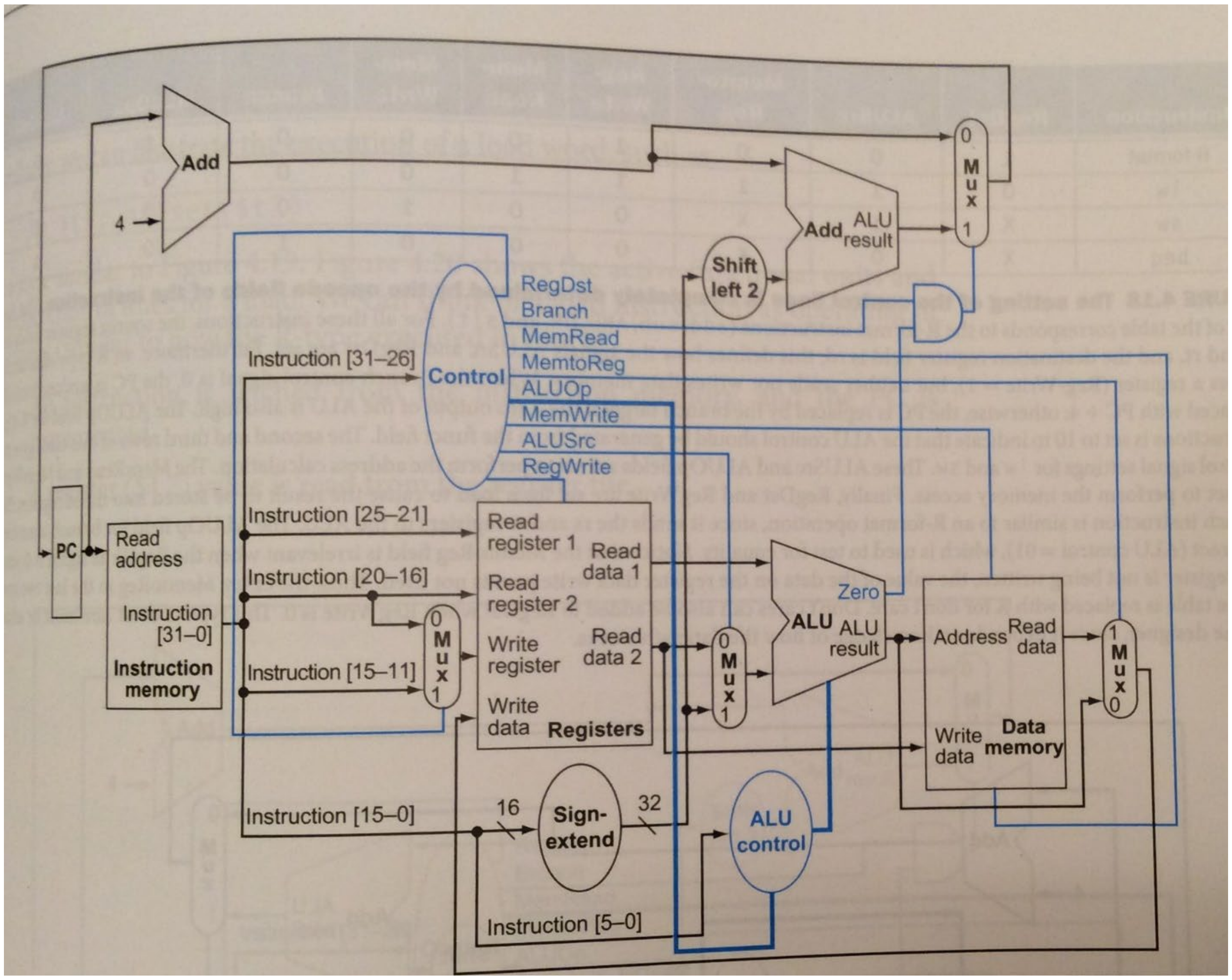
CS 231 – Review for test #2

Here are some questions that appeared on past exams. Some questions will refer to this diagram:



1. What is the difference between overflow and underflow?

2. A typical integer instruction takes 5 pipeline stages, but beq, sw, and jump instructions can be accomplished in fewer stages. For each of these 3 types of instructions, identify the number of clock cycles needed and what operations are performed during each stage.

