University of Maryland Baltimore County
Department of Computer Science and Electrical Engineering

CMSC 411, Computer Architecture

Assignment #2 **Solutions** Due: Monday 10/5/17 in class

Question 1: *(24 Points)*

Given the bit pattern: 0010 0100 1001 0010 0100 1001 0010 0100
What does it represent, assuming that it is:

A) a two's complement integer? **6 points**

Left most = 1, thus a positive integer

Since the number is positive, no need to flip bits and plus 1 like negative number. So you can just convert the binary numbers.

$2^{29} + 2^{26} + 2^{23} + 2^{20} + 2^{17} + 2^{14} + 2^{11} + 2^{8} + 2^{5} + 2^{2} = + 613566756$

B) an unsigned integer? **6 points**

It is the same number as question 1.A
$2^{29} + 2^{26} + 2^{23} + 2^{20} + 2^{17} + 2^{14} + 2^{11} + 2^{8} + 2^{5} + 2^{2} = 613566756$

C) a single precision floating point number? **6 points**

0 01001001 00100100100100100100100

Single precision floating point number: $(-1)^{\text{sign bit}} * (1 + \text{fraction}) * 2^{(\text{exponent} - 127)}$

Sign bit (left most bit): 0
Exponent (next 8 bits): 01001001 = 73
Fraction: 00100100100100100100100 = $2^{-3} + 2^{-6} + 2^{-9} + 2^{-12} + 2^{-15} + 2^{-18} + 2^{-21}$ = (1/8 + 1/32 + 1/512 + 1/4096 + 1/32768 + 1/262144 + 1/2097152 = 0.158482074737548828125

The answer is : $(-1)^0 * (1 + 0.158482074737548828125) * 2^{(73-127)}$
$= 1.158482074737548828125) * 2^{-54}$

D) a MIPS instruction? **6 points**

The first 6 bits (left most 6 bits) represent the opcode: 0010 01 = 8 + 1= 9. This indicates **addiu** (add immidiate unsigned) instruction. Note that the immediate value can be 0 - 2^{16} which is 65535 (zero-extended).

The instruction: addiu $t, $s, i
The operation: $t = $s + SE(i)
Instruction Encodings: oooooss sssttttt iiiiiiii iiiiiiii

In details, 001001 00100 10010 0100100100100100

opcode - 6 bits	rs - 5 bits - dest base	rt - 5 bits - source	Immidiate value - 16 bits
0010 01 = 9	00 100 = 4	1 0010 = 18	0100100100100100 = 18724

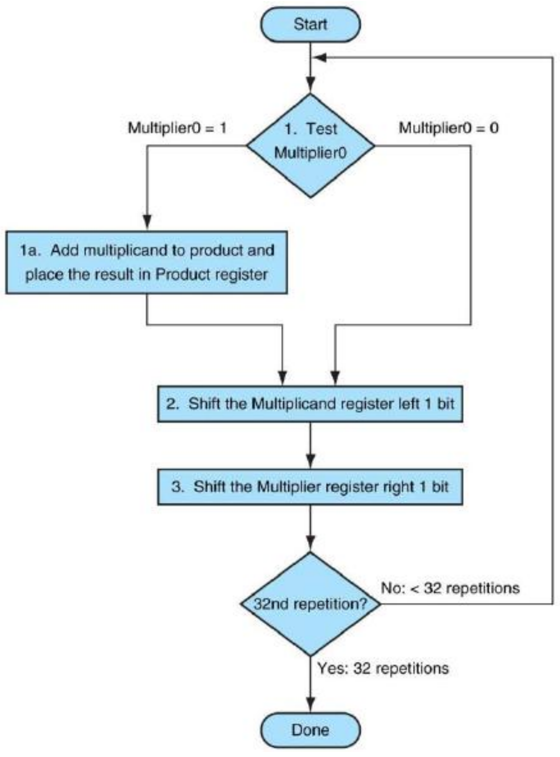
The instruction is:
addiu $18, $4, 18724 # add the immediate value of 18724 with the value of register $4 and put the result in the register $18

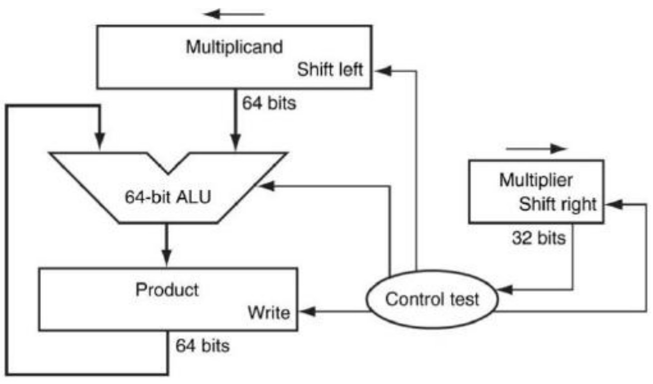
Note that: rs and rt are the registers (two of 32) as source and destination respectively, not the values of 4 and 18. For those who refer to a value (no symbol $), you are wrong.

