Homework #2

Author: ZHU He (Ryker Zhu) **Student ID**: 202283930036 **Major**: Software Engineering

Exercises (The answer to the questions are all in orange): . . . .

Exercise 3.1 What is 5ED4 − 07A4 when these values represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

Solution The column method for the hexadecimal subtraction is same as the decimal one:

```
   5ED4
 − 07A4
 ------
   5730
```

. . . .

Exercise 3.3 Convert 5ED4 into a binary number. What makes base 16 (hexadecimal) an attractive numbering system for representing values in computers?

Solution The corresponding binary representation of the hexadecimal number 5ED4 is `0101 1110 1101 0100`. The reason why hexadecimal numbers are attractive is that a single digit can represent up to 16 patterns, which is enough to employ two hexadecimal digits for representing a single byte. Besides, since it is more human-readable than the original bits with shorter length, it is widely accepted.

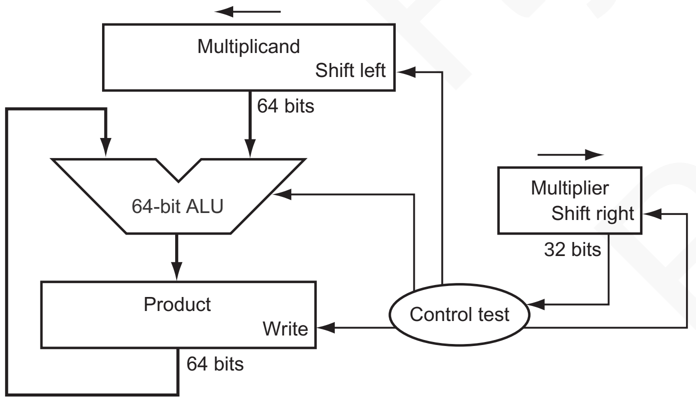
. . . .

Exercise 3.8 Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 − 122. Is there overflow, underflow, or neither?

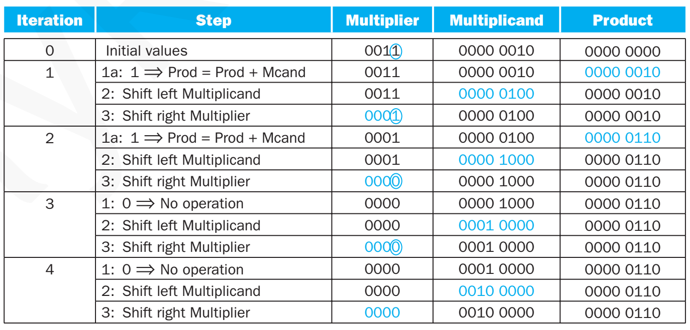
Solution What we need to calculate is $-57 - 122$, whose binary representation is `1011 1001` − `0111 1010`. By subtracting the two numbers, we would obtain `1 1011 0011`, which is a 9-bit number and beyond the capacity of an 8-bit system. There is overflow in the operation because the computed result requires more bits than available in an 8-bit system to represent the magnitude alongside the sign.

. . . .

Exercise 3.12 Using a table similar to that shown in Figure 3.6, calculate the product of the octal unsigned 6-bit integers 62 and 12 using the hardware described in Figure 3.3. You should show the contents of each register on each step.



(a) Figure 3.3 First version of the multiplication hardware



Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

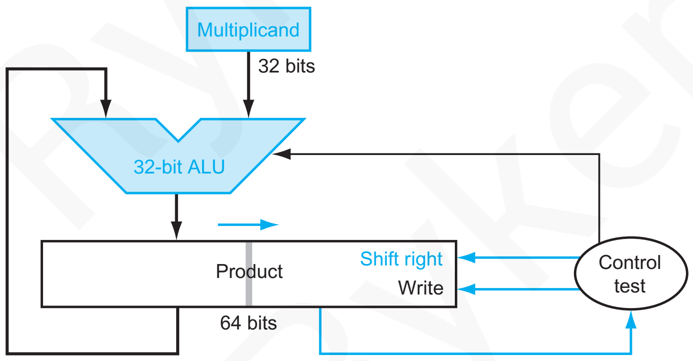
(b) Figure 3.6 Multiply example using the hardware in Fig 3.5

