CDA 3103: Study Set 4

ARITHMETIC LOGIC UNIT, ARITHMETIC FOR BINARY INTEGERS, REPRESENTING REAL NUMBERS, ARITHMETIC FOR REAL NUMBERS

Review: Binary Addition

Binary number can be added the same way decimal numbers are added. We begin with the least significant bits and add them. 0 and 1 can be placed in the correct position of the answer. Higher results like 2 and 3 (10 and 11 respectively) must be split up. The least significant bit is placed in the answer and the most significant bit is carried over.

3 + 2 = 5

	0	0	1	1
+	0	0	1	0
	0	1	0	1

Example: Binary Addition

Given: Add the binary values 0101 and 0001 together using binary addition.

Partial Credit 1: Stack both binary values, aligning the least significant bits to the right. Either value can be placed on top, as this does not affect our algorithm or efficiency.

	0	1	0	1
+	0	0	0	1

Solution 1:

Example: Binary Addition

<u>Given</u>: Add the binary values 0101 and 0001 together using binary addition.

<u>Partial Credit 2</u>: Begin by adding the least significant bits: those in the “one’s” position. Both of these are 1, so when we add them together we will get 10. The less significant bit, 0, will go into the answer. The more significant bit, 1, will be carried to the “two’s” position.

<u>Solution 2</u>:

			1	
	0	1	0	1
+	0	0	0	1
				0

Example: Binary Addition

Given: Add the binary values 0101 and 0001 together using binary addition.

Partial Credit 3: Repeat the process for the "two's" position. Adding 1+0+0 will result in 1. You can also think of this as 01. The 1 goes into the answer and 0 is carried to the next column.

Solution 3:

		0	1	
	0	1	0	1
+	0	0	0	1
			1	0

Example: Binary Addition

Given: Add the binary values 0101 and 0001 together using binary addition.

Partial Credit 4: Repeat the process for the "four's" position. Adding 0+1+0 will result in 1. You can also think of this as 01. The 1 goes into the answer and 0 is carried to the next column.

Solution 4:

	0	0	1	
	0	1	0	1
+	0	0	0	1
		1	1	0

Example: Binary Addition

Given: Add the binary values 0101 and 0001 together using binary addition.

Partial Credit 5: Repeat the process for the "eight's" position. In 2's complement binary you can also thing of this as the "negative eight's" position. Adding 0+0+0 will result in 0. You can also think of this as 00. A 0 goes into the answer and 0 is carried to the next column.

Solution 5:

0	0	0	1	
	0	1	0	1
+	0	0	0	1
	0	1	1	0

Review: Overflow

Overflow occurs when the result of an arithmetic operation is too large or too small to represent.

- In our 4-bit examples, that would occur if the result is less than -8 or greater than 7.
- Consider what would happen if we had 8 bits. We could express values up to 127 and down to -128.
- For any collection of N bits, 2's complement binary can represent positive values up to 2^{N-1} -1 and negative values down to -2^{N-1}.

One way to detect overflow is to compare the carry-in to the carry-out of the most significant bit. If they are the same – either both are 0 or both are 1 – then there is no overflow. If they are different then there is overflow.

Example: Binary Overflow

Given: Add the binary values 0101 and 0001 together using binary addition. Determine whether or not overflow has occurred.

Partial Credit 1: Use the same process as from binary addition. Compare the carry-in and the carry-out of the most significant bit. In this case they are both 0. This tells us no overflow has occurred.

Solution 1:

0	0	0	1	
	0	1	0	1
+	0	0	0	1
	0	1	1	0

Review: Binary Subtraction

2's complement binary numbers are subtracted by adding their negative equivalent. A – B becomes A + -B. No changes are made to A. B must be converted to a negative number: we invert each bit of B and then add 1 to the result.

Example: Binary Subtraction

Given: Suppose A = 0101 and B = 0001. Calculate A-B and state whether or not overflow has occurred.

Partial Credit 1: Calculate -B in order to calculate A + -B. Perform a bitwise inverse on B and then add 1.

Solution 1:

$$\begin{array}{rcccc} & 1 & 1 & 1 & 0 \\ + & 0 & 0 & 0 & 1 \\ \hline & 1 & 1 & 1 & 1 \end{array}$$

Example: Binary Subtraction

Given: Suppose A = 0101 and B = 0001. Calculate A-B and state whether or not overflow has occurred.

Partial Credit 2: Now, add A and –B. Start with the least significant bit; the bits in the "one's" position.

Solution 2:

$$\begin{array}{ccccc} & & & 1 & \\ & 0 & 1 & 0 & 1 \\ + & 1 & 1 & 1 & 1 \\ \hline & & & & 0 \end{array}$$

Example: Binary Subtraction

Given: Suppose A = 0101 and B = 0001. Calculate A-B and state whether or not overflow has occurred.

Partial Credit 3: Repeat with the "two's" position.