Question 2: *(36 Points)*

In this question we would like to compare the speed of two multipliers.

A) Calculate the time necessary to perform a multiply using the approach below (and also discussed in class). Assume that it takes *T* time units to perform a step in the multiplication process (block in the flowchart). Assume that in step "la" an addition is always performed either the multiplicand will be added, or a 0 will be. Also assume that the registers have already been initialized (you are just counting how long it takes to do the multiplication loop itself). If this is being done in hardware, the shifts of the multiplicand and multiplier can be done simultaneously. If this is being done in software, they will have to be done one after the other. Solve for each case. **12 points for each case, total 24 points**


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done



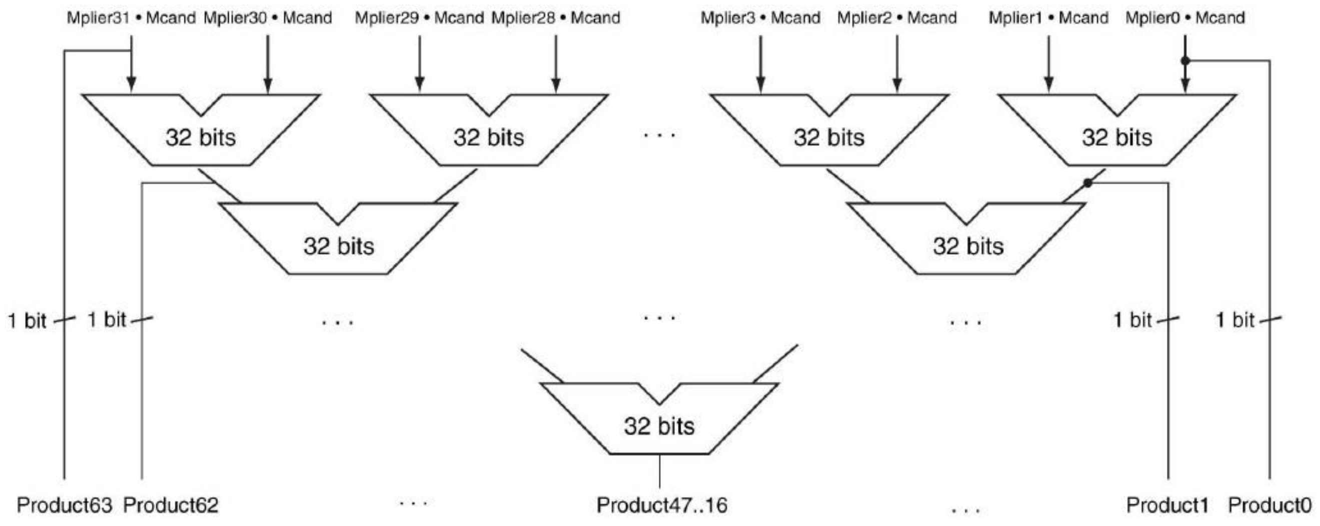
For hardware case, it takes 1 cycle to do the add (step 1a), 1 cycle to do the shift (step 2, 3), 1 cycle to decide if we are done and if not we are doing 1 cycle to decide at step 1 again. So the loop takes (4 x *T*) cycles. There are 32 repetitions.

So the the time necessary to perform a multiply for hardware case is 4*T* x 32 = **128*T***

For software case, it takes 1 cycle to do the add (step 1a), 1 cycle to do each shift (step 2, 3 = 2 cycles) , 1 cycle to decide if we are done and if not we are doing 1 cycle to decide at step 1 again. So the loop takes (5 x *T*) cycles. There are 32 repetitions.

