Part I: Fundamentals

1. Module 1: C and Assembly Programming
   
   (a) Short answer: \texttt{addi \$t3,\$s0,-7}
   
   Max 3 points. 1 point for correct instruction (\texttt{addi}), 1 point if both operands are correct (\$t3 and \$s0), and finally one point if the immediate value is correct (\(-7\)).
   
   (b) Short answer: \(a = 2, \ b = 160, \ c = \text{unknown}, \ d = 30, \ 	ext{and} \ e = 0x7fff0028\).
   
   Comment: Note that \(c\) is \text{unknown} because \(d\) is not initiated to any value and can contain any value that happened to be in the memory when the program started.
   
   Max 5 points. One point for each correct variable.

2. Module 2: I/O Systems
   
   (a) Short answers:
   
   i. True.
   
   ii. False. A synchronous bus is using a common clock signal.
   
   iii. False. DMA is used to offload the processor, by automatically transferring data between different parts of the main memory. It has nothing to do with file access to the hard drive.
   
   iv. True.
   
   Max 4 points. One point for each correct answer. In the case of false, a short motivation must be given.
   
   (b) Short answer:
   
   \texttt{get_status:}
   
   \begin{align*}
   \text{lui} & \quad \$t0,0xabab \\
   \text{lw} & \quad \$v0,0x20(\$t0) \\
   \text{andi} & \quad \$v0,\$v0,0x4 \\
   \text{srl} & \quad \$v0,\$v0,0x4 \\
   \text{jr} & \quad \$ra
   \end{align*}
   
   Max 4 points. 1 point for correct handling of the address (the pseudo instruction \texttt{li} is also ok), 1 point for correct use of load word. One point for correct handling of \texttt{andi} and \texttt{srl}. One point for giving a label and correct return using \texttt{jr} and returning the value in \$v0.
3. **Module 3: Logic Design (for IS1500 only)**

(a) Short answer:

<table>
<thead>
<tr>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$Y_1$</th>
<th>$Y_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Max 4 points. One point for each correct row.

(b) Short answer: 2048 bytes.

Elaborated answer. The register file has a 10-bit address, which means that there are $2^{10} = 1024$ data items. The output is a 16-bit port, which means that it can store $1024 \times 2 = 2048$ bytes.

Max 2 points. Two points for a correct answer, otherwise zero points.

(c) Short answer:

i. False. A combinatorial circuit cannot have any memory, whereas sequential circuits can contain memory building blocks, such as registers.

ii. True.

Max 2 point. One point for each correct answer.
4. Module 4: Processor Design

(a) Short answers: `ori $a0,$s4,0x122`

Max 4 points. 1 point for `ori`, one point for each register, and one point for the immediate value.

(b) In this exercise, there are a few different alternative answers that are all counted as correct. One of the following answers would be counted as a correct solution:

i. There is only one data hazards between instructions `lw` and `sw` with the dependency in register `$t0`. This hazard is solved using both stalling and forwarding.

ii. There are read-after-write hazards both at the `sw` and at the `or` instructions. In both cases they use `$t0` before it is written to the register. The first hazards are solved using stalling and forwarding.

Max 4 points. 1 point for stating that it is a data hazard or a read-after-write (RAW) hazard. 1 point for stating that it is between `lw` and `sw` or that it is at both the `sw` and at the `or` instructions. 1 point for stating that it is due to `$t0`. One point for stating that it is solved by both stalling and forwarding.

Note that if the hazard at `sw` is solved using both stalling and forwarding, there is actually no hazard for `or` since the value has after the stalling been able to be written to the register (the first part of the cycle in the write stage writes and the second part of the decode stage reads).

5. Module 5: Memory Hierarchy

(a) Short answer: The tag field size is 21 bits, the set field 8 bits, and the byte offset field 3 bits.

Elaborated answer: The block size is \( \frac{4096}{512} = 8 \) bytes. Hence, the byte offset field size is 3 bits. There are \( \frac{512}{2} = 256 \) sets, which means that the set field size is 8 bits. Hence, the tag field is \( 32 - 8 - 3 = 21 \) bits.

Max 3 points. One point for each of the numbers.

(b) Short answer: The data cache hit rate is 40%. The instruction cache hit rate is \( \frac{10}{23} \).

