COMPUTER ARCHITECTURE & ASSEMBLY LANGUAGE
14:332:331
Rutgers University
Maria Striki Spring 2021

Homework 3A

Issue Date: Fri Apr 02 2021, **Due Date**: Wed Apr 16th 2021, 8.00 pm

Total Points: 52 = 8 + 10 + 7 + 8 + 5 + 8 + 6

Your names: Ashwin Anand , John F Crespo , Jacques Trege

Exercises Allocation Table: All 3 of us contributed to each problem with completing different parts of all 7 questions

Problem 1 (8 marks)

Part A. (4 marks) Suppose A = C5 and B = 6D (both in hexadecimal). (both in hexadecimal). Show the step by step result multiplying A and B, using Booth's algorithm. Assume A and B are 8-bit two's complement integers, stored in hexadecimal format.

Part B. (2 marks) Describe in one paragraph with your own words why Booth's algorithm works. Also, what does the mythical bit in Booth's algorithm stand for? Sketch/justify why it is needed and why it works.

Part C. (2 marks). What is the RISC-V instruction that gives the upper 64 bits of the above multiplication of signed integers? What is the RISC-V instruction that gives the lower 64 bits of the above multiplication? Name one register where you will be placing the result and fill in all 32 bit positions for this instruction (write it both in assembly and in machine language).

Solution:

Part A:

Question 1.

n	A	Q	q0	Comment
8	0 0 0 0 0 0 0 0	1 1 0 0 0 1 0 1	0	initialization.
	1 0 0 1 0 0 1 1	1 1 0 0 0 1 0 1	0	A = A − M
7	1 1 0 0 1 0 0 1	1 1 1 0 0 0 1 0	1	ASR AQ
	0 0 1 1 0 1 1 0	1 1 1 0 0 0 1 0	1	A = A + M
6	0 0 0 1 1 0 1 1	0 1 1 1 0 0 0 1	0	ASR AQ
	1 0 1 0 1 1 1 0	0 1 1 1 0 0 0 1	0	A = A − M
5	1 1 0 1 0 1 1 1	0 0 1 1 1 0 0 0	1	ASR AQ0
	0 1 0 0 0 1 0 0	0 0 1 1 1 0 0 0	1	A = A + M
4	0 0 1 0 0 0 1 0	0 0 0 1 1 1 0 0	0	ASR AQ0,
3	0 0 0 1 0 0 0 1	0 0 0 0 1 1 1 0	0	ASR AQ
2	0 0 0 0 1 0 0 0	1 0 0 0 0 1 1 1	0	ASR AQ,
	1 0 0 1 1 0 1 1	1 0 0 0 0 1 1 1	0	A = A − M
1	1 1 0 0 1 1 0 1	1 1 0 0 0 0 1 1	1	ASR AQ,
0	1 1 1 0 0 1 1 0	1 1 1 0 0 0 0 1	1	ASR AQ,

1 1 1 0 0 1 1 0 1 1 1 0 0 0 0 1

$$-2^{15} + 2^{14} + 2^{13} + 2^{10} + 2^{9} + 2^{7} + 2^{6} + 2^{5} + 2^{0}$$

-6431

Part B:
The booth algorithm uses a well known fact, multiplication cannot do much to a string of 0's but it can multiply multiple times for string of 1's. The normal arithmetic operations are applicable and used for the reduction of number of operations for string of 1's by using 01111110 which can be broken down to 10000000 - 00000010 which will reduce the total number operations to 2 operations for beginning and ending of the string. Bits that are paired together to be 11 or 00 are ignored. This is because 10 and 01 are added and subtracted initially to get the correct answer. The mysterious bits known as mythical bits deal with initial least significant bit (lsb) of 0 which means no action or an initial least significant bit (lsb) of 1 which indicates 10 or subtraction for the starting of a string of 1's.

