) `c`
   (b) `a, b`
   (c) `a, b, c`
   (d) `+, =`

   Answer:

   - **(b), in normal convention.**
   - **(c), for the purposes of this course.**

4. Why does MIPS not support the `add` instruction with more operands or variable number of operands?

   Answer: MIPS focuses on maintaining a simple, efficient, and predictable instruction set. Supporting instructions with variable numbers of operands would go against these principles, introducing additional complexity that could hinder the design goals of simplicity, performance, and ease of implementation.

5. Why should there be a separate `addi` instruction when there's already an `add` instruction?

   (a) the hardware to perform the addition arithmetic would be different
   (b) addition of a constant is a common case in HLL code
   (c) no specific reason, the designers felt generous in deciding the set of instructions

   Answer: (b)

6. Which of the following is a common case use of adding a constant to a variable in HLL code?

   (a) function calls
   (b) if-then-else conditional statements
   (c) for loop
   (d) switch case statements

   Answer: (c)

7. What is the benefit (compared to HLL code) of supporting base+register addressing as opposed to just register based addressing?

   (a) accessing the first element of an array
   (b) accessing an arbitrary element of an array
   (c) accessing members of a structure

   Answer: (c)

8. Is a `subi` (subtract immediate) instruction needed where a constant value is subtracted from a register?

   (a) Yes, because subtracting a constant is a common case in HLL code
   (b) No, because subtracting a constant is not a common case in HLL code
   (c) Yes, because `subi` cannot be replaced by an equivalent `addi` instruction
   (d) No, because `subi` can be replaced by an equivalent `addi` instruction in most cases

   Answer: (d)

9. Is a separate `sub` instruction needed, when we already have an `add` instruction in the MIPS instruction set?

   (a) Yes, because subtract is a common case operation in HLL code

   (b) No, because subtract is not a common case operation in HLL code

   (c) Yes, because `sub` cannot be replaced with an equivalent `add` instruction

   (d) No, because `sub` can be replaced with an equivalent `add` instruction

   Answer: (c)

10. What additional information do you need to translate `a[300] = x + a[200];` to MIPS assembly language?

    (a) We need to know whether array `a` indeed has 300 elements

    (b) We need to know whether `x` is positive or negative

    (c) We need to know whether `a[200]` is positive or negative

    (d) We need to know what registers correspond to variables `a` and `x`

    Answer: (d)

11. Translate `a[300] = x + a[200];` from C code into MIPS assembly language, assuming that `a` in in `$s0` and `x` is in `$s1`.

    Answer:

```
# a in s0, x in s1
lw $t0, 800($s0)
add $t1, $t0, $s1
sw 1200($s0), $t1
```

12. Translate `a[300] = x + a[i + j];` from C code into MIPS assembly language. Assume that `a` is in `$s0`, `x` is in `$s1`, `i` and `j` are in `$s2` and `$s3` respectively.

    Answer:

```
# a in s0, x in s1
# i in s2, j in s3
add $t2, $s2, $s3
muli $t2, $t2, 4
add $t3, $t2, $s0
```

```
lw $t0, 0($t3)
add $t1, $t0, $s1
sw 1200($s0), $t1
```

13. How can you achieve bit-by-bit toggling (NOT) by using bit-by-bit NOR?

Answer:

$$\text{NOR}(A, B) = \neg(A \vee B)$$

$$\text{NOT}(x) = \text{NOR}(x, x)$$

To perform a bitwise NOT on a complete binary number using NOR, apply the above formula to each bit of the number.