Solution The binary representations of the octal unsigned 6-bits integers are: $62_8 = 11\,0010_2$, $12_8 = 00\,1010_2$. Hence,

Ryker Zhu

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	00 1010	0000 0011 0010	0000 0000 0000
1	1: 0 ⇒ No operation	00 1010	0000 0011 0010	0000 0000 0000
	2: Shift left Multiplicand	00 1010	0000 0110 0100	0000 0000 0000
	3: Shift right Multiplier	00 0101	0000 0110 0100	0000 0000 0000
2	1: 1 ⇒ Prod += Mcand	00 0101	0000 0110 0100	0000 0110 0100
	2: Shift left Multiplicand	00 0101	0000 1100 1000	0000 0110 0100
	3: Shift right Multiplier	00 0010	0000 1100 1000	0000 0110 0100
3	1: 0 ⇒ No operation	00 0010	0000 1100 1000	0000 0110 0100
	2: Shift left Multiplicand	00 0010	0001 1001 0000	0000 0110 0100
	3: Shift right Multiplier	00 0001	0001 1001 0000	0000 0110 0100
4	1: 1 ⇒ Prod += Mcand	00 0001	0001 1001 0000	0001 1111 0100
	2: Shift left Multiplicand	00 0001	0011 0010 0000	0001 1111 0100
	3: Shift right Multiplier	00 0000	0011 0010 0000	0001 1111 0100
5	1: 0 ⇒ No operation	00 0000	0011 0010 0000	0001 1111 0100
	2: Shift left Multiplicand	00 0000	0110 0100 0000	0001 1111 0100
	3: Shift right Multiplier	00 0000	0110 0100 0000	0001 1111 0100
6	1: 0 ⇒ No operation	00 0000	0110 0100 0000	0001 1111 0100
	2: Shift left Multiplicand	00 0000	1100 1000 0000	0001 1111 0100
	3: Shift right Multiplier	00 0000	1100 1000 0000	0001 1111 0100

Exercise 3.13 Using a table similar to that shown in Figure 3.6, calculate the product of the octal unsigned 8-bit integers 62 and 12 using the hardware described in Figure 3.5. You should show the contents of each register on each step.



(a) Figure 3.5 Refined version of the multiplication hardware

Solution The binary representations of the octal unsigned 8-bits integers are: $62_8 = 0011\,0010_2$, $12_8 = 0000\,1010_2$. Hence,

Iteration	Step	Multiplicand	Product
0	Initial values	0011 0010	0000 0000 0000 1010

Continued on next page

Ryker Zhu

(Continued)

Iteration	Step	Multiplicand	Product
1	1: 0 ⇒ No operation	0011 0010	0000 0000 0000 1010
	2: Shift right product	0011 0010	0000 0000 0000 0101
2	1: 1 ⇒ Prod += Mcand	0011 0010	0011 0010 0000 0101
	2: Shift right product	0011 0010	0001 1001 0000 0010
3	1: 0 ⇒ No operation	0011 0010	0001 1001 0000 0010
	2: Shift right product	0011 0010	0000 1100 1000 0001
4	1: 1 ⇒ Prod += Mcand	0011 0010	0011 1110 1000 0001
	2: Shift right product	0011 0010	0001 1111 0100 0000
5	1: 0 ⇒ No operation	0011 0010	0001 1111 0100 0000
	2: Shift right product	0011 0010	0000 1111 1010 0000
6	1: 0 ⇒ No operation	0011 0010	0000 1111 1010 0000
	2: Shift right product	0011 0010	0000 0111 1101 0000
7	1: 0 ⇒ No operation	0011 0010	0000 0111 1101 0000
	2: Shift right product	0011 0010	0000 0011 1110 1000
8	1: 0 ⇒ No operation	0011 0010	0000 0011 1110 1000
	2: Shift right product	0011 0010	0000 0001 1111 0100

Exercise 3.14 Calculate the time necessary to perform a multiply using the approach given in Figure 3.3 and Figure 3.5 if an integer is 8 bits wide and each step of the operation takes 4 time units. Assume that in step 1a an addition is always performed — either the multiplicand will be added, or a zero 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.