Part C:

Assuming both operands are signed, the mulh instruction providesThe lower 32 bits multiplication of the two signed integers is held in register x10. x5 and x6 hold the multiplier and multiplicand. The assemble instructions would be:

mul x7, x5, x6 //store lower 64 bits of x5*x6 in x7

mul: opcode = 0110011; rd = 00111; funct 3 = 000; rs1 = 00101; rs2 = 00110; funct7 = 0000001

Machine code: 0000001 00110 00101 000 00111 0110011

For
mulh: mulh x7, x5, x6 //store upper 64 bits of x5 * x6 in x7

mulh: opcode = 0110011; rd = 00111; funct3 = 001; rs1 = 00101; rs2 = 00110; funct7 = 0000001

Machine code: 0000001 00110 00101 001 00111 0110011

Problem 2 (10 marks)

a) **(4 marks)**. Suppose A = E4 and B = 5E (both in hexadecimal). Show the step by step result multiplying A and B, using the multiplier hardware shown in Fig. 1. Assume A and B are 8-bit unsigned numbers, stored in hexadecimal format.

b) **(2 marks).** What is the RISC-V instruction that gives the lower 32 bits of the above multiplication of signed integers? Name one register where you will be placing the result and fill in all 32 bit positions for this instruction (write it both in assembly and in machine language).

c) **(2 marks)**. Suppose for an 8-bit number, each step of operation (either addition or shift) takes 2ns. Please calculate the time necessary to perform a multiply using the approach given in Fig. 1. Assume the registers have been initialized. In hardware, please note that the shifts of the multiplicand and multiplier can be done simultaneously.

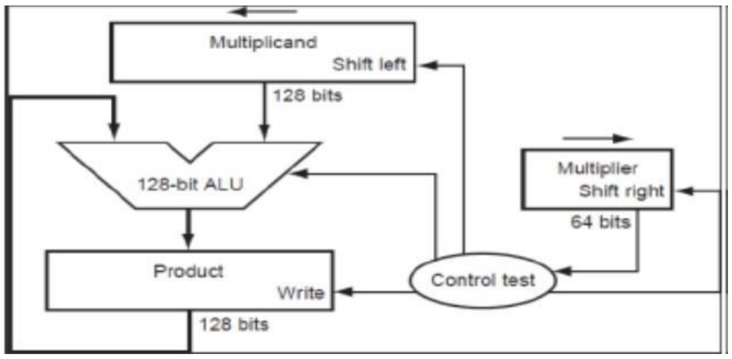


Fig. 1: HW of First (naïve) Multiplication Algorithm

d) **(2 marks).** Suppose for an 8-bit number, each step of operation (either addition or shift) takes 2ns. Please calculate the time necessary to perform a multiply using the multiplication refined approach given in Fig. 2. Assume the registers have been initialized.

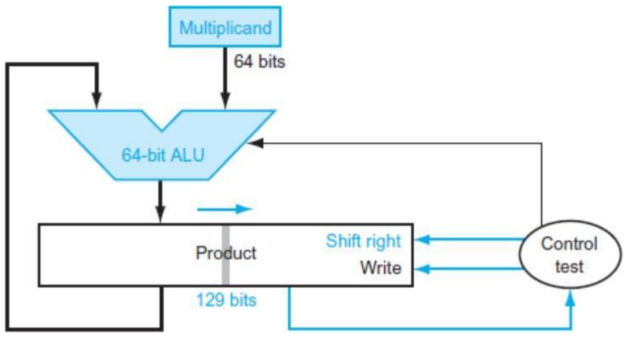


Fig. 2: Refined HW of Multiplication Algorithm

Solution:

Part A:
A = 11100100
B = 01011110

Step by Step Table:

number	Multiplier	Multiplicand	Product	Action
0	0101 1110	0000 0000 1110 0100	0000 0000 0000 0000	initial values
1	0101 1110	0000 0000 1110 0100	0000 0000 0000 0000	multiplier = 0 , none
	0010 1111	0000 0001 1100 1000	0000 0000 0000 0000	shift right multiplier, shift left multiplicand
2	0010 1111	0000 0001 1100 1000	0000 0001 1100 1000	multiplier = 1, add

				multiplicand to product
	0001 0111	0000 0011 1001 0000	0000 0001 1100 1000	shift right multiplier, shit left multiplicand
3	0001 0111	0000 0011 1001 0000	0000 0101 0101 1000	multiplier = 1, add multiplicand to product
	0000 1011	0000 0111 0010 0000	0000 0101 0101 1000	shift right multiplier, shift left multiplicand
4	0000 1011	0000 0111 0010 0000	0000 1100 0111 1000	multiplier = 1, add multiplicand to product
	0000 0101	0000 1110 0100 0000	0000 1100 0111 1000	shift right multiplier, shift left multiplicand
5	0000 0101	0000 1110 0100 0000	0001 1010 1011 1000	multiplier = 1 , add multiplicand to product
	0000 0010	0001 1100 1000 0000	0001 1010 1011 1000	shift right multiplier, shift left multiplicand
6	0000 0010	0001 1100 1000 0000	0001 1010 1011 1000	multiplier = 0 , none
	0000 0001	0011 1001 0000 0000	0001 1010 1011 1000	shift right multiplier, shift left multiplicand
7	0000 0001	0011 1001 0000 0000	0101 0011 1011 1000	multiplier = 1, add multiplicand to product
	0000 0000	0111 0010 0000 0000	0101 0011 1011 1000	shift right multiplier, shift left multiplicand
8	0000 0000	0111 0010 0000 0000	0101 0011 1011 1000	multiplier = 0, none
	0000 0000	1110 0100 0000 0000	0101 0011 1011 1000	shift right multiplier, shift left multiplicand

Part B:
The instruction that will give the lower 32 bits of above multiplication of signed integers is mulw in RISCV instruction. mulw and mul are similar instructions other than that mule saves the lower 32 bits compared to 64 bits in mul. The instruction that will give the lower 32 bits of above multiplication of signed integers is mul in RISCV instruction. The syntax for the instruction contains two source registers and one destination register. The register x10 can hold the result of lower 32 bit multiplication of two signed integers and temporary registers x6 and x5 can hold the multiplicand and multiplier. The assembly language instruction would be:

mulw x10,x5,x6

Machine language instruction read:
0000001 00110 00101 000 01010 0111011

Part C:
There are many steps involved in the operation to get to completion, to be specific there are three steps. They are, a) shift the register holding the multiplicand to the right by 1 bit, b) shift the register holding the multiplier left by 1 bit, c) add the multiplicand to the product destination register. Individually each steps takes 2ns and each number is of length 8 bits. If the operations are not performed in parallel then the time taken could be 48ns to complete. On the other hand if the operations are done in parallel like in the hardware, then 2 operations will be executed in the time of 32 ns to complete.

Part D:
Given two numbers are 8 bits each, the multipliers stored in the right half of the product register is of size 16 bits in the optimized hardware version. The multiplicand is stored in an 8 bit register. In all two operations can occur in this multiplication, they are a) multiplicand is added to the left half of the product given bit0 of product is 1, b) product register is shifted 1 bit. The two operations in all will take 32 ns to perform (8* 2* 2). If these operations are performed in parallel then the time to execute coil is 16 ns. Thus the optimized version of the hardware is faster than the unoptimized one.

Problem 3 (7 marks)

a) **(5 marks)** Suppose A = 0111 and B = 0101. Show the step by step result computing A divide B, using the hardware shown in Fig. 2. Assume A and B are 4-bit unsigned numbers. Dividend is initially loaded into the Remainder register. Please show the steps in the given table. Please expand the table if more rows needed.

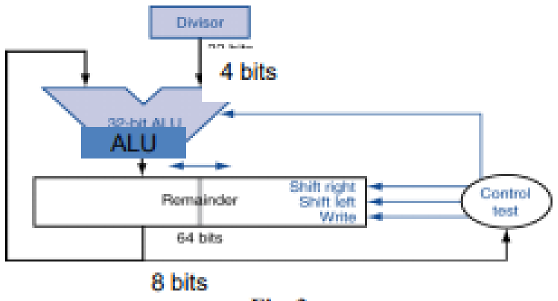


Fig. 3

b) **(2 marks)** Suppose that each operation (addition, subtraction or shift) takes 2ns. Please calculate the time necessary to perform the above division using the given hardware. Assume the registers have been initialized.