Elaborated answer: The data cache hit rate is only affected when the load word instruction is executed. Due to the index -4 of the load instruction, the first load word starts in the first part of a block. Hence, the first two loads will be cache misses, followed by a hit. This is followed by a miss and then a hit. Hence, the hit rate is \( \frac{3}{5} = 0.4 \), that is, 40%.

The `lui` instruction starts at the beginning of an instruction cache block. There are in total \( 3 + 4 \times 5 = 23 \) memory accesses when fetching instructions. There are in total 4 instruction cache misses. Hence, the hit rate is \( \frac{10}{23} \).

Max 5 points. 3 points for correct data cache hit rate and 2 points for correct instruction cache hit rate.
6. Module 6: Parallel Processors and Programs

(a) Short answer: 75%

Elaborated answer. Using Amdahl’s law, we know that the theoretical speedup is achieved when we have infinitely many cores, that is, when the parallelizable part’s execution time goes to zero. Hence, the sequential part is \( \frac{20}{T} = 5 \text{s} \). This means that the sequential part is \( \frac{5}{20} = 25\% \). Hence, the parallel part is 75%.

(b) Short answer:

- True.
- False. ILP is very common in modern processors. It gives good performance gain, compared to not using ILP.
- False. What is described is VLIW. A superscalar processor performs the scheduling in hardware.
- True.
- True.

Max 5 points. 1 point for each correct answer, including motivation if the answer is false.

Note: The last bullet can also be seen as correct, if is stated as false with a valid motivation. For instance, if the argument is that mutex is a general concept, and does not have to be implemented using binary semaphores, it may also be seen as correct. Note however that binary semaphores is a common way to implement a mutex. Also, if the answer is false due to the fact that the exam had the typo “current” (see the exam), it will also be treated as correct.
Part II: Advanced

7.

Hi Kalle,

I am not really sure why you share your notes, but I took the time to go through them anyway. It seems like you have misunderstood some of the concepts completely. Please let me explain...

- Moore’s law has nothing to do with Moore machines. Moore’s law is about doubling the number of transistors every 18 to 24 months, whereas Moore machines are a special kinds of state machines where the outputs are not directly dependent on the inputs. Furthermore, combinatorial circuits cannot contain memory. However, both Moore and Mealy machines can include memory states.

- Yes, this is true. Pipelining is a technique to shorten the critical path to increase the clock frequency.

- This is really false because DRAMs are volatile: if the power is turned off, all data will disappear. It depends what you mean by expensive, but DRAMs are much cheaper than, for instance, SRAMs. It is very common to use DRAMs in multicore systems. Hence, the last statement is also false.

- This is wrong. Hardware threads are really done in the hardware and not in the operating system. You are mixing up hardware threads and software threads. Note also that software threads share virtual memory, but even software threads have separate stacks and registers.

- SIMD and MIMD are good ways to get speedup on a multicore processor, but there are other techniques. For instance: i) you can unroll loops and then get better use of ILP, and ii) if you divide computations into memory blocks, then you can sometimes utilize the cache more efficiently.

These are all my main comments. Good luck on the exam!

Max 15 points. Max 3 points for each good answer.
8. There are several possible solutions that can give points.

In the following, we create a temp buffer array on the stack (not the heap). We then first reverse the list and write it on the temp list, and then write back this list. Note that we have pointers to pointers, and we are therefore only moving pointers around.

```c
void reverse(char** list, int len){
    char* tmplist[len];
    int i;
    for(i=0; i<len; i++)
        tmplist[len-i-1] = *(list+i);
    for(i=0; i<len; i++)
        *(list+i) = tmplist[i];
}
```

As a second alternative, we do not create the buffer. Instead, we swap each pointer and we just iterate over half of the array in one direction. This second solution is a bit better because it uses less memory and fewer operations.

```c
void reverse(char** list, int len){
    char* tmp;
    int i = 0;
    int j = len-1;
    for(; i<j; i++,j--){
        tmp = list[i];
        list[i] = list[j];
        list[j] = tmp;
    }
}
```

