CS 250 Spring 2017 Homework 08 Grading Guide
Please enter scores in Blackboard by April 14, 2017

Maximum points = 15

1. **[3 points]** Let store indirect be defined as

   ```
   STOREI (regX, offset, regY)      # store the contents of regX into the memory
                                    # location pointed to by the address found in
                                    # memory at location offset+regY
   ```

   Devise a way to implement STOREI as a macro written in the instruction set architecture (ISA) of Figure 6.1. For full credit, your implementation of STOREI may use no more storage locations and no different storage locations that does a single store instruction and must be expressed in the form of a macro.

   Answer:

   ```
   macro STOREI (regX, offset, regY)  # store regX into the address found in memory at
                                        location offset+regY
     load regY, offset + regY       # Load the indirect address from memory at location
                                      offset + regY but do not place in regX because that is
                                      holding the value to be stored
     store regX, 0 + regY           # Using the unaltered (because offset = 0) indirect address,
                                      write the value to memory at the indirect address location
   endmacro
   ```

2. **[3 pts. for part (b)]** Assume that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as a percentage of the execution time *when the enhanced mode is in use*. Recall that Amdahl's Law depends on the fraction of the original, *unenhanced* execution time that could make use of the enhanced mode. Thus, we cannot directly use this 50% measurement to compute speedup with Amdahl's Law.
   a. What is the speedup we have obtained from fast mode?
      Answer: In this scenario the execution time when using the enhancement is comprised of two parts: 50% of the time is using the unenhanced mode and 50% of the time is using the unenhanced mode, which is a factor of 10 less time than would be used if not for the enhancement. Thus,
      $\text{Speedup}_{\text{overall}}$ = Execution $\text{time}_{\text{original}}$ / Execution $\text{time}_{\text{enhanced}}$
      = (50% + 50%*10) / (50% + 50%) = 5.5/ (0.5 + 0.5) = 5.5
   b. What percentage of the original execution time has been converted to fast mode?
      Answer: The percentage of the original time that was converted to the fast mode is
      Percent converted = Portion converted / Total execution time
      = 50%*10 / (50%+50%*10) = 5/5.5 = 0.91 = 91%

3. **[3 pts. total. 1 point each for a completely correct set of answers for one byte addresx; grade for the byte addresses shown in gray highlighting in the table.]**
   Assume that a computer has a physical memory organized into 64-bit words. Give the word address and offset within the word for each of the following byte addresses: 0, 9, 27, 31, 120,

and 256.
Answer: Words are aligned in memory, these words are 64-bits or 8 bytes in size, so words will start on addresses that are a multiple of 8. The offset (byte position) within a word will be the remainder after the byte address is divided by 8. Therefore,

Given byte address	Corresponding word address is Quotient(Byte address, 8)	Corresponding offset, or byte position, within the word is Remainder(Byte address, 8)
0	0	0
9	1	1
27	3	3
31	3	7
120	15	0
256	32	0

4. Consider the following code snippet and Figure 11.6 from our text.

```
for (i = 0; i <= 1023; i = i + j) {
        array[i] =  array[i] + 7;
}
```

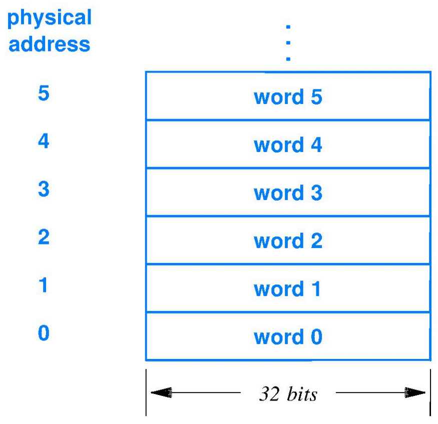

Figure 11.6 Physical memory addressing on a computer where a word is thirty-two bits. We think of the memory as an array of words.

