Only Problem Set Two will be graded. **Turn in only Problem Set Two** on December 2, 2016 (Friday) by 5:00 pm.

Submit homework via gradescope <gradescope.com> as a PDF. Other formats may or may not be viewed and evaluated accurately.

**Problem Set One**

1. Hennessy & Patterson (5th Ed) B.1
2. Hennessy & Patterson (5th Ed) B.3
3. Hennessy & Patterson (5th Ed) B.9
4. Hennessy & Patterson (5th Ed) B.10
5. Hennessy & Patterson (5th Ed) 2.2
6. Hennessy & Patterson (5th Ed) 2.4
7. Hennessy & Patterson (5th Ed) 2.9
8. Hennessy & Patterson (5th Ed) 2.11
9. Hennessy & Patterson (5th Ed) 2.12
Problem Set Two

1 (VLIW Global Code Scheduling)

This question probes your understanding of VLIW machines, a statically issued, statically scheduled superscalar machine, particularly useful for DSP applications. VLIW machines as you remember have a number of instructions in a very long instruction word all executed in parallel. The whole long instruction word stalls simultaneously if any of its parts need to stall in the pure VLIW implementation, a position which we will adopt here.

One of the shortcomings of a VLIW architecture is that finding consistently the number of instructions (and in the distribution dictated by the specific architecture) may prove to be challenging. This difficulty is exacerbated by the incidence of branches which splinter the basic blocks, reducing the number of instructions at play to fill in the bandwidth of the VLIW instruction. Global Code scheduling (i.e., moving instructions out of and into control dependent status) has been proposed to increase the number of instructions in a basic block, thus increasing the occupation load of VLIW instructions.

The main bulk of this question consists of you analyzing a piece of code composed of the \textit{then} and \textit{else} part of a conditional and preceded by some code prior to the fork point of the branch and some code following the join part of the branch. You will be asked to fill in the VLIW instructions corresponding to this code under various sets of assumptions. Throughout you will assume a 3 instruction “MIW” (Medium Instruction Word) so that you can focus on the relevant concepts rather than filling in numerous instructions of the conventional VLIW. The 3 instructions supported by this MIW correspond to an Integer Arithmetic, FP Arithmetic, and a LD/ST/BR instruction in that order. The branch that you are given has been profiled and is known to execute the \textit{then} part of the branch with 65\% frequency. Furthermore, there is 1 available free register (either floating point or integer), R12, that can be used if needed.

We are replicating this code at the end of the test in case an extra copy may prove useful.

\begin{verbatim}
PRE-BRANCH:
ADD R1, R2, R2 / FSUB R5, R5, #3 /
MUL R1, R1, #3 / FMUL R6, R5, #2 /
SUB R3, R4, R3 / /
DIV R1, R1, R3 / / BNE R3, R6, ELSE

THEN:
ADD R1, R1, #1 / FSUB R8, R5, R6 /
 / FMUL R7, R6, #3 /
 / FMUL R10, R6, #4 / JMP JOIN

ELSE:
ADD R1, R1, #2 / FADD R7, R6, #1 /
 / FSUB R8, R8, R7 /
 / FADD R9, R5, R8 /

JOIN:
ADD R2, R1, R2 / /
ADD R2, R3, #4 / FDIV R7, R5, R10 /
SUB R2, R4, #1 / FADD R10, R10, #2 /
\end{verbatim}
(Part A) Please schedule this code so that you minimize the number of instructions and maximize the overall performance by moving instructions out of control dependent status. No compensation code can be used. Please let us know the scheduled code, the total number of MIW instructions, and the average execution time in terms of MIW instructions.

(Part B) Please schedule this code so that you minimize the number of instructions and maximize the overall performance by moving instructions out of control dependent status. Please use compensation code if it will improve performance (even if it increases code size). Please let us know the scheduled code, the total number of MIW instructions, and the average execution time in terms of MIW instructions.
(Part C) You remember a canceling instruction to be an instruction that can be canceled prior to commitment upon predication. Your MIW architecture can accommodate any one of the three instructions in the MIW to be designated as cancelable through a 2-bit indicator denoting the location of the cancelable instruction (with the combination “00” denoting none of the instructions in the MIW is cancelable).

Please schedule this code so that you minimize the number of instructions and maximize the overall performance by moving instructions out of control dependent status. No compensation code can be used. You can use cancelable instructions to improve performance and code size over (Part A). Please let us know the scheduled code, the total number of MIW instructions, and the average execution time in terms of MIW instructions.

(Part D) Please schedule this code so that you minimize the number of instructions and maximize the overall performance by moving instructions into control dependent status. No compensation code can be used. Please let us know the scheduled code, the total number of MIW instructions, and the average execution time in terms of MIW instructions.
Throughout the quarter we have been examining largely architectural techniques for making a processor more efficient. Occasionally, a collaboration of the software aspects with architectural hardware components has delivered improved results; examples of these we saw with branch delay slot filling and more recently when we discussed statically scheduled, dynamically executed super scalar machines. It is time to see if we can add onto such collaborations in the hope of improving the efficiency of processors.