Solution For the hardware described in Fig 3.3:

Hardware It takes one cycle to add, one cycle to shift, and one cycle to determine if the multiplication is done. Thus the loop takes 3×8 cycles, due to its 8-bit width, with 4 time units long for each cycle. Hence the hardware implementation would finally take $3 \times 8 \times 4$ time units = 96 time units.

Software It takes one cycle to decide what to add, one cycle to add, one cycle to do each shift, and one cycle to determine if the multiplication is done. So the software implementation finally takes $5 \times 8 \times 4$ time units = 160 time units.

For the hardware described in Fig 3.5:

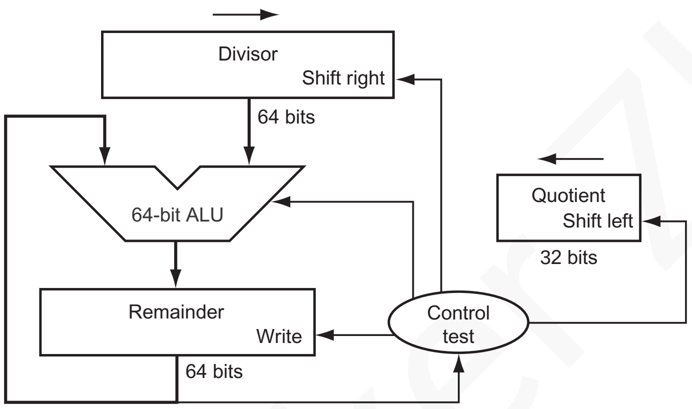
Hardware It takes one cycle to add, one cycle to shift, and one cycle to determine if the multiplication is done. Thus the loop takes 3×8 cycles, due to its 8-bit width, with 4 time units long for each cycle. Hence the hardware implementation would finally take $3 \times 8 \times 4$ time units = 96 time units.

Software It takes one cycle to decide what to add, one cycle to add, one cycle to shift, and one cycle to determine if the multiplication is done. So the software implementation finally takes $4 \times 8 \times 4$ time units = 128 time units.

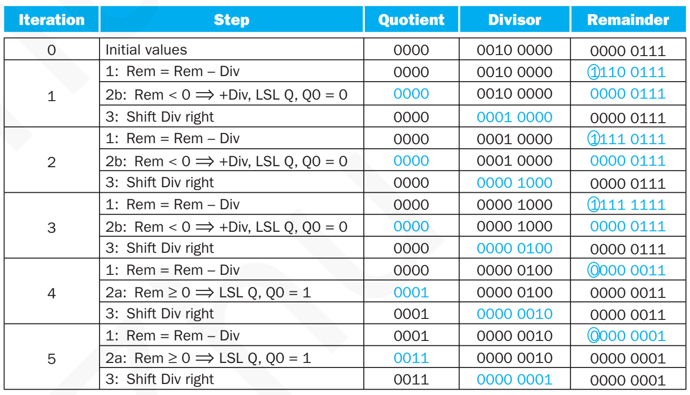
Exercise 3.15 Calculate the time necessary to perform a multiply using the approach described in the text (31 adders stacked vertically) if an integer is 8 bits wide and an adder takes 4 time units.

Solution There are 7 adders since the bitwidth is 8 and given that it takes 4 time units to perform an addition, so it will finally cost $7 \times 4 = 28$ time units.

Exercise 3.18 Using a table similar to that shown in Figure 3.10, calculate 74 divided by 21 using the hardware described in Figure 3.8. You should show the contents of each register on each step. Assume both inputs are unsigned 8-bit integers.



