CS 231 – Review Questions for Final Exam

Some of the questions have appeared on exams in the past. To study for the final, please look at the first two review documents as well. Besides the review questions, look over your old tests & quizzes, labs and notes in the course. Good luck!

1. List some similarities and differences between a high-level language and assembly language programming. Name 2 things that are easier to do in HLL than in assembly. Name 2 things that are easier to do in assembly than in HLL.

2. Write MIPS instructions that cause the machine to beep. The ASCII code for the beep is 7.

3. Show how we can save the registers $s0 and $s1 onto the system stack using just 3 instructions.

4. The 6-bit one's complement representation of –12 is the same as the 6-bit unsigned representation of x. The 6-bit sign-magnitude representation of –12 is the same as the 6-bit two's complement representation of y. What is the 8-bit biased-127 representation of x+y?

5. How does sign-magnitude sign extension differ from 2's complement sign extension?

6. Explain why the MIPS instruction set has two shift right instructions yet only one shift left instruction.

7. Write a MIPS procedure octal_input that reads a positive octal integer from standard input. You may assume that a newline character immediately follows the integer. The value of this integer should be stored into register $v0. Be sure to make your procedure computationally efficient.

8. Suppose the branch pseudo-instruction

   ```
   bgt $12, 100, loop
   ```

   is synthesized into true assembly as

   ```
   addi $1, $12, –100
   bgtz $1, loop
   ```

   a. Explain how the branch instruction bgtz can make an incorrect branch decision when the addi instruction overflows.

   b. Translate the above branch pseudo-instruction into correct true assembly language code.

9. Use Booth's algorithm to write efficient assembly code that implements the assignment statement: x = y * 100;

10. Convert the decimal value –27.1 into single-precision IEEE FPS notation. Express your answer in hexadecimal.

11. Calculate the largest finite number that can be represented in single-precision IEEE FPS. How much larger is it than the 2nd largest finite number?

12. Compile the statement total += 0.05; You may assume that total is a double-precision number. Be sure to show how the value 0.05 is loaded into a register.

13. Let $f12 be a single-precision floating-point number represented in IEEE FPS notation. Write a pseudocode algorithm that describes how the bits of $f12's representation are modified when we multiply its value by 2.

14. Examine figure 4.17 in the textbook. Distinguish the 2 inputs to the multiplexor located in the upper right portion of the figure. In other words, explain when each input would be appropriate.

15. The execution of the typical add instruction add A, B, C involves several steps taken by the CPU and memory. These steps are listed below, but they are out of order. Write the numbers 1-6 in the blanks indicating the proper order of these steps.

    _____ The values of B and C are obtained from memory.

    _____ The opcode is decoded.

    _____ The program counter is incremented.

    _____ The value of A is recorded in memory.

    _____ The address in PC is sent to memory and the instruction is fetched.

    _____ The addition operation is performed.

16. Examine figure 4.60 in the textbook. According to this figure, what information is contained in the EX/MEM pipeline register?

17. What is the CPI for a machine that has a MIPS rating of 200 and a 500 MHz clock rate?

18. Machine A can run program X in 6 seconds and program Y in 7 seconds. Machine B can run program X in 3 seconds and program Y in 4 seconds. Suppose that program X performs some essential operation that must run twice each minute, so our throughput is measured by how many times we can run Y. If machine A costs $5000, how much should we be willing to pay for machine B?

19. Suppose a machine's instruction set consists of 4 classes of instructions

Class	CPI	% of all instructions executed
A	1	60
B	2	20
C	3	10
D	4	10

    a. What is the average CPI for this machine?

b. If a program P executes 10 million instructions and the clock rate is 400 MHz, what is the execution time of P?
c. Suppose the machine is improved in such a way that each class C instruction can be replaced by two class A instructions. Re-calculate the average CPI and the execution time for program P.

20. How do control (branch) hazards come about?

21. In the pipelined datapath, in which we added a forwarding unit to the EX section of the CPU, what inputs are necessary for the forwarding unit? In other words, what does the forwarding unit need to know in order to determine whether and what to forward?

22. What inputs are necessary for the hazard detection unit?

23. Why does a 2-bit branch predictor perform better than a 1-bit predictor?

24. Write a code fragment in which a cache miss may overlap with a stall. Show the overlap with a pipeline diagram.

25. When designing a cache, what does spatial locality have to do with the cache line size?

26. Consider the following fragment of assembly code:

```
add $4, $5, $6
lw $7, 0($8)
sub $5, $6, $7
```

Assume that each instruction is required to spend at least one cycle in each of the 5 pipeline stages, and that the pipeline is initially empty. How long will it take to completely execute the above instructions, given that the first 2 hit in cache and the 3rd one misses, and that the miss penalty is 9 cycles?

27. Suppose an instruction cache is 2-way set-associative, has 2 sets, and each cache line has 2 words. Classify each of the following instruction address fetches as hit or miss, assuming the cache is initially empty:

0, 4, 8, 12, 36, 40, 44, 16, 20, 4, 8, 28, 32

Also give the final contents of the cache.