As you remember from our lengthy discussions regarding out-of-order execution with particular emphasis on Tomasulo’s algorithm, memory disambiguation is typically effected by having Loads check preceding Stores for effective address matches, while ensuring that effective address computations are taking place in order. This technique while capable of ensuring semantic correctness, can nonetheless induce a significant pipeline slowdown as Loads may need to be delayed unnecessarily, leading to both an increased RAW frequency and also an increased occupation of the Load queue slots.

Yet as our analysis of the beginning of Chapter 3 has indicated, a number of loops can be analyzed to determine whether any dependencies hold that can preclude the carefree execution of load instructions. This question examines your understanding of memory access disambiguation techniques in an out-of-order Tomasulo’s execution pipeline, as well as your understanding of static techniques for analyzing loop behavior.

You will be given in this question a set of loops listed below executed in increasing index order. Feel free to restructure the loops as needed in order to deliver the performance speedups enabled by the software and hardware collaboration described earlier. Please assume throughout a Tomasulo’s pipeline with 3 Load and 3 Store slot queues. You can assume as many reservation stations and functional units as you would like to ensure that they don’t stall the pipeline.

```plaintext
for (i = 0 to 99) x[i] = x[i+4];

for (i = 0 to 99) x[i+5] = x[i]*3;

for (i = 0 to 99) x[2*i+1] = x[4*i] + 23;

for (i = 0 to 99) x[4*i] = x[2*i+16] + 47;

for (i = 0 to 99) {
    x[i] = z[i] + 3;
    z[i] = x[i];
    x[i] = -x[i];
}

for (i = 0 to 99) {
    x[i+3] = y[i];
    z[i] = x[i+4];
}
```
(Part A) Do any of the loops display loop-carried dependencies? For the ones that you think they do, can you restructure them (by doing instruction retiming as we discussed in class) to convert them to a form that no longer exhibits a loop carried dependency. For the ones that still exhibit a loop-carried dependency, identify the type of dependency (i.e., RAW, WAR, WAW) or combination thereof.

(Part B) Obviously, the loops that display no loop-carried dependencies are perfectly capable of being unrolled and fully parallelized by posting the Loads early and the Stores late to ensure that latency stalls are alleviated. For the ones that you think do exhibit loop-carried dependencies, can you do partial unrolling? If so, what is the maximum number of loops that you can unroll without the movement of Loads and Stores generating hazards capable of violating semantics?
(Part C) One technique for making the loops execute more efficiently is to simply hint to the pipeline at the beginning of a particular loop that this loop will still preserve semantic correctness even if no checking of preceding store effective addresses are effected for loads. Looking back at all the loops listed in the preamble, which ones do you think would benefit and which ones would result in semantic violations by employing such a technique.

(Part D) For the loops that you thought could not benefit from the technique discussed in (Part C), perhaps an alternate idea can be entertained to ensure that they can at least partially benefit. Perhaps parts of the loop need to complement conventional out-of-order Tomasulo machine memory disambiguation while other parts can be safely executed without it (i.e., by utilizing the hinting mechanism discussed in (Part C)). Can you fragment any of the loops that you were unable to speed up in (Part C) by fragmenting them so that a maximum number of load instructions can benefit from loads executing without store disambiguation? Can you show us the decomposed loops and which parts would thus benefit by rewriting the relevant loops?
Simultaneous multi-threading takes advantage of the fact that executing different threads provides *guaranteed* ILP without having to check for dependencies between instructions from different threads. However, SMT also raises the chances of unpredictably hurting the performance of any one thread. One of the ways in which a thread’s performance might be hurt by SMT is caching behavior.

Examine the following two blocks of code representing two threads executing simultaneously on a processor via SMT. Assume the word address of A = 0x0006 and word address of B = 0x1004. *Note that instruction addresses are also word addresses.* Instruction and Data caches each have 64 direct-mapped blocks, with 2 words/block (a word is 4 bytes). The data cache is also write-allocate. *Assume the two threads run in lock-step* (that is, both loops step to the next iteration at the same time).

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i=1 to 10000</td>
<td>for i=1 to 10000</td>
</tr>
<tr>
<td>A[i] = A[i]*10</td>
<td>x = B[i]</td>
</tr>
<tr>
<td></td>
<td>y = B[i+2]</td>
</tr>
<tr>
<td></td>
<td>z = B[i+128]</td>
</tr>
<tr>
<td></td>
<td>B[i-1] = x+y+z</td>
</tr>
</tbody>
</table>

*Instr. Addresses* 0x2000-0x2004 0x3280-0x3287

*(Note that 32 = 0x20, 64 = 0x40, 128 = 0x80)*

**Part A** Identify a specific cache access pattern that could result in a conflict of each of the following types:

1. Instruction Cache conflict between Thread 1 and 2.
2. Data Cache conflict where Thread 2 hurts Thread 1.
3. Data Cache conflict where Thread 1 hurts Thread 2.
(Part B) While you are scratching your head over how to solve your SMT problems, you stumble across a research paper that proposes a **hybrid-associativity cache**. The usual trade-off between lower and higher associativity is that higher associativity results in fewer conflict misses, but hurts hit time due to the added complexity. However, there is also the potential for a scenario where, among the cache lines aliased when moving to higher associativity, one of them was prone to conflicts but the other had an extremely high hit rate and aliasing them could cause everything to now conflict!