Suppose we have a 4-bit number $N = (n_3, n_2, n_1, n_0)$, where n_i are the individual bits of the number. To compute $\text{NOT}(N)$, you would apply the NOR operation to each bit:

$$\text{NOT}(N) = (\text{NOR}(n_3, n_3), \text{NOR}(n_2, n_2), \text{NOR}(n_1, n_1), \text{NOR}(n_0, n_0))$$

Example

Consider the 4-bit binary number 1101:

1. Break down the number into its bits:

$$N = (1, 1, 0, 1)$$

2. Apply NOR to each bit with itself:

$$\begin{aligned}
&\text{NOR}(1, 1) = \neg(1 \vee 1) = \neg 1 = 0 \\
&\text{NOR}(1, 1) = 0 \\
&\text{NOR}(0, 0) = \neg(0 \vee 0) = \neg 0 = 1 \\
&\text{NOR}(1, 1) = 0
\end{aligned}$$

3. The bitwise NOT of 1101 is:

$$\text{NOT}(N) = (0, 0, 1, 0)$$

14. By what value will PC be changed from instruction to the next during sequential execution?

    Answer: 4

15. What are the instances of non-sequential execution in HLL code you can think of?

    Answer:

1 Conditional Statements

Conditional statements change the flow of execution based on certain conditions. Examples include:

1.1 If Statements

```
if (condition) {
    // code to execute if condition is true
} else {
    // code to execute if condition is false
}
```

1.2 Switch Statements

```
switch (expression) {
    case value1:
        // code to execute if expression == value1
        break;
    case value2:
        // code to execute if expression == value2
        break;
    default:
        // code to execute if expression does not
           match any case
}
```

2 Loops

Loops allow code to execute repeatedly based on a condition:

2.1 For Loops

```
for (int i = 0; i < 10; i++) {
    // code to execute in each iteration
}
```

2.2 While Loops

```
while (condition) {
    // code to execute as long as condition is true
}
```

2.3 Do-While Loops

```
do {
    // code to execute
} while (condition);
```

3 Function Calls

Function calls cause execution to jump to a different part of the code where the function is defined. Control returns to the calling location after the function completes.

```
void myFunction() {
    // code to execute
}

int main() {
    myFunction(); // jumps to myFunction
    // code after function call
}
```

4 Exception Handling

Exception handling mechanisms alter the flow of execution when errors occur:

```
try {
    // code that may throw an exception
} catch (ExceptionType e) {
    // code to handle the exception
}
```

5 GOTO Statements

The ‘goto‘ statement allows an unconditional jump to another part of the code. This is generally discouraged but is still a valid non-sequential execution mechanism.

```
start:
    // code
    goto end;
    // more code that is skipped
end:
    // code to execute after jump
```

6 Multithreading and Concurrency

Multithreading and concurrent programming introduce non-sequential execution by allowing multiple threads or processes to run in parallel.

```
#include <thread>

void threadFunction() {
    // code to execute in a separate thread
}

int main() {
    std::thread t(threadFunction); // start a new
      thread
```

```
    // code to execute concurrently with
       threadFunction
    t.join(); // wait for the thread to finish
}
```

7 Event-Driven Programming

In event-driven programming, the flow of execution is driven by events such as user interactions, timers, or messages.

```
document.getElementById("myButton").addEventListener
  ("click", function() {
    // code to execute when button is clicked
});
```

8 Coroutines

Coroutines allow execution to be suspended and resumed at different points, facilitating non-linear control flow.

```
async def myCoroutine():
    await someAsyncOperation()
    # code to execute after the async operation
       completes
```

9 Callbacks

Callbacks are functions passed as arguments to other functions, which are then executed at a later time, potentially altering the flow of execution.

```
function doSomething(callback) {
    // code
    callback(); // call the callback function
}

doSomething(function() {
```

```
    // code to execute in the callback
});
```

10 Signal Handling

In some programming languages, especially in systems programming, signals can be used to handle asynchronous events, interrupting the normal flow of execution.

```
#include <signal.h>

void signalHandler(int signal) {
    // code to handle signal
}

int main() {
    signal(SIGINT, signalHandler); // register
      signal handler
    // code
}
```

16. Translate the given if-then-else code from C to MIPS assembly language, using the instructions learnt so far:

```
if (x == 0) {
    y = x + y;
} else {
    y = x - y;
}
```

Answer:

```
# s0 is x, s1 is y
bne $s0, $zero, ELSE
add $s1, $s0, $s1
j EXIT
ELSE:
sub $s1, $s0, $s1
```

```
EXIT:
# Further instructions below
```