Solution:

Part A:

n	Divisor	Remainder	Operation
0	0101	0000 0111	initial values
	0101	0000 1110	Shift Remainder Left by 1 , R = 0
1	0101	1011 1110	Remainder = Remainder - Divisor
	0101	0000 1110	Remainder = Remainder + Divisor
	0101	0001 1100	Shift Remainder Left by 1 , R = 0
2	0101	1100 1100	Remainder = Remainder - Divisor
	0101	0001 1100	Remainder = Remainder + Divisor
	0101	0011 1000	Shift Remainder Left by 1 , R = 0
3	0101	1110 1000	Remainder = Remainder - Divisor
	0101	0011 1000	Remainder = Remainder + Divisor
	0101	0111 0000	Shift Remainder Left by 1, R = 0
4	0101	0010 0000	Remainder = Remainder - Divisor
	0101	0100 0001	Shift Remainder Left by 1 , R = 0
END	0101	0010 0001	Shift Remainder Right by 1

Quotient = Remainder = 0001

Remainder = 0010

Part B:

Addition operations = 3
Subtraction operations = 4
Shift operations = 6
Division = (3 + 4 + 6) * 2ns = 13 * 2s = 26 ns for full operation to perform

Problem 4: (8 marks)

a) **(4 marks)** You intend to calculate the product of the hexadecimal unsigned 8-bit integers 7E and 23 either using the hardware described in Figure 8R and 4 (refined version of the multiply) or the hardware for Booth's Algorithm multiplication. You are not asked to do the multiplication. You are asked to compare how many steps each of the two multiplication will take. How many computations each will result in? What is the saving in computations by the system that results in most savings? Now, assume that we were given numbers 8E and 23. Again, we want to use both the aforementioned schemes and compare exactly the same metrics as before, but also explain what the changes we need to make are, when:

1) 8E and 23 are unsigned.
2) 8E and 23 are signed (following 2's complement).

b) **(4 marks)** Using a table similar to that shown below (Fig 6), calculate 7E divided by 23 using the hardware in Figure 5. The numbers are hexadecimal unsigned 8-bit integers. You should show the contents of each register on each step.

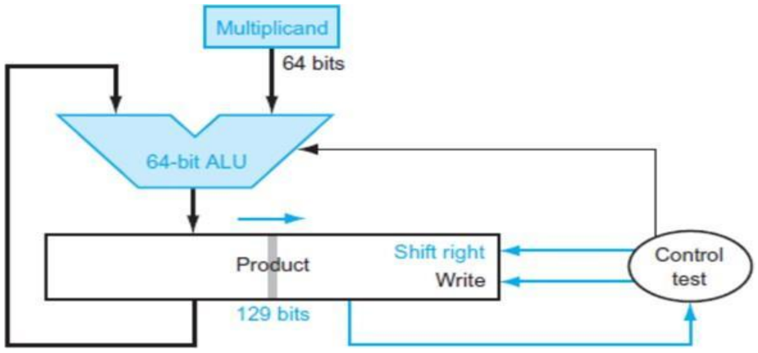


Fig. 4: Refined version of multiplication hardware.

