This exam has nine (9) problems. You should submit your answers to six (6) of these nine problems. You should not submit answers for the remaining three problems. Since all problems are equally weighted (carry the same number of points), carefully choose which six problems you will answer.

Write your answers/solutions clearly and legibly. Illegible answers will be treated as wrong answers. You should clearly mark on top of each page of your answers which problem the page corresponds to, and you should not use the same page for two problems (it’s fine to use multiple page per problem if you need to). Although there is no restriction to how long your answer should be, try to keep your answers and solutions short and to the point.

Good luck!
**Problem 1: Dynamic Instruction Scheduling**

**Dynamic instruction scheduling** is one of the core components in modern, high-performance main-stream microarchitectures. Even though we normally think of it as one big "scheduler" black box, there are actually several critical and interacting components consisting of the wakeup logic, the select logic, and relevant equivalents for handling memory instructions.

**The regular instruction scheduler:** The latest Intel Core 2 processor has a highly optimized divider unit that can terminate its computation early when the inputs have reduced precision. For example, dividing zero by anything will result in a very low latency operation, dividing two small integers may take a little longer but still be relatively quick, and dividing two double-precision floating point numbers (e.g., 1.0/π) may take several tens of cycles.

**Part (A):** Given that instruction scheduling and instruction execution are decoupled processes (scheduling happens several cycles before an instruction actually reaches the ALU), what are the complications compared to a traditional fixed-latency divide unit? What changes, if any, are required of the instruction scheduling logic to handle such a division unit? Explain and justify your answer, giving an example if necessary.  
*Expected Answer Length: Approx. two or three full paragraphs.*

**The memory scheduler (LSQ):** Except for the Alpha 21264 and the Core 2 microarchitectures, most prior processors did not support the out-of-order execution of load instructions in the presence of earlier unresolved store addresses. The Core 2 processor brought back this idea to improve performance.

**Part (B):** Compared to the simpler memory schedulers, what changes are necessary to the LSQ to support speculative disambiguation of load instructions. In particular, discuss the changes needed to enable the actual speculative execution, decision of whether-to-speculate-or-not, and what hardware is necessary to detect misspeculations. By “discuss”, include explanations of why the conventional (non-speculative) LSQ logic cannot handle these functions, and how the modified structures work with each other; use an example if appropriate.  
*Expected Answer Length: Approx. two or three short paragraphs.*

**Part (C):** Speculative memory disambiguation can increase performance by exposing more instruction level parallelism, but as you described in part (a), there are quite a few additional hardware structures that need to be augmented to support this. All of these extra structures require more power, and any misspeculations result in pipeline flush which consume yet more power. Given that modern processor architects (including those for the Core 2) are painfully aware of power efficiency these days, how do you justify the extra power cost to support this technique? Provide a compelling argument that Intel either did or did not do the right thing (choose only one position). Specifically address the issues of performance, power, cost, and the fact that Intel’s products target multiple market segments (i.e., from low-cost/budget CPUs all of the way to expensive server parts).  
*Expected Answer Length: two-to-four paragraphs.*
Problem 2: Support for Virtualization

Intel (VTx), AMD (Pacifica), and even ARM have recently added processor extensions for more efficient software virtualization. These extensions include many changes, but they always add support for virtual page table lookups, and more efficient system calls.

**Part (A):** Why are page table lookups slow in virtual machines? Describe the hardware changes needed to ameliorate this problem.

**Part (B):** Why are system calls slow in virtual machines? Describe the hardware/software changes needed to ameliorate this problem.

**Part (C):** List another significant overhead imposed by virtual machines, and describe a possible hardware addition to make it more efficient (we are looking for a high-level answer here; along the lines of "VMs create a lot of additional divisions, so you should increase the number of function units with divide, and possibly issue width", although this answer is clearly wrong).
Problem 3: Prefetching

For a processor that has a finite instruction window size that is greater than 10, discuss benefits of employing a prefetcher when a processor has

- **Part (A):** a long memory latency with infinite memory bandwidth.
- **Part (B):** a 10 cycle memory latency with finite memory bandwidth.

Suppose that processor A employs a stream prefetcher and processor B has a Markov prefetcher. Both stream and Markov prefetchers provide 10% performance benefit relative to a processor that does not employ any prefetcher. However, if cache size is doubled, the stream prefetcher still provides 10% performance benefit but the Markov prefetcher does not provide any performance benefit.

- **Part (C):** Discuss why the Markov prefetcher might not provide any benefit.

- **Part (D):** Discuss why the stream prefetcher could still provide 10% benefit.