Assume that the Figure 11.6 memory is built from a single memory chip that can be read during one CPU clock cycle but then requires 3 more CPU clock cycles before the chip is again ready to be accessed by the CPU. Assume that the constant 7 in the code snippet and the current values of variables i and j are available in CPU registers at all times. Finally, assume that the CPU stalls until array[i] can be read from data memory.

a. Describe the stalls that the CPU will experience due only to data memory read access of elements of array[i] as a function of the value of j for 1 <= j <= 1024. Ignore data memory write accesses; ignore all instruction fetch stalls (such as one due to a control hazard). With respect to execution time, what are the best values of j and the worst

values?
Answer: Because all data memory accesses will be made to the same chip, the nature of the stalls produced by the memory and experienced by the CPU will be invariant with respect to the value of j, only the total number of stalls will change because the number of array elements accessed does depend on j. Specifically, each new array[i] access will require 3 clock cycles of stall before the next access, array[i+j], may proceed. The total number of array elements is 1024 (indexed from 0 to 1023) and j sets the "skip" amount. As we get close to the end of the array if the ratio of the (number of elements beyond our current index) to (j) is less than 1, then on our next "skip" we will skip past the end of the array, and the loop will end along with those pesky data memory access stalls. Thus, the total number of array elements accessed as a function of j is floor(1024/j). So, the CPU will experience 3 * (floor(1024/j) -1) stalls, where the minus 1 term is because the very first access does not stall.

b. Now assume all is the same as in part (a) of this question except that the memory system configuration is now as shown in Figure 11.13.

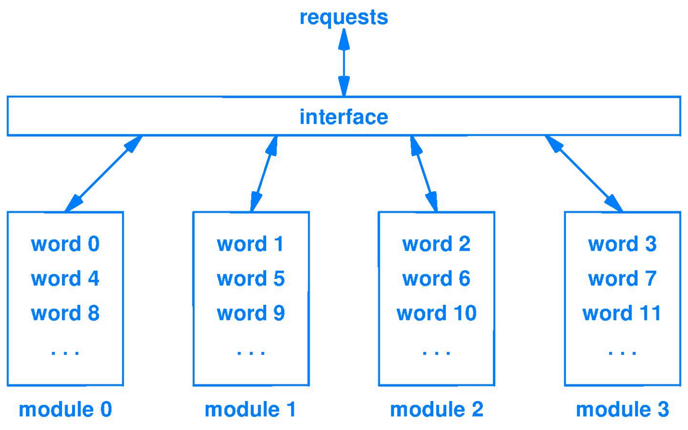

Figure 11.13 Illustration of 4-way interleaving that illustrates how successive words of memory are placed into memory modules to optimize performance.

Describe the stall cycles that the CPU will experience due only to data memory read access of elements of array[i] as a function of the value of j for 1 <= j <= 1024. Ignore data memory write accesses; ignore all instruction fetch stalls, such as one due to a control hazard. With respect to execution time what are the best values of j and the worst values?
Answer: With memory chips that achieve 1 cycle access and 3 cycle "recovery time," a 4-module interleaved memory can, by accessing module 0 then 1 then 2 then 3 and then 0 again in a repeating pattern, support continuous access to sequential memory addresses without any stalling. Now, however, as the "skip" amount between accessed array elements changes as a function of the value of j, the modules that actually are accessed need not be all four. If fewer than 4 modules are accessed there is no way to hide all 3 cycles of recovery time latency. The following table sets forth the action for each j value. In the table, cc means "clock cycle," "s=x" means stalls

for x clock cycles before this memory read can be performed.

j	Module 0	Module 1	Module 2	Module 3	**Stalls**
j=1	i=0,cc=0,s=0	i=1,cc=1,s=0	i=2,cc=2,s=0	i=3,cc=3,s=0	**0**
	i=4,cc=4,s=0	i=5,cc=5,s=0	i=6,cc=6,s=0	…	
j=2	i=0,cc=0,s=0	skip module (not accessed)	i=2,cc=1,s=0	skip module (not accessed)	
	i=4,cc=2,**s=2** (stall until cc=4)	skip module (not accessed)	i=6,cc=5,s=0	skip module (not accessed)	**2**
	i=8,cc=6,**s=2** (stall until cc=8)	skip module (not accessed)	i=10,cc=9,s=0	… *(Uses only 2 modules: 0 & 1.)*	
j=3	i=0,cc=0,s=0	i=9,cc=3,s=0	i=6,cc=2,s=0	i=3,cc=1,s=0	
	i=12,cc=4,s=0	i=21,cc=7,s=0	i=18,cc=6,s=0	i=15,cc=5,s=0	**0**
	… *(Accesses for*	*j=3 are a permu*	*-tation of the*	*j=1 pattern.)*	
j=4	i=0,cc=0,s=0	skip module	skip module	skip module	
	i=4,cc=1,**s=3** (stall until cc=4)	skip module	skip module	skip module	**3**
	i=8,cc=5,**s=3**	skip module	skip module	skip module	
	… *(Uses only*	*one of the four*	*memory*	*modules.)*	
j=5	i=0,cc=0,s=0	i=5,cc=1,s=0	i=10,cc=2.s=0	i=15,cc=3,s=0	
	i=20,cc=4,s=0	i=25,cc=5,s=0	i=3-,cc=6,s=0	i=35,cc=7,s=0	**0**
	… *(Uses all*	*four of the*	*memory*	*modules, again.)*	
j=6	analogous to j=2	…	…	…	**2**
j=7	analogous to j=1	…	…	…	**0**
j=8	analogous to j=4	…	…	…	**3**