So the the time necessary to perform a multiply for software case is 5*T* x 32 = **160*T***

B) Calculate the time necessary to perform a multiply using the configuration below, if an adder takes *T* time units. **12 points**



From the diagram, the adders are arranged in a tree structure (leaves at the top level). Also from diagram, it is 32-bit adder, thus to reach the root, it will require $\log_2 32 = 5$ levels

For 32 bits word, it requires 31 adders (16 + 8 + 4 + 2 + 1) in 5 levels = 5 x *T* = **5*T***

Question 3: *(40 Points)*

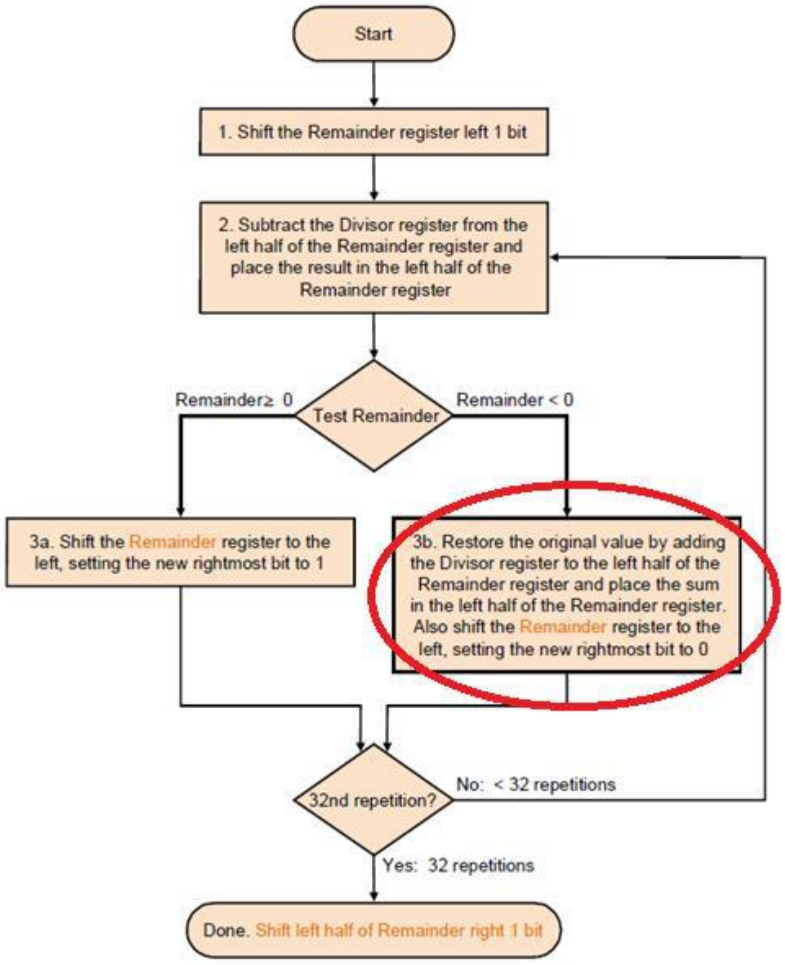
The division algorithm discussed in class is called *restoring division*, since each time the result of subtracting the divisor from the dividend is negative you must add the divisor back into the dividend to restore the original value. Recall that shifting left is the same as multiplying by two. Let's look at the value of the left half of the Remainder again, starting with step 3b of the divide algorithm and then going to step 2:

$$(\text{Remainder} + \text{Divisor}) \times 2 - \text{Divisor}$$

The value is created from restoring the Remainder by adding the Divisor, shifting the sum left, and then subtracting the Divisor. Simplifying the result we get

