Part I: Fundamentals

1. Module 1: Logic Design (for IS1500 only)

   (a) Short answer: Case 1: $R = 0011_2$. Case 2: $B = 0110_2$.

   Elaborated answer: Note that the circuit acts as an added if $S = 0$, and as a subtracter if $S = 1$. It is then easy to see that i) $R = A - B = 5 - 2 = 3 = 0011_2$ and ii) $A + B = R \implies B = R - A = 8 - 2 = 6 = 0110_2$

   Max 4 points. Two points for each correct answer.

   (b) Answer:

   For all cases, we show in a truth table that the values on the right hand side and left hand side are equal.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$(A \cdot B) + (A \cdot C)$</th>
<th>$A \cdot (B + C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

   Max 3 points. One point if there is a truth table with 8 cases for $A$, $B$, and $C$. Two points if the truth table is almost correct, with minor mistakes. 3 points if the truth table is correct.

   (c) The purpose of a register is to be able to store bits of information. A register can be seen as a simple memory.

   Max 1 point. To get one point, it is enough to say that a register is used for “storing information” or that it is a kind of “memory”.

   1
2. **Module 2: C and Assembly Programming**

(a) Statement #1:
*e = *e + v;

Statement #2:
*a++;

Max 4 points. 2 points for each correct statement. 1 point is removed from a statement if there is only a minor error.

(b) Instruction #1
`lw $t0,0($a0)`

Instruction #2
`addi $a0,$a0,4`

Please note that the pointer should be incremented with 4 because each integer is 4 bytes long.

Max 4 points. 2 points for each correct statement. 1 point is removed from a statement if there is only a minor error.

3. **Module 3: Processor Design**

(a) Short answer: The signal values are as follows:

\[ S_1 = 0x1 \]
\[ S_2 = 0x000000f0 \]
\[ S_3 = 0x00000000 \]

Elaborated answer: The first step is to decode the actual instruction. After decoding the instruction we see that the instruction is `and $t1,$t2,$t0`. Since this is an R-type instruction, the write enable (WE) signal \( S_1 \) must be 1. We can see that \( S_2 \) is the value that is read out from \( rs \), which in this case is \( $t2 \). Finally, signal \( S_3 \) is the output of performing bitwise \textit{and} on the values stored in \( $t2 \) and \( $t0 \).

Max 4 points. 1 points if \( S_1 \) is correct, 1 point if \( S_2 \) is correct, and 2 points if \( S_3 \) is correct.

(b) Short answer: Because the critical path for each clock cycle is shorter and the processor may therefore be clocked at a higher clock frequency.

Elaborated answer: Note, however, that cycles per instruction (CPI) can actually be larger for a pipelined processor, than for a single cycle processor.

Max 2 points. Two points if the answer clearly shows that the performance gain is due to clock frequency (or reduced clock period).

(c) The hazard is a control hazard and the misprediation can be detected in the decode stage.

Max 2 points. One point for control hazard and one point for stating that it is detected at the decode stage.
4. Module 4: Memory Hierarchy

(a) Short answer: The set number for \texttt{sw} is \(0x86\) and the set number for \texttt{lb} is \(0x87\). Elaborated answer: The number of sets are \(4096/16 = 256\), which means that the tag size is 8 bits. Since the block size is 16 bytes, the byte offset takes 4 bits. This means that we can directly read the set numbers from the address. The \texttt{sw} instruction has an offset of 4, which means that the instruction still reads from block \(0x86\). However, \texttt{lb} has an offset of 16 bytes, which means that it will read from the next block, that is, block \(0x87\).

Max 4 points. 2 points for each correct number.

(b) Short answer: The tag field is 22 bits, the set field 7 bits, and the byte offset 3 bits. Elaborated answer: The cache has 4 ways, each way with the size of \(4096/4 = 1024\) bytes. There are \(1024/8 = 128\) sets (rows), which means that the set field is 7 bits \((2^7 = 128)\). The block size is 8 bytes, which means that the byte offset is 3 bits. Finally, the tag size is \(32 - 7 - 3 = 22\) bits.

Max 3 points. 1 point for each correct number.

(c) Short answer: A Write-Through Policy.

Max 1 point. 1 points for correct answer.

5. Module 5: I/O Systems

(a) Short answer:
\begin{verbatim}
lui $t1,0x8000
addi $t1,$t1,0x20f0
addi $t0,$0,0x7c
sw $t0,0($t1)
\end{verbatim}

Elaborated answer: The address is 32-bit long. We therefore need two instructions \texttt{lui} and \texttt{addi} to write the address into a register. To display digit 6, the bits for indices 2-6 need to be set. Hence, the binary value that should be written is \texttt{1111100}, which is the same as the hexadecimal value \texttt{0x7c}. Alternatively, digit 6 can also be encoded as \texttt{1111101}, which is the same as the hexadecimal value \texttt{0x7d}. Both answers are treated as correct answers.