Max 15 points. 5 points if the signature is correct, including the use of pointers to pointers. 5 points if pointers to pointers are in general used in a correct way (both dereferencing and []-syntax are OK). 5 more points if the semantics of the reverse logic are correct. For minor mistakes, 1 point is removed for each error.
9. The instruction cache has the capacity of 2048 bytes with a block size of 16 bytes. Since
the cache is directly mapped, the number of sets are \( \frac{2048}{16} = 128 \). Hence, there set field
is 7 bits, and the byte offset field is 4 bits. We have in total 11 instructions after the lui
instruction, which means that we will always use the same tag for the instruction cache.
Hence, for the instruction cache, no instruction will evict another instruction. The first
4 instruction memory accesses will result in instruction cache: miss, hit, hit, and miss.
The eight instruction accesses inside the loop will give hit, hit, hit, miss, hit, hit, hit, miss
(in the first iteration of the loop). In the rest 9 iterations, there will be 8 hits. Hence, the
number of instruction cache hits is: \( 2 + 6 + 9 \cdot 8 = 80 \). The total number of instruction
accesses is \( 4 + 10 \cdot 8 = 84 \). Hence, the instruction cache hit rate is \( \frac{80}{84} \).

For the data cache, we have the capacity of 512 bytes, with a block size of 8 bytes. This
means that we have \( \frac{512}{8} = 64 \) sets and a block size field of 3 bits. As a consequence, the
tag field is \( 32 - 6 - 3 = 23 \) bits. There are only two instructions that access the data
cache memory: the two lw-instructions. These instructions access using a base address
stored in register \( t2 \), which will have a sequence of values 0, 4, 8, 12, etc. Because of
the index values \( (0x8000 \text{ respectively } 0x0) \), the sequence of memory addresses for the
first lw-instruction is \( 0x8000, 0x8004, 0x800c, \text{etc.} \) and \( 0x0, 0x4, 0xc, \text{etc.} \) for the
second lw-instruction. We can now see that both instructions will use the same set, but
with different tags. Because we have a direct mapped cache, the second lw-instruction
will always evict the cache block. As a consequence, the next time the first lw-instruction
is executed in the next iteration of the loop, we will have a cache miss again. Hence, there
will be cache misses for both lw-instructions, which means that we will get \( 0\% \) data
cache hits.

To calculate the CPI (cycles per instruction), we need to calculate both the number of
execution cycles and the number of instructions executed in the program. The number
of instructions can be calculated as follows: \( 4 + 8 * 10 = 84 \) because there are 4
instructions before the loop, 8 instructions within the loop, and the loop executes 10 times.
There are data hazards between several instructions, e.g. between the sll-instruction and
the first add-instruction and between the last two add instructions, but all of them can be
solved using forwarding, so there is no penalty. However, there is a data hazard between
the second lw-instruction and the second add-instruction that needs to be solved using
forwarding and stalling of one clock cycle. We assume the policy of branch not taken.
For the first 9 iterations, there is also one penalty for taking the branch (jumping at bne),
but in the last iteration, the branch is not taken, i.e., no penalty. Note, however, that
we will have a data hazard that needs forwarding and one clock cycle stalling between
the addi and the bne instruction because the comparison for branching is done in the
declare stage. This penalty will occur in every iteration. In summary, the number of clock
cycles (without cache penalties) is \( 4 + 8 \cdot 10 + 2 \cdot 10 + 1 \cdot 9 = 113 \) cycles. We assume that
the data cache and the instruction cache are independent and do not interfere with each
other. There are in total \( 84 - 80 = 4 \) instruction cache misses, which results in \( 4 \cdot 10 = 40 \)
extra clock cycles for instruction cache misses. For the data cache, we had 100% cache
misses, which means we have \( 10 \cdot 2 \cdot 10 = 200 \) clock cycle penalty. The total number of
clock cycles is: \( 113 + 40 + 200 = 353 \). As a consequence, the CPI = \( \frac{353}{84} \).

In summary, the instruction cache hit rate is \( \frac{80}{84} \), the data cache hit rate \( 0\% \), and the CPI is
\( \frac{353}{84} \).
Max 20 points. Max 5 points for calculating the instruction cache hit rate, max 5 points for the data cache hit rate, and max 10 points for the CPI.

For the CPI, we will also treat the answer $\frac{354}{84}$ (or $\frac{177}{42}$) as correct solutions, with the motivation that the last instruction (bne) that is executed also needs to execute the decode stage to be sure that the branch is indeed not taken.