3. Assume that x is a 32-bit 2's complement integer, and y is an integer between 1 and 31 inclusive. Assume that x = 0x1e4, and z is assigned the value (x << y) >> y. In other words, starting with the number x, we perform a left shift of y bits, and then an arithmetic right shift of y bits, in order to obtain z. For which values of y will z be:
   a. The same as x?

   b. Negative?

4. Suppose that the sra instruction were not available. The same functionality can be accomplished by using ror and Boolean operations. For instance, it is possible to replace the instruction sra $s0, $s0, 6 by the following code:

```
        bltz $s0, negative
        ____ $s0, $s0, 0x______________
        j after
negative:
        ____ $s0, $s0, 0x______________
after:
        ror  $s0, $s0, 6
```

   Two of the above instructions are incomplete. Fill in the blanks to complete these instructions by specifying the appropriate Boolean operations and hexadecimal constant operands so that the overall effect of the code is the same as the instruction sra $s0, $s0, 6.

5. Suppose x and y are integer variables on a machine that only supports 8-bit 2's complement representation and arithmetic. If x = 0x5a, then what values of y will cause overflow when performing the operation x + y?

6. Suppose that when adding two single-precision real numbers x and y, it turns out that before the addition takes place, the exponents must be normalized, and after the addition, the exponent of the result must also be normalized. Illustrate how this can happen with your choice of values for x and y, and show the steps of the floating-point addition process.

7. Estimate the percentage of single-precision IEEE FPS representations that are devoted to floating-point numbers between 1 and 1 million. Justify your answer.

8. Refer to the attached processor diagram. For each multiplexor, discuss when each input is appropriate.

9. Consider the expression ~n that sometimes appears in high-level language programs. Re-write this expression using only arithmetical (not bitwise) operations.

10. What is the effect of this assignment statement (that could occur in languages such as C, C++, Python or Java) on the bits of x? Assume that ~ has higher precedence than <<.

```
x ^= ~(~0 << 9) << 5
```

11. 0x80001200 is the single-precision floating-point representation of what number?

12. Give 8-bit representations for the number – 46 in the following representation schemes:
    a. Sign magnitude

    b. 2's complement

    c. 1's complement

    d. Biased 127

13. Booth's algorithm is used to make multiplication by a constant more efficient.
    a. Show how the instruction mul $s1, $s1, 0x237 can be replaced by a more efficient sequence of instructions, using only shift, add and/or subtract instructions. You may use additional registers to hold temporary values.

    b. Why would Booth's algorithm likely not help with simplifying an assignment statement such as z = x * y?

14. By default, the OS functions for printing floating-point numbers give too much precision. Explain in pseudocode how we can print a floating-point value to exactly 2 decimal places, using the functionality for printing integers and strings.

15. Refer to the processor diagram. Discuss how the following operation is performed: sw $15, 16($17). Be sure to describe the role of each functional unit that is used to execute this instruction.

16. Consider this code fragment.

```
and $1, $2, $3
or $4, $5, $1
xor $7, $1, $4
```

    Identify the instances of forwarding that are needed to avoid stalls. Write a pipeline diagram that shows how these 3 instructions execute.

17. Consider the following sequence of three instructions. Draw a pipeline diagram that shows how these instructions proceed through the CPU. Assume that the pipeline is initially empty, and each instruction requires 1 cycle in each stage, except that mul requires 7 cycles in the EX stage.

```
mul $s1, $s2, $s3
lw $s4, 0($s1)
add $s5, $s5, $s4
```

Answers to review questions.

1. Overflow occurs when the result of an operation cannot be represented because it lies beyond the range of possible values.
   Underflow occurs when the result of an operation is so small in magnitude that it cannot be distinguished from zero.