The above table shows a clear pattern where all odd j values access all 4 memory modules and have no stall cycles. The even values not divisible by 4 access 2 modules with 2 stall cycles accruing on every access to Module 0 after the first access. For j values divisible by 4, only Module 0 is used resulting in 3 stall cycles for every access after the first. The overall formulas are:
(1) When j is odd (congruent to either 1 or 3 modulo 4) then Stall cycles = 0,
(2) when j is congruent to 2 modulo 4 then Stall cycles = 2 * (floor(1024/j) -1),
(3) and when j is congruent to 0 modulo 4 then Stall cycles = 3 * (floor(1024/j) -1).

Clearly, there is a huge execution time advantage to be had by writing code that spreads its accesses as widely as possible over all available memory modules.

5. **[3 pts.; 1 point for each answer]** The CPU time equation is as follows, CPI means Clock cycles per instruction: CPU Time = (Instructions/Program) * (CPI)*(Seconds/Clock cycle). For each of the three factors in the CPU Time equation, answer the following questions: (1) Can loop unrolling ALONE improve this factor, worsen this factor, or cannot affect this

factor? (2) If loop unrolling either improves or worsens the factor, how does this occur?
Answer:
Instructions/Program: IMPROVED.
Explanation: Loop unrolling will reduce Instructions/Program (a dynamically counted amount because it is part of an equation of time) by reducing the number of iterations and thus the number of iteration- management instructions. Iteration management instructions are those that update loop counters and test the loop count or condition to determine whether another iteration is needed.
CPI: IMPROVED. Explanation: Loop unrolling reduces the number of dynamically executed branch instructions and, thus, reducing the number of taken-branch stalls.
Seconds/Clock cycle: CANNOT AFFECT. Explanation: Loop unrolling cannot affect the seconds/clock cycle factor because this factor is a function of the hardware only. It is not affected by software. Deeper consideration leads to the question, "Is the CPI just for taken branch instructions less than the average CPI for the other instructions in the loop?" Assume that the instruction mix in the loop is heavy with several-cycles to execute FP instructions. Then removing a relatively faster taken branch would actually increase (worsen) the resulting CPI. This is also a nice time to also note that benchmarks that report their results in terms of CPI or its reciprocal, instructions per second, can be compromised if you allow the person generating the results to pad the benchmark with low-CPI instructions, such as NOPs. Add enough NOPs to any program and you can, in the limit as number of executed instructions approaches infinity, show any computer running any benchmark at 1 instruction per clock cycle. Don't think that this has never been done in the computing business, before we got wise.

6. **[3 pts.; 1 point for each answer]** Once a loop has been unrolled, which factor(s) of the CPU Time equation can be improved, worsened, or cannot be affected through the application of instruction scheduling? Explain.
Answer:
Instructions/Program: NOT AFFECTED. Explanation: The larger basic block created by unrolling should not support further reduction in instructions per program unless there was an inefficiency in the coding before unrolling as well. If such inefficiency exists, then it could be improved before any loop unrolling.
CPI: IMPROVED. Explanation: CPI can be improved, perhaps significantly, because unrolling creates a larger basic block that can enable better scheduling of instructions to eliminate stall cycles. Stall cycles increase CPI.
Seconds/Clock cycle: CANNOT AFFECT. Explanation: Again, this factor is determined only by hardware technology and the arrangement of that hardware into the processor circuit. Clock speed is what it is, regardless of software.