(a) Figure 3.8 The division hardware



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, LSL 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, LSL 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, LSL 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 ⟹ LSL 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 ⟹ LSL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

(b) Figure 3.10 Division example using the hardware in Fig 3.8

Solution The bit patterns for the two decimal numbers are: $74_{10} = 0100\,1010_2$, $21_{10} = 0001\,0101$. Hence,

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000 0000	0001 0101 0000 0000	0000 0000 0100 1010
1	1: Rem -= Div	0000 0000	0001 0101 0000 0000	1110 1011 0100 1010
	2b: Rem $< 0 \Rightarrow$ +Div, LSL Q, Q0 = 0	0000 0000	0001 0101 0000 0000	0000 0000 0100 1010
	3: Shift Div right	0000 0000	0000 1010 1000 0000	0000 0000 0100 1010
2	1: Rem -= Div	0000 0000	0000 1010 1000 0000	1111 0101 1100 1010
	2b: Rem $< 0 \Rightarrow$ +Div, LSL Q, Q0 = 0	0000 0000	0000 1010 1000 0000	0000 0000 0100 1010
	3: Shift Div right	0000 0000	0000 0101 0100 0000	0000 0000 0100 1010
3	1: Rem -= Div	0000 0000	0000 0101 0100 0000	1111 1011 0000 1010
	2b: Rem $< 0 \Rightarrow$ +Div, LSL Q, Q0 = 0	0000 0000	0000 0101 0100 0000	0000 0000 0100 1010
	3: Shift Div right	0000 0000	0000 0010 1010 0000	0000 0000 0100 1010
4	1: Rem -= Div	0000 0000	0000 0010 1010 0000	1111 1101 1010 1010
	2b: Rem $< 0 \Rightarrow$ +Div, LSL Q, Q0 = 0	0000 0000	0000 0010 1010 0000	0000 0000 0100 1010
	3: Shift Div right	0000 0000	0000 0001 0101 0000	0000 0000 0100 1010
5	1: Rem -= Div	0000 0000	0000 0001 0101 0000	1111 1101 1111 1010
	2b: Rem $< 0 \Rightarrow$ +Div, LSL Q, Q0 = 0	0000 0000	0000 0001 0101 0000	0000 0000 0100 1010
	3: Shift Div right	0000 0000	0000 0000 1010 1000	0000 0000 0100 1010

Continued on next page

(Continued)

Iteration	Step	Quotient	Divisor	Remainder
6	1: Rem -= Div	0000 0000	0000 0000 1010 1000	1111 1111 0100 0010
	2b: Rem $< 0 \Rightarrow$ +Div, LSL Q, Q0 = 0	0000 0000	0000 0000 1010 1000	0000 0000 0100 1010
	3: Shift Div right	0000 0000	0000 0000 0101 0100	0000 0000 0100 1010
7	1: Rem -= Div	0000 0000	0000 0000 0101 0100	1111 1111 1111 0110
	2b: Rem $< 0 \Rightarrow$ +Div, LSL Q, Q0 = 0	0000 0000	0000 0000 0101 0100	0000 0000 0100 1010
	3: Shift Div right	0000 0000	0000 0000 0010 1010	0000 0000 0100 1010
8	1: Rem -= Div	0000 0000	0000 0000 0010 1010	0000 0000 0010 0000
	2a: Rem $\geq 0 \Rightarrow$ LSL Q, Q0 = 1	0000 0010	0000 0000 0010 1010	0000 0000 0010 0000
	3: Shift Div right	0000 0010	0000 0000 0001 0101	0000 0000 0010 0000
9	1: Rem -= Div	0000 0010	0000 0000 0001 0101	0000 0000 0000 1011
	2a: Rem $\geq 0 \Rightarrow$ LSL Q, Q0 = 1	0000 0011	0000 0000 0001 0101	0000 0000 0000 1011
	3: Shift Div right	0000 0011	0000 0000 0000 1010	0000 0000 0000 1011

Exercise 3.20 What decimal number does the bit pattern 0x0C000000 represent if it is a two's complement integer? An unsigned integer?

