Part I: Fundamentals

1. Module 1: Logic Design (for IS1500 only)
   (a) Short answer: 11110101
   Elaborated answer: Convert number 11 into binary form 00001011. Invert all the bits: 11110100. Add one to the result: 11110101.
   (b) Short answer: \( Y_0 = 0, Y_1 = 0, Y_2 = 1 \), and \( N = 2 \).
   (c) Short answer: \( A = 4, B = 9 \), and \( C = 5 \).

2. Module 2: C and Assembly Programming
   (a) The following is printed to standard output:
   
   11 42
   
   (b) The MIPS assembly instruction is:
   
   `nor $t3, $s1, $s4`

3. Module 3: Processor Design
   (a) Short answer: i) The interaction between the `addi`, the `sw`, and the `sub` instructions cause hazards because the result in \( s0 \) is not available in the register file when `sw` and `sub` need it. ii) These are so called data hazards. iii) The data hazard can be solved by forwarding (for instance, when \( s0 \) is computed in the execute stage of `addi`, the result can be forwarded to the execute stage of the `sw` instruction for computing the address). iv) This solution with forwarding does not result in any extra cost in clock cycles.
   
   (b) The signal values are as follows:

   \[
   \begin{align*}
   S_1 &= 1 \\
   S_2 &= 0 \\
   S_3 &= 0x00ff0008 \\
   S_4 &= 0x00000014 \\
   S_5 &= 0x00000020 \text{ or “don’t care”}
   \end{align*}
   \]

---

4. Module 4: Memory Hierarchy

(a) Short answer: The associative number is 4.
   Elaborated answer: Because the set field is 6 bits, there must be $2^6 = 64$ sets (also called rows). There are in total 1024 bytes of data and each block consist of 4 bytes. Hence, there are in total $1024/4 = 256$ blocks. Hence, the associativity number is $N = 256/64 = 4$. Consequently, it is a 4-way cache.

(b) • Shower Answer: The hit rate is 75%.
   Elaborated answer: We get a cache miss at every forth instruction because the instruction size is 4 bytes and we read 16 bytes at each cache miss. Note also that we start at an address where the first instruction in the sequence is the first instruction in the cache block.
   • Short answer: Besides the data, valid bits and tag data must be stored in the cache. In this case, the valid bit is 1 bit and the tag data is $32 - 8 - 4 = 20$ bits.

(c) Short answer: It is called virtual memory. Another benefit with virtual memories is that memory is protected from other concurrently running processes in the operating system (OS).

5. Module 5: I/O Systems

(a) The following C code shows a possible solution.

```c
volatile int* push_buttons = (volatile int*) 0x800;
volatile int* LED = (volatile int*) 0x700;
while(1){
    if((*push_buttons) & 8)
        *LED = 0x3f;
    else
        *LED = 0x0;
}
```

Note that the volatile keyword must be used to make sure that the compiler does not optimize the memory mapped I/O accesses. Note also that we accept the solution where button 3 has bit pattern 100, i.e, that the if-expression is if((*push_buttons) & 4), because it might be possible to misinterpret the exercise.

(b) Possible short answers:
   A keyboard can trigger an interrupt each time a key is pressed.
   A timer can trigger an interrupt periodically.

(c) Short answer: In an synchronous bus, data is transmitted using a clock, whereas in an asynchronous bus, a handshaking protocol is used.
6. **Module 6: Parallel Processors and Programs**

(a) Short answer: This is an example of subword parallelism, single-instruction multiple-data (SIMD), or data-level parallelism.

(b) Short answer: The maximal speedup is $\frac{1}{0.8}$.
   Elaborated answer: Using Amdahl’s law, we get the maximal speedup $\frac{1}{\frac{1}{N} + 0.8} = 1/0.8 = 1.25$ when $N \to \infty$.

(c) Short answer: The CPI is $\frac{18}{27} = \frac{2}{3}$
   Elaborated answer: The instructions executed before the loop take 2 cycles. The loop is executed 8 times and the instructions in each iteration takes 2 cycles. Hence, the total number of cycles are $2 + 8 \times 2 = 18$.
   Before the loop, 3 instructions are executed. Inside the loop, 3 instructions are executed in each iteration and the loop is executed 8 times. Hence, the number of executed instructions are $3 + 8 \times 3 = 27$.
   Consequently, the CPI is $\frac{18}{27} = \frac{2}{3}$. 

Part II: Advanced

7. Hi!

Great that you are going to buy a new computer, but you have unfortunately misunderstood some concepts. Here are some clarifications.

- The performance of a computer is not only determined by the clock frequency. For instance, the cache size and associativitiy of the cache have also significant impact on the performance. Moreover, different technologies can affect how many instructions that can be fetched and executed in parallel. Finally, it is not possible to buy a computer with a 8 GHz processor. The computer would be way too hot; there is no manufacturer that can deliver such processor.

- Fortunately, Moore’s law still holds. The number of transistors on a chip still doubles every 18 to 25 months. This is the exact reason for why we can see that more and more cores are available on new processors. So, you can expect that the number of cores will increase in the future. Buying a computer with 4 cores today seems like a good buy. But, you cannot expect a speedup of 4 because very few programs can really utilize multicore very well today.

- It is true that most modern processors have instruction level parallelism (ILP). In particular, many commercial processors are superscalar processors where instructions are fetched and dynamically scheduled in parallel. The main benefit is, like you say, that programmers do not need to change their programs to gain any extra speedup. However, there is a limitation with this approach, mostly depending on the dependencies between instructions.