Max 4 points. Remove 1 point for each error in the code. More than 3 errors give zero points.

(b) If a pointer is not declared to be \texttt{volatile}, the optimization phase of the C compiler may remove the instruction that reads from the memory mapped I/O port.

Max 2 points. Two points if the answer clearly states that the problem is related to the compiler optimization.

(c) The main benefit of DMA is that data memory transfers can be performed at the same time as the processor is performing some other task.

Max 2 points. Two points if the answer states that DMA transfers can be performed while the processor is doing something else. One point if the answer only states that the DMA can perform memory transfers.
6. Module 6: Parallel Processors and Programs

(a) The main difference is that superscalar processors schedule instructions in parallel dynamically, whereas a compiler needs to schedule the instructions statically at compile time for a VLIW processor.
Max 2 points. 1 point if it is clear that superscalar processors schedule instructions dynamically in hardware. 1 point if it is clear that VLIW needs to schedule the instruction at compile time (by the compiler).

(b) Short answers:
   i. Multicore $\mapsto$ MIMD
   ii. VLIW (very long instruction word) $\mapsto$ ILP
   iii. Multimedia extension $\mapsto$ SIMD
   iv. Data-level parallelism $\mapsto$ SIMD
   v. Hardware multithreading $\mapsto$ Hiding latencies
   vi. Multiple issue (instructions) $\mapsto$ ILP
Max 3 points. Each item gives 0.5 points. Round off upwards. For instance, counting 1.5 points result in 2 points.

(c) Short answer: The total execution time on one core is 70ms.
Elaborated answer: Let $x$ be the execution time when the whole control program is executed on one core. We know that the decision task is sequential, since it takes always 10ms, regardless of the number of cores. Hence, the time for the parallel task is $x - 10$. If we insert the numbers in Amdahl’s law, we have
\[
4 = \frac{x}{(x - 10)/8 + 10}
\] (1)
After solving the equation, we have $x = 70$ milliseconds.
Max 3 points. Give 3 points if the answer is exactly correct, else zero points.
Part II: Advanced

7. • **Multicore processor with low clock frequency vs. unicore processor with high clock frequency.** A benefit with a unicore processor is that the sequential programs can execute fast, if the clock frequency is high. However, there is a limitation on the clock speed, often referred to as the power wall. If a task can be parallelized on a multicore processor, such a processor can be made more energy efficient, assuming that both the frequency and the voltage is lower. A major problem with multicore processors is how to program them efficiently. Multicores also have problems with caches, since it introduced a cache coherence problem that does not exist for unicores.

• **Processors with pipeline vs. processors without pipeline.** A processor without a pipeline has limitation of the clock frequency that it can be executed. The critical path can be very long. A pipelined processor, however, can reduce the critical path since several different parts of the pipeline can be done in parallel. For instance, one instruction can be fetched, while another instruction is decoded. Pipelined processors may, however lower the CPI. The main reason for this is pipeline hazards. A processor without a pipeline is significantly simpler than one with a pipeline. No hazard detection unit is needed for

• **RISC processors vs. CISC processor.** A RISC processor is simpler than a CISC processor. For instance, RISC processors typically have only a few instructions that can access the memory. This means that less hardware is typically needed for a simple RISC processor. However, for a RISC processor, more instructions need to execute to perform the same task as for a CISC processor. Many RISC ISAs have fixed length of an instruction, where CISC processors such as Intel x86 processors have variable length, which makes decoding complex.

• **Assembler programming vs. C programming.** Assembly programming is more low level and therefore more error prone. C programming makes programs more portable; the same C program can be executed on different platforms with different ISAs. Today’s optimized C compilers are extremely good at optimizing programs and to generate efficient machine code. Hence, in practice, engineers seldom write assembly programs.

• **SIMD instructions vs. ILP (instruction level parallelism).** SIMD instructions can be very efficient in applications that have data-level parallelism. The compiler needs, however, to be able to see the parallelism in the program and generate the correct SIMD instructions. Instruction level parallelism, e.g., in a superscalar processor, tries to fetch and execute instructions in parallel, automatically. There is, however, a fundamental limitation of the speedup: sequential programs tend to have a lot of dependencies between instructions. For multimedia applications, SIMD can still achieve considerable speedup in the future, when processors are offered with larger registers. ILP seem to have reached the limit already due to instruction dependencies.