Solution Since the MSB of the bit pattern is 0 so the two's complement integer and the unsigned integer is identical, which is equal to 201,326,592.

Exercise 3.21 If the bit pattern 0x0C000000 is placed into the Instruction Register, what MIPS instruction will be executed?

Solution The MIPS architecture defines the highest 6 bits are the operation code, while in this case, opcode = 3, which means it is a Jump and Link instruction according to the manual. Since the target field is 0, so the corresponding MIPS instruction is `jal 0`.

Exercise 3.22 What decimal number does the bit pattern 0x0C000000 represent if it is a floating point number? Use the IEEE 754 standard.

Solution According to the IEEE 754 single precision floating point standard, the bit pattern will be interpreted into three parts: $\underbrace{0}_{\text{sign}}\underbrace{0001\,1000}_{\text{exponent}}\underbrace{000\,0000\,0000\,0000\,0000\,0000}_{\text{fraction}}$, which means it has a positive sign with a biased exponent 24 and mantissa 0. After subtracting bias from the exponent, we can obtain the final result: $1.0 \times 2^{24-127} = 1.0 \times 2^{-103}$.

Exercise 3.23 Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 single precision format.

Solution First convert the decimal number into its equivalent binary form, $63.25_{10} = 111111.01_2$ before we normalize the binary number, which would produce 1.1111101×2^5. By adding the bias = 127, thus we obtain exponent = 5 + 127 = 132. Hence the final bit pattern is $\underbrace{0}_{\text{sign}}\underbrace{1000\,0100}_{\text{exponent}}\underbrace{111\,1101\,0000\,0000\,0000\,0000}_{\text{fraction}}$

Ryker Zhu

Exercise 3.24 Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 double precision format.

Solution The first two steps are same as the single precision one, skipped. The only difference between single and double precision is the exponent bias, and in this case, the biased exponent is $5 + 1023 = 1028$ so the final bit pattern is

$$\underbrace{0}_{\text{sign}}\ \underbrace{100\,0000\,0100}_{\text{exponent}}\ \underbrace{1111\,1010\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000}_{\text{fraction}}$$

. . . .

Exercise 3.27 IEEE 754-2008 contains a half precision that is only 16 bits wide. The leftmost bit is still the sign bit, the exponent is 5 bits wide and has a bias of 15, and the mantissa is 10 bits long. A hidden 1 is assumed. Write down the bit pattern to represent -1.5625×10^{-1} assuming a version of this format, which uses an excess-16 format to store the exponent. Comment on how the range and accuracy of this 16-bit floating point format compares to the single precision IEEE 754 standard.

Solution $-1.5625 \times 10^{-1} = -0.15625_{10} = -0.00101_2 = 1.01_2 \times 2^{-3}$. So the exponent to be represented is obtained via adding the bias of 15, i.e. $e = -3_{10} + 15_{10} = 12_{10} = 0\,1100_2$. Since the leading 1 is hidden, so the fraction would be 01. Hence, we can finally obtain its IEEE 754-2008 format $\underbrace{1}_{\text{sign}}\ \underbrace{0\,1100}_{\text{exponent}}\ \underbrace{01\,0000\,0000}_{\text{fraction}}$.

. . . .

Exercise 3.41 Using the IEEE 754 floating point format, write down the bit pattern that would represent $-\frac{1}{4}$. Can you represent $-\frac{1}{4}$ exactly?

Solution For the IEEE 754 single precision floating point format, $-\frac{1}{4}$ can be exactly represented in binary, i.e. $\underbrace{1}_{\text{sign}}\ \underbrace{0111\,1101}_{\text{exponent}}\ \underbrace{000\,0000\,0000\,0000\,0000\,0000}_{\text{fraction}}$.

. . . .

Ryker Zhu