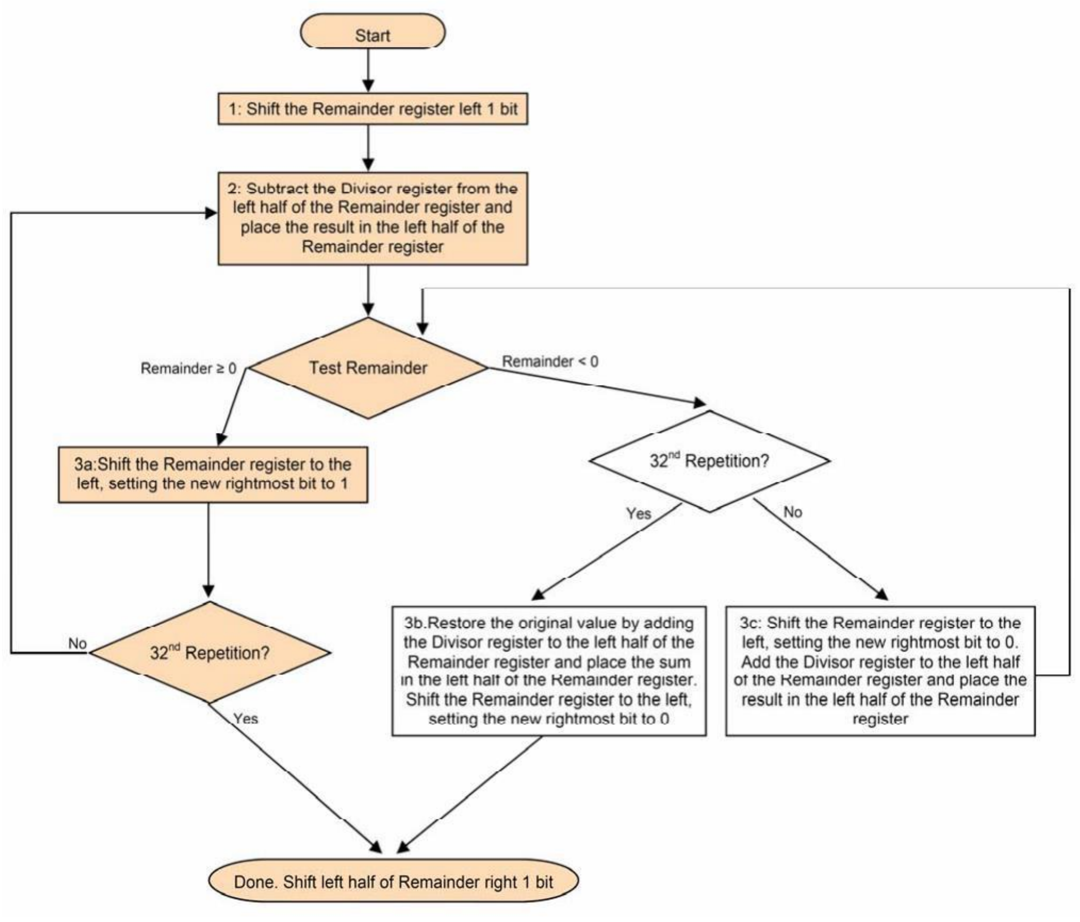
$$\text{Remainder} \times 2 + \text{Divisor} \times 2 - \text{Divisor} = \text{Remainder} \times 2 + \text{Divisor}$$

Based on this observation, write a *non-restoring division* algorithm. Show that your algorithm works by dividing (0000 0111) by (0010) **Modification 20 points, show algorithm works 20 points**


Start
1. Shift the Remainder register left 1 bit
2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register
Test Remainder
Remainder≥ 0
Remainder < 0
3a. Shift the Remainder register to the left, setting the new rightmost bit to 1
3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done. Shift left half of Remainder right 1 bit

From Lecture 8 Slide 11 under "Divide Algorithm Version 3", we will modify the oval part of the flowchart. Our *non-restoring division* algorithm will match the restoring division algorithm in all respects, except for the case of the remainder being negative after step 2. We replace this logic first with a test to see if this is the 32nd repetition. If it is we run a restore and go to the "Done" step. If it is not, then we use our new logic to replace steps 3b and 2 (step 3c is added).

The following represents the new algorithm in a flow chart:



Start
1: Shift the Remainder register left 1 bit
2: Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
3a:Shift the Remainder register to the left, setting the new rightmost bit to 1
32nd Repetition?
Yes
No
3b.Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Shift the Remainder register to the left, setting the new rightmost bit to 0
3c: Shift the Remainder register to the left, setting the new rightmost bit to 0. Add the Divisor register to the left half of the Remainder register and place the result in the left half of the Remainder register
32nd Repetition?
No
Yes
Done. Shift left half of Remainder right 1 bit

The below table demonstrates the *non-restoring division* algorithm dividing 0000 0111 (7) by 0010 (2).

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	1: Shift Rem left 1	0010	0000 1110
1	2: Rem = Rem - Div	0010	1110 1110
	3c: Rem < 0, Rep 1; Shift Rem left 1, Rem0=0, Rem = Rem + Div	0010	1111 1100
2	3c: Rem < 0, Rep 1; Shift Rem left 1, Rem0=0, Rem = Rem + Div	0010	0001 1000
3	3a: Rem >= 0; Shift Rem left 1, Rem0=1	0010	0011 0001
4	2: Rem = Rem - Div	0010	0001 0001
	3a: Rem >= 0; Shift Rem left 1, Rem0=1	0010	0010 0011
5	Shift left half of Rem right 1	0010	0001 0011