Max 20 points. Give 2 points for each good and well motivated argument. Reduce points if the arguments are weak or if the solution includes incorrect arguments. Max 4 points for each item.
8. The processor needs to have a multi-level cache. The reason for not having just one big cache is that smaller caches tend to be faster than larger caches. However, since smaller caches can result in a higher miss rate, there is a clear tradeoff between the size of the cache and the performance. As a consequence, it is a good idea to have smaller caches closer to the memory and larger (slower) caches closer to the main memory. Also, since this is a multicore machine, it is a good idea to have a fast cache close to each processor. If the different processors are not in conflict with each other, each cache can work independently, thus resulting in high performance. In this case, we decided to have a two level cache (called L1 cache and L2 cache), where the L1 cache is local to each core, whereas the L2 cache is shared between the cores. Main cache properties: the L1 cache is 32KiB and both the code and the data is shared. The L2 cache is 256KiB and has a unified code and data in the cache. When code and data caches are shared, they can conflict with each other. But, on the other hand, both have more freedom to place data in a larger cache. We chose to have a approximate least recently used (LRU) replacement policy for both the caches. This is a common cache replacement policy and gives good performance in general. The L1 cache is 4-way (for both code and data) and the L2 cache is 8-way. We prefer to have a faster cache closer to the memory and therefore chose to have a lower associativity for the smaller cache.

The main problem we have with the cache for multicore is cache coherence. This means that the L1 caches may be inconsistent. We need to include a cache coherence protocol to solve this problem. One possibility is to use a snooping protocol. We assume that we have made some measurements and realized that 32 bytes is a good size for the cache block. This means that we have 256 rows (sets) and therefore is the set field of the address 8 bits. The byte offset field is 5 bits since the block size is 32 bytes. Hence, the tag must be $64 - 8 - 5 = 51$ bits. The actual cache memory stores, besides the cache lines, the tag and a valid bit. For each row, there are 51 bits for the tag, 1 bit for the valid bit, and 32 bytes for the cache block. Since there are 4 ways and 256 rows, there are in total $4 \times 256 = 1025$ bits of valid bits. There are also extra logic for handling the cache coherence protocol, but a detailed description if this is outside the scope of this exercise.

Max 9 points. Up to 5 points for the discussion about the cache design, including multi-level caches, replacement policies, cache properties, and problems with caches in a multicore setting. Up to 4 points for the description of the fields and the organization of the cache memory.

The two main kind of hazards that can occur are data hazards and control hazards. The code example below shows an example where we have data hazards.

The code example below shows an example where we have data hazards.

```assembly
lw   $t1, 0($t2)
addi $t2, $t1, 100
and  $t3, $t1, $t2
```

If we assume that the pipeline has a stage that writes back data to the register, which is loaded from the memory, we have a data hazard if the following instructions needs to read the data from the register before it is written. This is a read-after-write hazard. In the code example, the `addi` instruction uses `$t1` before it is written. Depending on the structure of the pipeline, data hazards can sometimes be solved.
using *forwarding*. This means that when the data is available in the pipeline for one instruction, it is directly forwarded to the next instruction, before it is even written to the register file. For instance, the result from the *addi* instruction can be forwarded to the *and* instruction. However, it is not always possible to solve data hazards using forwarding. In such cases, the pipeline needs to be stalled.

The second kind of hazard is a control hazard. The following MIPS code exemplifies this:

```mips
slt  $t1,$t2,$t3
beq  $t1,$0, skip
sub  $t1,$s1,$s2
...
skip:
    add  $t1,$s1,$s2
```

When the conditional branch instruction is fetched, several instructions have already been fetched. For instance, several of the instructions between the *beq* instruction and the *skip* label may already have been fetched and executed. A control hazard cannot be solved by forwarding. Instead, the processor needs to predict which path the execution will take, so that the processes does not have to stall the pipeline. This technique is called *branch prediction*. A simple static branch prediction strategy is predict branch not taken. However, for a modern superscalar processor, a dynamic *branch predictor* is recommended. In such a case, the hardware dynamically keeps track of previous branches and uses this information to predict if the branch will be taken or not.

Max 6 points. 3 points for a good explanation of data hazards, including code examples and forwarding. 3 points for a good explanation of control hazards, include code example and the use of branch predictors.
9. • A possible interpretation of the MIPS program is as follows:

```c
/* Example function that sorts array [a] of length [len] */
void foo(int *a, int len){
    int i; /* Variable in the inner loop */
    int swapped = 1; /* Boolean flag that says if two elements are swapped */
    while(swapped){ /* Loop until no more swapping */
        swapped = 0; /* Restart swap flag */
        for(i=0; i<len-1; i++) { /* Loop for len-1 elements */
            if(a[i+1] < a[i]){ /* Should we swap? */
                int tmp = a[i+1]; /* Yes, swap. */
                a[i+1] = a[i];
                a[i] = tmp;
                swapped = 1; /* Current iteration has swapped */
            }
        }
    }
}
```

Max 10 points. Assume that the solution is correct. Remove 2 points for each error in the code. If there are no comments at all, max 9 points are given.

• The function foo sorts an array of elements. The result is stored back in the same array. This is an implementation of the simple bubble sort algorithm.

Example: If the following lines of code are executed

```c
int t1[] = {3,5,7,1,9,10,2};
int len = sizeof(t1)/sizeof(int);
foo(t1, len);
```

then array t1 will contain values 1,2,3,5,7,9,10.

Max 5 points. Give 3 points if it is clearly stated that this is a sorting algorithm. Give 2 points if the example is clear.