17. Why is there a special register always fixed to the value zero?

    (a) This is a design error and a waste of a register in the register set

    (b) This is an implementation bug

    (c) HLL code heavily uses the value zero, and by providing a register always set to zero, we save the step of initializing a register to zero

    Answer: (c)

18. Translate the given while loop from C to MIPS assembly language, using the instructions learnt so far:

```
while (a[i] != 0) i++;
```

    Answer:

```
# s0 is a, s1 is i
BEGIN:
sll $t0, $s1, 2
add $t0, $t0, $s0
lw $t1, 0($t0)
beq $t1, $zero, EXIT
addi $s1, $s1, 1
j BEGIN
EXIT:
# Further instructions below
```

19. What does the following assmebly language statement do?

```
sll $t0, $s1, 2
```

    Answer: It takes the value in register `$s1`, shifts it left by 2 positions, and stores the result in register `$t0`. This operation is equivalent to multiplying the value in `$s1` by 4.

20. Is the additional register `$t1` necessary in the given solution, or can we make do with just `$t0`?

    Answer:

- The `lw` instruction loads the value of `a[i]` into `$t1`.
- The `beq` instruction uses `$t1` to compare `a[i]` with zero.

Since the `beq` instruction needs the value of `a[i]` for comparison, `$t1` is essential for this operation.

Simplified Code Without $t1

You can avoid using `$t1` by reusing `$t0` for both loading the value and performing the comparison:

```
# s0 is a, s1 is i
BEGIN:
sll $t0, $s1, 2       # $t0 = i * 4
add $t0, $t0, $s0     # $t0 = address of a[i]
lw $t0, 0($t0)        # Load a[i] into $t0
beq $t0, $zero, EXIT # If a[i] == 0, jump to EXIT
addi $s1, $s1, 1      # Increment i
j BEGIN              # Jump to BEGIN
EXIT:
# Further instructions below
```

In this revised code, `$t1` is not used; instead, `$t0` is reused for both the load and comparison operations.

21. What is the `slt`-based instruction to set `$t0` if `$s0` is greater than `$s1`? (Use all small case, a space between the instruction and the operands, and a single comma and no space separating adjacent operands).

    Answer: `slt $t0, $s1, $s0`

22. What is the `slt`-based instruction to set `$t0` to zero if `$s0` is less than or equal to than `$s1`? (Use all small case, a space between the instruction and the operands, and a single comma and no space separating adjacent operands).

    Answer: `slt $t0, $s1, $s0`

23. What is the `slt`-based instruction to set `$t0` to zero if `$s0` is greater than or equal to than `$s1`? (Use all small case, a space between the instruction

and the operands, and a single comma and no space separating adjacent operands).

Answer: `slt $t0, $s0, $s1`

24. Why does MIPS not support a specific blt instruction? Why does it force a two-instruction sequence for branching based on less-than comparison?

    (a) HLL code does not really use branching based on less-than comparison very much

    (b) This was a design error while deciding the MIPS instruction set

    (c) Supporting a complicated instruction potentially reduces the overall performance of the processor

    Answer: (c)

25. Translate the given for loop from C to MIPS assembly language, using the instructions learnt so far:

```
for(i = 0; i < 10; i++) {
    a[i] = 0;
}
```

Answer:

```
# s0 is i, s1 is a
addi $s0, $zero, 0       # Initialize i to 0
BEGIN:
slti $t0, $s0, 10        # Set $t0 to 1 if i < 10,
  otherwise 0
beq $t0, $zero, EXIT     # If $t0 == 0 (i >= 10), jump
   to EXIT
sll $t1, $s0, 2          # $t1 = i * 4 (to compute the
   byte offset)
add $t2, $t1, $s1        # $t2 = address of a[i] (base
   address of a + offset)
sw 0($t2), $zero         # Store 0 into a[i]
addi $s0, $s0, 1         # Increment i
j BEGIN                  # Jump to BEGIN
EXIT:
# Further instructions below
```

26. What is the minimum number of temporary registers needed for the for loop implementation above?

    (a) 0

    (b) 1

    (c) 2

    (d) 3

    Answer: (b)