2. The minimal stages needed for beq, sw, and j:

	IF	ID	EX	MEM	Total cycles
beq	Fetch	Decode; compute target address	Compare registers; set PC if condition true		3
sw	Fetch	Decode	Perform address calculation, based on register value, not just instruction encoding	Store value into memory	4
j	Fetch	Decode; compute target address for new PC value			2

3. x = 0000 0000 0000 0000 0000 0001 1110 0100

   And note that the bits that are set are 8, 7, 6, 5 and 2.

   a. We can shift left as long as we still have at least one leading zero that started at position 9. Since 31 – 9 = 22, we conclude that 1 <= y <= 22.

   b. We want to shift left to the point where a 1 becomes the sign bit at position 31. Observe that:
   31 – 8 = 23
   31 – 7 = 24
   31 – 6 = 25
   31 – 5 = 26
   31 – 2 = 29.
   These are the solutions for y: 23, 24, 25, 26, 29.

4. Here is the completed code. Note that we have to set the mask values before we rotate. In one case, we want to “or” with 6 consecutive 1’s, and in the other case we want to “and” with 6 consecutive 0’s.

```
        bltz $s0, negative
        and  $s0, $s0, 0xffffffc0
        j after
negative:
        or   $s0, $s0, 0x0000003f
after:
        ror  $s0, $s0, 6
```

5. Since x is positive, y must also be positive and x+y must be negative in order for us to experience overflow.
   "y is positive" means that 0 <= y <= 0x7f.
   "x+y is negative" means that 0x80 <= x+y <= 0xff. Subtracting x = 0x5a, we obtain:
   0x26 <= y <= 0xa5.
   If you combine these conditions, the resulting condition on y is 0x26 <= y <= 7f.

6. We can choose x and y with different exponents so that the exponent for x+y is larger than the exponent of either number. For example, we can let
   x = 7 (exponent 2)
   y = 2 (exponent 1)
   x+y = 9 (exponent 3)

   $7 = 1.11_2 * 2^2$
   $2 = 1.0_2 * 2^1$

   We add:
   0 10000001 (1)11
   0 10000000 (1)

   First, we match the smaller exponent to the larger:
   0 10000001 (1)11
   0 10000001 (0)1

   Second, we add mantissas. Conversion between sign-magnitude and 2's complement is trivial because they are positive numbers.

```
    0(1)11
+   0(0)1
--------------
    0(10)01        Note the two hidden bits.
```

   Third, we form our final answer.
   0 10000001 (10)01 which needs to be normalized as

   0 10000010 (1)001 and we see this equals 9.

7. 1 is 2^0 and 1 million is approximately 2^{20}.

   For the sign bit we have 1 choice: it must be 0.
   For the exponent, we have 20 choices for the values 0-19.
   For the mantissa, we have 2^{23} choices since there are 23 bits.
   And we know that the total number of representations is 2^{32}.

   So, the desired proportion is 1 * 20 * 2^{23} divided by 2^{32}, which is 20 / 2^9 or about 4%.

8. Multiplexor at top: 0 input means we default to fetching the next instruction. 1 input means we fetch the instruction that is the target of a conditional branch.

   Multiplexor before bank of registers: 0 input means the destination register is given by bits 20-16 in the instruction, meaning that we have the lower 16 bits allocated to an immediate value, so this is a load instruction. 1 input means the destination register is in bits 15-11 for ordinary register-type instructions that specify 3 registers rather than an immediate value.

   Multiplexor before ALU: 0 input means we want the 2nd operand of the ALU to be the contents of a register, e.g. for an add instruction. 1 input means we want instead the sign extended constant value stored in the instruction, e.g. for addi.

   Multiplexor after data memory: 0 input means the value to store in a register is coming straight out of the ALU, e.g. for an add instruction. 1 input means the value to put in a register is coming out of memory, e.g. for a load instruction.

9. 1's complement is one less than 2's complement. For example, ~0 is –1, so we conclude that
   ~n = – n – 1.

10. ~0 << 9 is $1^{23}0^9$. And, putting a ~ in front of this number gives us:
    ~(~0 << 9), which becomes $0^{23}1^9$.

    Then, the << 5 tells us to shift the bits left by 5, and now we have $0^{18}1^90^5$. Notice where the 1's are in this number.

    The statement will invert the 9 bits of x, starting from the sixth bit from the right.