28. Explain how an instruction can behave as a "first hit" in the instruction cache. In other words, it's a hit the first time it is referenced, but thereafter it is a miss in cache. Be specific. Give an example cache configuration and outline the code sequence that would run.

29. Suppose that a direct mapped cache has a miss rate of 8%, and the miss penalty is 9 cycles. If a set-associative design would reduce the miss rate to 6% but increase the hit time by 10%, which cache design would have a faster average access time?

30. Give an example showing how a memory reference can incur a TLB miss, but not a page fault.

31. How could it be possible for a program to remove itself? For example, a program could open itself for writing. When we do so, its original contents are lost. Yet if the program is gone, how can it be running?

Brief Answers to Review Questions

1. You could choose any high-level language, but for fun I'll choose C++. Let's compare C++ and MIPS assembly. Some similarities include: they both support control structures (straight line, making choices, looping) and the ability to structure a program into procedures or functions. Both languages support fundamental arithmetical operations. The major difference is that C++ is machine-independent and can be used on any machine that has a C++ compiler. An Assembly language is specific to the machine it was designed for. In assembly, you are restricted to having only 1 instruction per line, and it can't have more than 3 operands. C++ gives you more freedom in this respect.

   C++ makes it easier (than assembly) to do almost anything, such as writing complex if-expressions like this: if (a > 1 and b <= 6) which would be more tedious in assembly. Another example would be the way we call a function with a lot of arguments. In C++, it doesn't matter how many parameters you put in parentheses, but in assembly, you have to carefully manage the system stack.

   In assembly it's easier to perform a bit rotate because there are specific rol and ror instructions. Also, it is rather straightforward to obtain the upper 32 bits of an integer multiply (using mfhi), something which would be more painful in C++.

2. la $a0, string
   li $t0, 7
   sb $t0, 0($a0)
   sb $zero, 1($a0)
   li $v0, 4
   syscall

3. Here is one way:
   ```
   addi $sp, $sp, –8
   sw $s0, 8($sp)
   sw $s1, 4($sp)
   ```

4. x = 51 and y = –20, so x+y in biased-127 looks like 158 in unsigned: 10011110

5. In sign-magnitude extension we move the sign bit to the new left end of the number and fill the bits in between with 0: note that where the sign bit used to be we will always have 0. For 2's complement, we pad the number with sign bits (all being either 0 or 1).

6. There is no "shift left arithmetic" because sll already correctly performs multiplication by powers of 2.

7. I wish we had the ability to read individual characters in assembly. Well, let's say we did. Let's assume I/O routine number 12 reads a single character, and stores that character's ASCII value into $v0. If this were an exam question, I would say that you could make this assumption.

```
    li $s1, 0                   # set value of number initially to 0
read_loop:
    li $v0, 12
    syscall
    move $s0, $v0               # $s0 = new character from input

    beq $v0, 10, exit_loop      # when newline, quit

    addi $v0, $v0, –48          # subtract '0' to get digit value
    sll $s1, $s1, 3             # multiply existing number by 8
    add $s1, $s1, $v0           # add new digit
    j read_loop

    move $v0, $s1               # we have to store result in $v0
```

8. Suppose $12 is a very large negative number, so large that subtracting 100 from it would produce a positive number, yielding overflow. The original branch condition bgt is false since any negative number is less than 100, so we should not branch. But in the synthesized code, the value of $1 is positive, which is greater than 0, so the bgtz performs the branch when it should not.

   A better way:

```
    li $13, 100
    slt $1, $13, $12
    bnez $t1, loop
```

9. Assume $s1 = x and $s2 = y.

```
    sll $t0, $s2, 7
    sll $t1, $s2, 5
    sub $t0, $t0, $t1
    sll $t1, $s2, 2
    add $s1, $t0, $t1
```

10. 0xc1d8cccd

11. The largest number is $2^{128} - 2^{104}$. To create the 2nd largest number, we need to clear the rightmost bit in the mantissa, which means we are subtracting 2^{104}.

12. We need to add a line to the data segment first:

```
    .data
    …
nickel:     .double 0.05
```

    and the code goes in the text segment. Assume $f12 = total.

```
.text
…
la $t0, nickel
l.d $f0, 0($t0)
add.d $f12, $f12, $f0
```

13. In most cases, all we need to do is add 1 to the exponent part. But there are some exceptions from special cases. If adding 1 to the exponent has just caused all the exponent bits to become 1, then we need to clear all the mantissa bits to signify infinity. If the number we started with is 0, then we should do nothing since the answer is also 0. If the original number is denormalized, we should instead shift the mantissa left by 1. And if doing so causes the leftmost bit of the mantissa to disappear into the hidden bit, we need to set the exponent to 00000001. If the original number is infinity or not a number, then we should do nothing to the representation.

14. Input 0 is for the usual incrementing of the PC by 4.
    Input 1 is for setting the PC based on the branch target address.

15. The numbers are 4, 3, 2, 6, 1 and 5.

16. We need to store the values in the control lines used in the MEM and WB stages, the result of the ALU operation, the number of the destination register, and the number of the register that is used as the operand to a load or store instruction, if applicable (i.e. load into which register / store from which register).