The proposed hybrid-associativity cache uses some complicated hardware that allows the associativity of the cache to be specified as 1-way (direct-mapped), 2-way or 4-way based on the thread accessing it. While the complexity of the hardware means that lower associativity does not benefit from a reduction in hit time, the access pattern of the threads themselves may dictate choosing one or the other. Note that, when increasing the number of ways in the cache, the cache is split into two and the top and bottom halves are used as the different ways (they are not interleaved!)

1. If Thread 2 were running alone, what associativity would it perform best with? Explain your reasoning.

2. If Thread 2 uses the hybrid cache as a direct-mapped cache, what should Thread 1 use the cache as? Explain your reasoning.

3. If Thread 1 uses the hybrid cache as a direct-mapped cache, what should Thread 2 use the cache as? Explain your reasoning.

4. Which of strategies 2 and 3 is better? *(Use this answer in Part D)*
(Part C) Of course, while you are playing games with the cache associativity, the question of how this cache is indexed comes up. The addresses in both threads are virtual addresses, but you need to choose between Virtually-Indexed, Physically Tagged (VIPT) and Virtually-Indexed, Virtually Tagged (VIVT) Caches. Before you move further on this question though, your boss lets you know that the company has had bad experience with adding process-id bits or any other such complication, and you should choose merely based on cache hit rate (not hit time). What are the advantages of a VIPT cache over a VIVT one or vice versa when it comes to a cache shared between multiple threads of execution?

(Part D) Now that you have made a decision on this tagging business, you are almost wrapped up with your end of the bargain for this SMT machine. However, the embedded system guys, who are designing the virtual memory system, come to you with one final question: what should the page size be? You realize at that moment that, depending on the size of the page, the address translation costs can be completely mitigated and make cache accesses much faster. However, you remember you are using a hybrid-associativity cache, and you have chosen different degrees of associativity for the two threads that will run on this processor. Using your answer from Part B, what should the minimum page size be in order to bypass address translation completely?
3 ((Cache Behaviors))

Assume two different cache organizations: (a) Direct-mapped cache and (b) 2-way set associative cache with an LRU replacement policy, both with 1 word/block and a 6-bit index.

(Part A) For the following unfinished code segment with a series of incomplete read references from arrays, please complete the code by filling in the index portion of the array references to minimize the cache reuse possibility (maximize cache miss rate) for both cache organizations described above. Please also give a brief explanation for your code behaviors. Assume the starting addresses of A[], B[] and C[] arrays in memory are 0, 520 and 144 (byte address), respectively.

(a) Direct mapped cache:

```c
int A[128], B[128], C[128]; // 4 byte integers
for i=0,116
    A[i+  ]
    A[i+  ]
    B[i+  ]
    B[i+  ]
    C[i+  ]
end
```

(b) 2-way set associative cache:

```c
int A[128], B[128], C[128]; // 4 byte integers
for i=0,116
    A[i+  ]
    A[i+  ]
    B[i+  ]
    B[i+  ]
    C[i+  ]
end
```
(Part B) We are now considering a 2-way set associative cache with an LRU replacement policy and also interested in incorporating a way-prediction hardware with a 2-bit saturating counter for each set. For the similar code segments presented in (Part A), please complete the code by filling the index portion of the array references along with the proper initial conditions of the way predictor to make the way prediction continuously wrong and also give a brief explanation for your code behaviors and way-prediction behaviors as well. We are giving you below the FSM for a 2 bit history way predictor, which is identical to the one we studied in class.

```
int A[128], B[128], C[128]; // 4 byte integers

for i=0,116
    A[i+  ]
    B[i+  ]
    A[i+  ]
    B[i+  ]
    C[i+  ]
end
```
(Part C) We now come back to the direct mapped cache, with 1 word/block but try to utilize software prefetching schemes to enhance the cache behavior. For the following Control Flow Graph (CFG), which consists of 6 basic blocks, all the memory references within the code fragments are shown in the CFG. The CFG shows the T/NT probabilities along with the cycle times between consecutive basic blocks, generated by static profiling, of both branches. Assume the starting addresses of integer arrays A[] and B[] are 0 and 520 (byte address). On a read cache miss, the memory would need to be accessed, which takes an additional 30 cycles. On the other hand, a write cache miss will not block the processor.

![Control Flow Graph](image-url)

We now have a total of four different load instructions from the basic blocks B2, B3, B4 and B5 which can be prefetched. Please rank the 4 different prefetching solutions insofar as the miss penalty is concerned.