11. First, let's determine the sign bit. Because the first hexadecimal digit is 8, and we know that 8 = 1000 in binary, we see that the sign bit is 1, meaning a negative number.

    In single precision, we have 1 sign bit, 8 bits for the exponent, and 23 for the fractional part of the mantissa. Writing the bits, we have:

    1000 0000 0000 0000 0001 0010 0000 0000

    And rearrange into groups of 1, 8 and 23:

    1 00000000 (0) 000 0000 0001 0010 0000 0000

    Because all the exponent bits are 0, we know we are dealing with a denomalized number. Therefore, the exponent is the lowest possible, which is 1 – 127 = –126. This means that the hidden bit stands for the binary place for 2^{-126}. Each mantissa bit to the right is one power of 2 lower.

    Our number only has 2 mantissa bits set, at the positions –137 and –140. Therefore, our answer is $-(2^{-137} + 2^{-140})$.

12. Representations
    a. Sign magnitude: $-46 = -(2^5 + 2^3 + 2^2 + 2^1)$ = 10101110
    b. 2's complement: +46 would have been 00101110, and we invert until the rightmost 1. So, we obtain 11010010.
    c. 1's complement: 1 less than 2's complement, which is 11010001.
    d. Biased 127: The unsigned representation of 127 – 46 = 81 is $2^6 + 2^4 + 2^0$ = 01010001.

13. Booth's algorithm…
    a. 0x237 = 0010 0011 0111 = $2^9 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0$. We can combine consecutive powers of two as follows: $2^5 + 2^4 = 2^6 - 2^4$, and $2^2 + 2^1 + 2^0 = 2^3 - 2^0$. This means we can write: $2^9 + 2^6 - 2^4 + 2^3 - 2^0$. But once again we see we have two consecutive powers of two. The third and fourth terms can be combined into a single term, and our expression can just take four terms: $2^9 + 2^6 - 2^3 - 2^0$.

```
sll $t0, $s1, 9
sll $t1, $s1, 6
add $t0, $t0, $t1
sll $t2, $s1, 3
sub $t0, $t0, $t2
sub $s1, $t0, $s1
```

b. The compiler does not know the value of x or y, so it cannot break the number up into powers of 2.

14. We want to print the value of x.
Add 0.005 to x.
Multiply x by 100.0
Convert: truncate x to an integer.
Divide the integer by 100: keep track of its quotient and remainder.
Print the integer quotient.
Print the "."
Print the remainder. Remember to print a leading 0 if the remainder is less than 10.

15. We ultimately want to store the value of $15 into memory at address $17 + 16. Here are the steps.

- The address of the instruction is in PC. This value is sent to instruction memory, so it can fetch the instruction for us. (The instruction itself will then reside in the instruction register).
- We decode the instruction. This means we split up the bits to figure out its parts. From the opcode, we recognize it to be a store instruction. This opcode will then determine the values of all of the control lines coming out of the control unit that will, in turn, influence how the multiplexors and later functional units behave. The numbers 15 and 17 are sent to the register file to get the values in those registers. The immediate value 16 is sent into the sign extender. Meanwhile, at the top of the diagram, we add 4 to the PC so that we know where the next instruction is.
- We execute the instruction. The ALU takes two inputs: the first input is the value in register 17. The second input, coming through a multiplexor, is the sign extended 16. The ALU adds these numbers to determine the address to send to data memory.
- We access data memory. The value from register 15 is stored in memory at the address that has just been computed by the ALU.

16. Although there are data dependencies, there should be no stalls thanks to forwarding. Let the and, or and xor instructions be numbered 1-3. The pipeline diagram looks like this:

cycle	IF	ID	EX	MEM	WB	Instruction being fetched:
1	1					`and $1, $2, $3`
2	2	1				`or $4, $5, $1`
3	3	2	1			`xor $7, $1, $4`
4		3	2	1		
5			3	2	1	
6				3	2	
7					3	

The destination of instruction 1, $1, can be forwarded to instruction 2 and instruction 3, because these two instructions both rely on $1 as a source register.

The destination of instruction 2, $4, can be forwarded to instruction 3.

17. Pipeline diagram

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