- It is true that many processors have multimedia extensions. The Advanced Vector Standard (AVX) that you mentioned is a good example. However, this is not an example of MIMD, but of Single-Instruction Multiple-Data (SIMD) that uses data-level parallelism. Each instruction can operate on multiple data items in parallel.

- It is common with hardware multi-threading in many processors. When this is combined with superscalar processors, it is often referred to as Simultaneous Multi-Threading (SMT). However, hardware multithreading is not about fetching and operating on multiple data items (this is the idea of SIMD). Instead, the main purpose of hardware multithreading is to hide latencies, such as cache misses. Basically, it means that the processor is kept busy working in other hardware threads while it is waiting on a memory access.

That’s it. I hope that this can be to some help for you.

Best regards, David
8. The following MIPS code is a possible translation.

```
bintext2int:
    addi $t0, $0, 0  # int n = 0
while_loop:
    add $t1, $a0, $t0  # address to text[n]
    lbu $t2, 0($t1)    # text[n]
    beq $t2, $0, done_while
    addi $t0, $t0, 1  # n++
    j while_loop

done_while:
    addi $t1, $0, 0  # k = 0
    addi $v1, $0, 0   # res = 0
    addi $t0, $t0, -1 # i = n-1
for_loop:
    slt $t2, $t0, $0   # i < 0
    bne $t2, $0, done_for # jump if i < 0
    add $t2, $t0, $a0  # address to text[i]
    lbu $t3, 0($t2)    # text[i]
    addi $t3, $t3, -48 # - '0'
    sllv $t3, $t3, $t1 # << k
    add $v0, $v0, $t3  # res +=
    addi $t1, $t1, 1
    addi $t0, $t0, -1
    j for_loop

done_for:
    jr $ra
```
• The instruction cache hit rate can be computed as follows. There are \(2048/8 = 128\) sets in the instruction cache. This means that there are 7 index bits and 3 byte offset bits. Because the program starts at an address where all these 10 bits are zero and that the program size is clearly smaller than 2kB, we can directly see that no code fetch will evict any block. Hence, since there are 10 instructions, and two instructions fit into each block, we will get 5 misses in total. Each instruction fetch will result in a memory access. This means that the number of executed instructions will be the same as the number of memory accesses. Before the loop, 3 instructions are executed. The loop contains 7 instructions and the loop will execute 100 times, which means that 700 instructions will be executed. At the end, the \texttt{slti} and the \texttt{beq} instructions will execute one more time, before the loop terminates. Consequently, the total number of executed instructions and memory accesses are
\[
3 + 700 + 2 = 705.
\]
Hence, the instruction cache hit rate is
\[
(705 - 5)/705 = \frac{700}{705}.
\]
The instruction cache shows both examples of temporal and spatial locality. For the first 10 instructions that are executed, spatial locality results in that we only get misses every second instruction. After that, in the loop, we get hits all the time, which is an example of temporal locality.

• The data cache hit rate can be computed as follows. There is only one instruction \texttt{lw} that accesses the data memory. Since the loop is iterating over one big array, the same address will never be accessed again. Consequently, there will be no temporal locality and the fact that the data cache is a two-way cache will not help the hit rate. There will be 100 memory accesses, one access in each loop. Because the block size is 16 bytes, there will be 25 cache misses and 75 cache hits. Hence, the hit rate is
\[
\frac{75}{100} = \frac{3}{4}.
\]
Only spatial locality is used.

• We first compute the number of clock cycles without considering cache misses, and then add these numbers at the end. The first 3 instructions take 3 clock cycles. Because the address for the \(j\) instruction is computed at the decode stage, there will always be one clock cycle miss penalty, so the \(j\) instruction always takes 2 clock cycles. The \texttt{slti}, \texttt{add}, and \texttt{addi} instructions take 1 clock cycle each. The \texttt{beq} instruction takes 1 clock cycle the first 100 times it is executed (we assume branch-not-taken) \(^2\). The last time, when it jumps to \texttt{done}, we have a branch miss prediction penalty of 1, so the \texttt{beq} instruction takes 2 clock cycles in the last case. Finally, the \texttt{lw} instruction takes 1 clock cycle, if we are not considering cache misses. However, there is a data hazard between the \texttt{lw} instruction and the next \texttt{add} instruction, so the pipeline needs to stall one cycle. In summary, the number of cycles are
\[
3 + 100(1 + 1 + 2 + 1 + 1 + 1 + 2) + 1 + 2 = 3 + 900 + 3 = 906
\]
cycles. We know that we have 5 instruction cache misses and 25 data cache misses and that the miss penalty is 5 cycles. Hence, the total number of clock cycles are
\[
906 + 5(5 + 25) = 1056.
\]
From the earlier exercise, we know that the total number of executed instructions is 705. Consequently, the CPI is
\[
\frac{1056}{705}.
\]
\(^2\)Note that there is a RAW dependency between the \texttt{slti} and the \texttt{beq} instruction. The result of the \texttt{slti} instruction will be in the execute stage in the same clock cycle as the \texttt{beq} instruction is in the decode state. If the hazard needs to be resolved using forwarding or stalling depends on if the previous operation can be finished before the second instruction uses the data (within the same clock cycle). Since the information is not provided in the exercise, either solution would be counted as a valid solution, as long as a motivation is given. If it would have been solved by stalling, the \texttt{beq} instruction would take one more clock cycle.