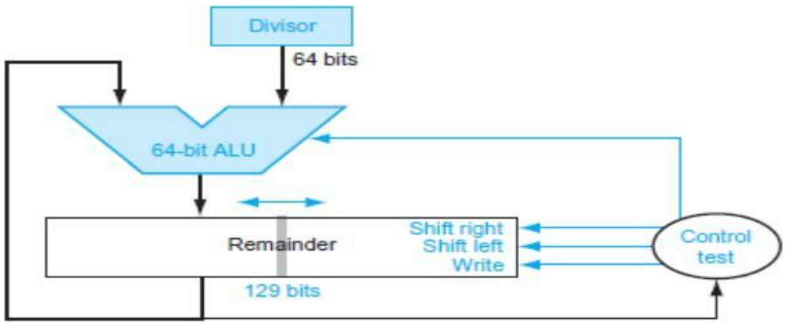
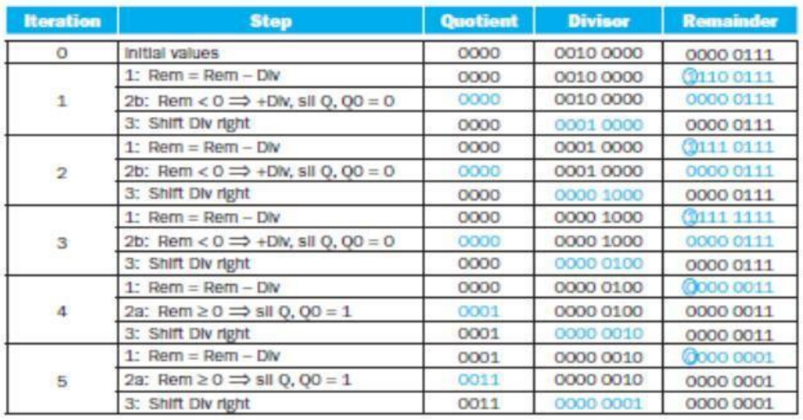


Fig 5. Refined version of division hardware.

Table 6.: Division Example using a simple divide algorithm.



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Solution:

a1) Originally considering 7E and 23 and doing unsigned multiplication, all processes should go through 7 iterations as 7E is represented by 7 bits and 23 is extended to accommodate that. The amount of steps required to complete could also depend on the number of bits of the number the user chooses to be the multiplier (not using Booths), this is assuming the user will not extend the smallest number of bits to match the longest. Sign extending the hex number 23 in its binary form makes it 8 bits long but there are 6 bits of essential data in there (2 zeros for extension that could lead to redundancy) . Thus if 23 is chosen to be the multiplier the minimum steps needed to get out a correct product is 6 (this avoids looking at redundant cases of the multiplier being 0 and still checking on the values). Now looking at 8E instead of 7E, 8E needs 8 bits to be represented and be unsigned. If you wish to accommodate the 8 bits then the consistent amount of steps needed for unsigned multiplication of these values is 8 rounds or steps for both sets of multiplication architecture and Booth's Algorithm.

a2) Originally considering 7E and 23 for signed multiplication the 2's complement of 7E must be taken in order to allow for easy steps. When we do this, 7E goes from binary 1111110 to 0000010 or 010, now 23 has the longest length of bits. Now that 23 is longest by only needing 7 bits to be properly represented, the rounds/iterations required for all processes is 7 .8E is the integer that would represent a negative number when doing signed multiplication, thus it must be made into a positive number in order to have the easiest process when proceeding. 8E after 2's complement makes the number is still 8 bits long, and still taking the length of the longest number and accommodating to it, all processes should take 8 rounds/steps.

b) Table for 7E divided by 23 (converted from Hex to Binary)

Iteration	**Step**	**Divisor**	**Remainder**
0	initialize	0100011	0000000 1111110
0	still initializing for cycles	0100011	0000001 1111100
1	rem = rem-div (only to left half)	0100011	1011110 1111100
1	if rem < 0, restore left half of remainder to original and shift left; set rightmost bit to 0	0100011	0000011 1111000
2	rem = rem-div (only to left half)	0100011	1100000 1111000
2	if rem < 0, restore left half of remainder to original and shift left; set rightmost bit to 0	0100011	0000111 1110000
3	rem = rem-div (only to left half)	0100011	1100100 1110000
3	if rem < 0, restore left half of remainder to original and shift left; set rightmost bit to 0	0100011	0001111 1100000
4	rem = rem-div (only to left half)	0100011	1101100 1100000
4	if rem < 0, restore left half of remainder to original and shift left; set rightmost bit to 0	0100011	0011111 1000000
5	rem = rem-div (only to left half)	0100011	1111100 1000000
5	if rem < 0, restore left half of remainder to original and shift left; set rightmost bit to 0	0100011	0111111 0000000
6	rem = rem-div (only to left	0100011	0011100 0000000