- **Part (E):** What kind of applications would show this behavior?
**Problem 4: Sharing of Caches in Multi-Cores**

In multicore architectures such as Intel's Core Two Duo, shared L2 cache management between two cores is a major challenge. Techniques such as cache resizing are being used in practice to divide the cache between two cores competing for the space.

**Part (A):** Enumerate possible schemes (other than resizing) for effective L2 sharing between two cores, discussing pros and cons of each. Please give your insight into your motivation behind each design (based on expected loads and stores going into L2).

**Part (B):** Pick one of the schemes and devise a detailed solution.

**Part (C):** Energy efficiency of caches is a major issue and is likely to dominate the design you proposed in Part (B), how will you make the proposed design energy-efficient?

**Part (D):** Compilers and knowledge of program characteristics can play a significant role in cache management. This could be especially useful in a shared setting which also needs to be energy efficient. Devise a scheme which will benefit from compiler analysis of programs in which cache management hints are generated and conveyed by the program to the cache management logic. Outline what kind of compiler analysis you may utilize/devise.
Problem 5: Support for Virtual Memory

Part (A): Give examples from the past of machines that did not support virtual memory and why they chose not to.

Part (B): Provide arguments for and arguments against supporting virtual memory in the architecture.

Part (C): Are such arguments still valid in the context of today's application and technology trends? If yes, why? If no, why not?
Problem 6: Static Instruction Scheduling

The following code has only pure dependencies.

I1: LDR R10,R6,#0
I2: LDR R11,R10,#1
I3: MUL R12,R6,#2
I4: ADD R13,R11,#10
I5: STR R13,R10,#0
I6: ADD R14,R11,#1
I7: STR R14,R6,#0

Part (A): Construct the data dependence graph for this code.

Part (B): Find the depth, height and slack for each instruction in the code. Assume: LDR latency is 2 cycles (pipelined), MUL latency is 3 cycles (pipelined), all other latencies are 1 cycle, and all source registers are read in any cycle before destination registers are written.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Depth</th>
<th>Height</th>
<th>Slack</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Part (C): Schedule the code for the following VLIW format. Perform cycle-based list scheduling. Assume the priority list order is (from highest to lowest-priority): A, B, C, D, E, F, G. It is sufficient to write the letter A-G to indicate the instruction. You may not need all rows in the table. Assume that all source registers are read in any cycle before destination registers are written.

<table>
<thead>
<tr>
<th></th>
<th>ADD</th>
<th>LDR/STR</th>
<th>MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 7: Speculative Lock Elision

Speculative lock elision (SLE) allows a thread to speculatively continue into a critical section where it would ordinarily have to wait.

Part (A): How does SLE affect power and energy consumption of the system? Specifically, discuss how (and if) SLE can increase power and/or energy consumption and how (and if) SLE can decrease power and/or energy consumption.

Part (B): The SLE hardware recognizes lock spin-waiting code automatically. What are the implications for writing synchronization libraries and operating system code?

Part (C): Discuss which aspects of SLE can be applied to barrier synchronization.
Problem 8: Reliability of Memory Protection Hardware

Mondrian Memory Protection (MMP) can be used to replace existing page-based protection mechanisms and achieve memory protection at word granularity. However, MMP requires more protection state to be kept and maintained.

**Part (A):** How would you compute the Architectural Vulnerability Factor (AVF) of existing page-based protection hardware. In particular, discuss the AVF of protection bits in the page table, and the TLB. How much do these structures affect the overall reliability of the processor?

**Part (B):** Estimate the AVF of MMP hardware. How much does this hardware affect the reliability of the processor?

**Part (C):** Assuming that the reliability of the memory protection mechanism is important, how would you improve this reliability for existing page-based protection and for MMP?
Problem 9: Multi-Core Coherence and Consistency

Early multi-core processors use a private L1 and L2 cache for each processor, then connect L2 caches via a shared bus that also connects them to the system bus and the memory. Some newer multi-core processors connect L1 caches to a shared bus, which also connects them to a shared L2 cache. It is expected that future processors will have an L1 cache and a small part of the shared L2 cache next to each processor, forming a “tile” these tiles would then be connected via an on-chip switched network (e.g. a torus).

Part (A): Why did processors designers switch form private L2 caches to a shared L2 cache? Why are they expected to switch to a distributed L2 cache in the future?

The following questions assume that we use a distributed shared L2 cache in a tiled architecture, and that a fixed function is used by the hardware to map the block’s physical address to the tile in whose portion of the L2 cache the block is going to be placed.

Part (B): How would you design this address-to-tile mapping function?

Part (C): A processor can quickly access its own part of the distributed shared L2 cache, but accesses to other parts of the L2 cache will take much more time. Is there any way we can migrate blocks closer to tiles that use them often, without changing the hardware that maps the blocks to tiles based on the block’s address?