17. 2.5 cycles per instruction

18. $ 9843.75

19. a. 1.7 cycles per instruction
    b. 0.0425 second
    c. about 1.45 cycles per instruction, and 0.04 second

20. When we have a conditional branch instruction, we don't know for sure which instruction to fetch next until we test the 2 register operations for ==. When the branch enters the ID stage we don't yet even know this is a branch instruction, so we just fetch its successor instruction anyway. If the branch winds up being taken, the successor instruction would need to be flushed from the pipeline. (We're assuming here that the machine does not have delayed branches and there is no special hardware that attempts to predict the behavior of a branch.)

21. As soon as we begin the EX stage, we need to know the source operands of this instruction, and the destination operands of the instructions in the MEM and WB stages.

22. The hazard detection unit needs to know the source operands of the instruction in the ID stage, the destination register of the instruction in the EX stage, and whether the EX instruction is a load instruction.

23. The one-bit predictor will be wrong twice in a row when there is a change in the branch behavior.

reality: … t t t n t t t…
1-bit: … t t t t n t t…
2-bit: … t t t t t t t …

It seems like the 1-bit predictor is in a struggle to catch up with what the branch is doing. The 2-bit predictor shows more faith that the branch will continue to be taken.

24. Let's say we have a code fragment like this:

```
lw $t1, 0($t2)
add $t3, $t2, $t1
sub $t5, $t6, $t7
```

cycle	IF	ID	EX	MEM	WB
1	lw				
2	add	lw			
3	sub	add	lw		
4	sub	add		lw	
5	sub		add		lw
6	sub				
7	sub				
8	sub				
9	sub				
10	sub				
11	sub				
12	sub				
13		sub			
14			sub		
15				sub	
16					sub

25. Spatial locality tells us that instruction I will likely be followed by I+1 and I+2. If instruction I misses in cache and we have to load it from main memory, it would be worthwhile to bring in I+1 and I+2 as well because they would have to be loaded eventually so do it now. Then I+1 and I+2 can count as hits, and we won't have to go to main memory as often. Having a wide (4 or 8 word) cache line allows us to bring in several consecutive instructions in just this manner.

26.

cycle	IF	ID	EX	MEM	WB
1	add				
2	lw	add			
3	sub	lw	add		
4	sub		lw	add	
5	sub			lw	add
6	sub				lw
7	sub				
8	sub				
9	sub				
10	sub				
11	sub				

```
12          sub
13                  sub
14                          sub
15                                  sub
16                                          sub
```

27. The hits are 4, 12, 44, 20 and second fetch of 8. All others are misses. The final contents of the cache:

	First cache line		Second cache line	
set 0	0	4	32	36
set 1	8	12	24	28

28. Let's say we have a direct-mapped cache big enough for only 8 words. It has 2 cache lines of 4 words each. Now let's say the program looks like this:

```
    inst 0  miss
    inst 1  hit
loop:
    inst 2  hit, miss, miss, …
    inst 3  hit, hit, hit, …
    inst 4  miss, miss, miss, …
    inst 5  hit, hit, hit, …
    inst 6  hit, hit, hit, …
    inst 7  hit, hit, hit, …
    inst 8  miss, miss, miss, …
    inst 9  hit, hit, hit, …
    inst 10  hit, hit, hit, …
    inst 11  hit, hit, hit, …
```

    where instruction 11 loops back to instruction 2. Note that fetching instruction 8 causes the first program line to be kicked out of the cache. So instruction 2 is our "first hit" instruction.

29. For the direct-mapped cache, the average access time = 1.72 cycles. For the set-associative cache, the average access time = 1.634 cycles. So in this case, we do better with the set-associative cache.

30. All we need to do is to fetch two instructions (or load two data values) that map to the same line in the TLB. Make sure that both instructions are already in RAM. If we follow the example in class: Let's suppose the TLB has 256 entries, the page size is 8 KB, the virtual memory size is 4 GB and the physical memory size is 256 MB. Therefore, every virtual address is expressed by an 8-digit hexadecimal number. The first 5 digits comprise the virtual page number, and the last 3 digits comprise the page offset. Remember that the virtual page number is itself partitioned into a TLB tag and TLB line number (analogous to tag and line numbers in a cache). Since the TLB has 256 entries, the TLB line number is 8 bits (2 hex digits) in length. Therefore, the 5 hex-digit virtual page number comprises a 3-digit TLB tag and 2-digit TLB line number.

    Let x have the virtual address 0x40000000 and y have the virtual address 0x50000000. They lie in virtual pages 40000 and 50000, respectively. Suppose we fetch x, y, x, y, and consider the last fetch of y. At that point, both virtual pages 40000 and 50000 are in physical memory, so it cannot be a page fault.

But these two virtual page numbers both map to TLB line number 00, so fetching y will incur a TLB miss, even though both x and y are in RAM.

31. The program that is running resides in main memory (though not for long!), while the program we are deleting is the copy on disk.