	half)		
6	if rem >= 0, shift remainder to the left and set rightmost bit to 1	0100011	0111000 0000001
7	rem = rem-div (only to left half)	0100011	0010101 0000001
7	if rem >= 0, shift remainder to the left and set rightmost bit to 1	0100011	0101010 0000011
Finish	Shift left half of Remainder right 1	0100011	0010101 0000011
Quotient: 0000011	Remainder: 0010101		

Problem 5: (6 points)

In RISC-V for integer instructions regarding multiplication we have the following options: `mul`, `mulh`, `mulhu, mulhsu`. You are asked to investigate which `mul` variations are available for floating point multiplication and why. In your textbook you are only provided with: `fmul.s, fmul.d, fmul.q`. Can you identify the remaining options? Are there any? Can you substitute `mulhsu` and `mulsu` with `fmul.q` perhaps? What does `fmul.q` do? Does it make sense to do this, why yes, why not?

Solution:

Some other options in RISC-V with mul commands are fmadd.s, fmadd.s, fmadd.q, fmsub.s, fmsub.d,fmsub.q, fmadd.s, fmadd.d, fmadd.q, fmnsub.s, fmnsub.d, fmnsub.q. in floating points multiplication, it is not mandatory to have 64 upper and lower bits. There is addition and multiplication of exponents respectively. The goal is to keep the most significant bits. There is not much care taken regarding the least significant bits and they are removed. The least significant bits help mostly with precision. In the case with multiplication in integers, the requirement is to store 64 most significant bits, hence this is different. In this case we need to store the most significant bits as they help with precision and are represented in the result. Ignoring this would lead to incorrect results. Based on the end target being single, double or quadruple precision requirement, the result is stored in one floating number. The instruction to get quadruple precision in multiplication of two numbers is fmul.q.

Problem 6 (8 points)

1. Write the binary representation of number 1037.379 in IEEE 754 standard in single precision. Express the result in binary, oct, and hex formats.
2. Write the binary representation of number 64.48 in IEEE 754 standard in double precision. Express the result in binary, oct, and hex formats.
3. Register f3 contains the 32-bit number 10101010 11100000 00000000 00000000. What is the corresponding signed decimal number? Assume IEEE 754 representation.

Solution:

Part 1:
1037.379 (dec) to binary
Single precision: 1 signed bit, 8 exponent, 23 fraction
1037/2 = 518 R 1
518/2 = 259 R 0
259 / 2 = 129 R 1
129 / 2 = 64 R 1
64/2 = 32 R 0
32/2 = 16 R 0
16/2 = 8 R 0
8/2 = 4 R 0
4/2 = 2 R 0
2/2 = 1 R 0 1/ 2 = 0 R 1
1037 = 10000001101
0.379 * 2 = 0.758
0.758 * 2 = 1.516
0.516 * 2 = 1.032
0.032 * 2 = 0.064
0.064 * 2 = 0.128
0.128 * 2 = 0.256
0.256 * 2 = 0.512
0.512 * 2 = 1.024
0.024 * 2 = 0.048
0.048 * 2 = 0.096
0.096 * 2 = 0.192
0.192 * 2 = 0.384
0.384 * 2 = 0.768
0.768 * 2 = 1.536
0.379 = 0.01100001000001
1037.379 = 10000001101.0110000100000 = 1.0000001101011000010000* 2^10 = 1 + .0000001101011000010000* 2^10
S = 0; E = 10+127 = 137 = 10001001; F = 0000001101011000010000
Together: SEF = 0100010010000001101011000010000 (binary)

10440326040 (Octal)
4481AC20 (Hex) 13

Part 2:
64 = 100000
0.48 * 2 = 0.96
0.96 * 2 = 1.92
0.92 * 2 = 1.84
0.84 * 2 = 1.68
0.68 * 2 = 1.36
0.36 * 2 = 0.72
0.72 * 2 = 1.44
0.44 * 2 = 0.88
0.88 * 2 = 1.76
0.76 * 2 = 1.52
0.52 * 2 = 1.04
0.04 * 2 = 0.08
0.08 * 2 = 0.16
0.16 * 2 = 0.32
0.32 * 2 = 0.64
0.64 * 2 = 1.28
0.28 * 2 = 0.56
0.56 * 2 = 1.12
0.12 * 2 = 0.24
0.24 * 2 = 0.48
0.48 * 2 = 0.96
0.96 * 2 = 1.92
0.92 * 2 = 1.84
0.84 * 2 = 1.68
0.68 * 2 = 1.36
0.36 * 2 = 0.72
0.72 * 2 = 1.44
0.44 * 2 = 0.88
0.88 * 2 = 1.76
0.76 * 2 = 1.52
0.52 * 2 = 1.04
0.04 * 2 = 0.08
0.08 * 2 = 0.16
0.16 * 2 = 0.32
0.32 * 2 = 0.64
0.64 * 2 = 1.28
0.28 * 2 = 0.56
0.56 * 2 = 1.12
0.12 * 2 = 0.24
0.24 * 2 = 0.48

0.48 * 2 = 0.96
0.96 * 2 = 1.92
0.92 * 2 = 1.84
0.84 * 2 = 1.68
0.68 * 2 = 1.36
0.36 * 2 = 0.72
0.72 * 2 = 1.44
0.44 * 2 = 0.88

0.48 = 0.011110101110000101000111101011100001010001111010
64.48 = 100000.011110101110000101000111101011100001010001111010 =
1.00000011110101110000101000111101011100001010001111010* 2^5

Double precision S = 1 bit E = 11bits F = 52 bits S = 0; E = 5+1023 = 1028 = 10000000100; F =
0000001111010111000010100011110101110000101000111101
SEF = 0100000001000000001111010111000010100011110101110000101000111101
= 401001727024365605075 (Octal)
= 40403D70A3D70A3D (Hex)

Part 3:

E = 01010101 = 85 (85-127=-42)
F = 11000000000000000000000
1.11* 2^-42 (binary)
7*2^-42 = 1.59162*10^-12 (dec)

Problem 7 (6 marks):

You are asked to design a system of FP numbers representation (with your own design choices), labeled Custom_FP_48, for which we have 48 bits at our disposal in order to represent a number, in analogy with the IEEE 754 prototype. You are asked to provide:

a) The types for evaluating this number
b) The width of representation of these numbers (upper and lower)
c) The maximum precision (i.e., the minimum difference between two successive numbers).

Solution:

a) Types of evaluation:

s	e	f
1 bit	10 bits	37 bits

The median value is bias b shown by the bits of e.

Bias = 511 (2^10 - 1 = 1023)

b) Width of representation:

The Z number value is as follows:

- If 0<e<1023, then Z = (-1)^s * 2^(e-511) * (1.f)
- If e = 1023 and f< > 0, then Z = no-number NaN ("NotANumber")
- If e = 1023 and f = 0 and s = 1, then Z = -∞
- If e = 1023 and f = 0 and s = 0, then Z = +∞
- If e = 0 and f <> 0, then Z = (-1)S * 2^-510 * (0.f) ("SubNormal")
- If e = 0 and f = 0 and s = 1, then Z = -0;
- If e = 0 and f = 0 and s = 0, then Z = 0;

c) Maximum precision

The smallest and largest numbers that can be represented become the representation width for numbers.

It shown as follows:

Zmin = ± 1.0 * 2^1-511 = ± 2^-510
Zmax= ± (1.111…1) * 2^1022-511 ≈ ± 2 * 2^511 = ± 2^512

In the case of SubNormal, the least difference between successive immediate numbers comes to
Dmin= 2^-510 * (0.000…01) = 2^-510 * 2^-37 = 2^-547.