Solution 3:

		1	1	
	0	1	0	1
+	1	1	1	1
			0	0

Example: Binary Subtraction

Given: Suppose A = 0101 and B = 0001. Calculate A-B and state whether or not overflow has occurred.

Partial Credit 4: Repeat with the "four's" position.

Solution 4:

	1	1	1	
	0	1	0	1
+	1	1	1	1
		1	0	0

Example: Binary Subtraction

<u>Given</u>: Suppose A = 0101 and B = 0001. Calculate A-B and state whether or not overflow has occurred.

<u>Partial Credit 5</u>: Repeat with the “eight’s” position.

<u>Solution 5</u>:

1	1	1	1	
	0	1	0	1
+	1	1	1	1
	0	1	0	0

Example: Binary Subtraction

<u>Given</u>: Suppose A = 0101 and B = 0001. Calculate A-B and state whether or not overflow has occurred.

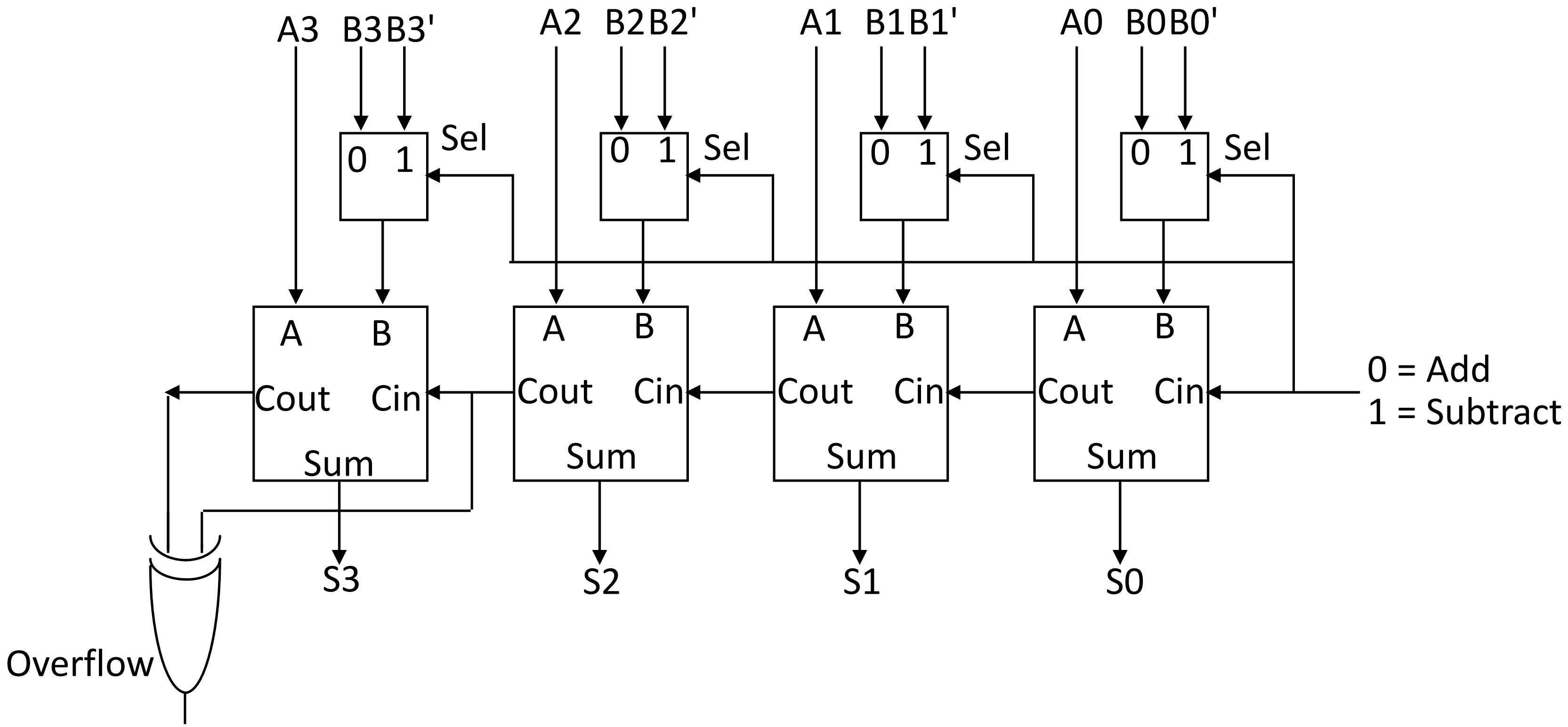
<u>Partial Credit 6</u>: To check for overflow compare the carry-in and the carry-out of the most significant bit. Since they are both 1 we determine that no overflow has occurred.

<u>Solution 6</u>:

1	1	1	1	
	0	1	0	1
+	1	1	1	1
	0	1	0	0

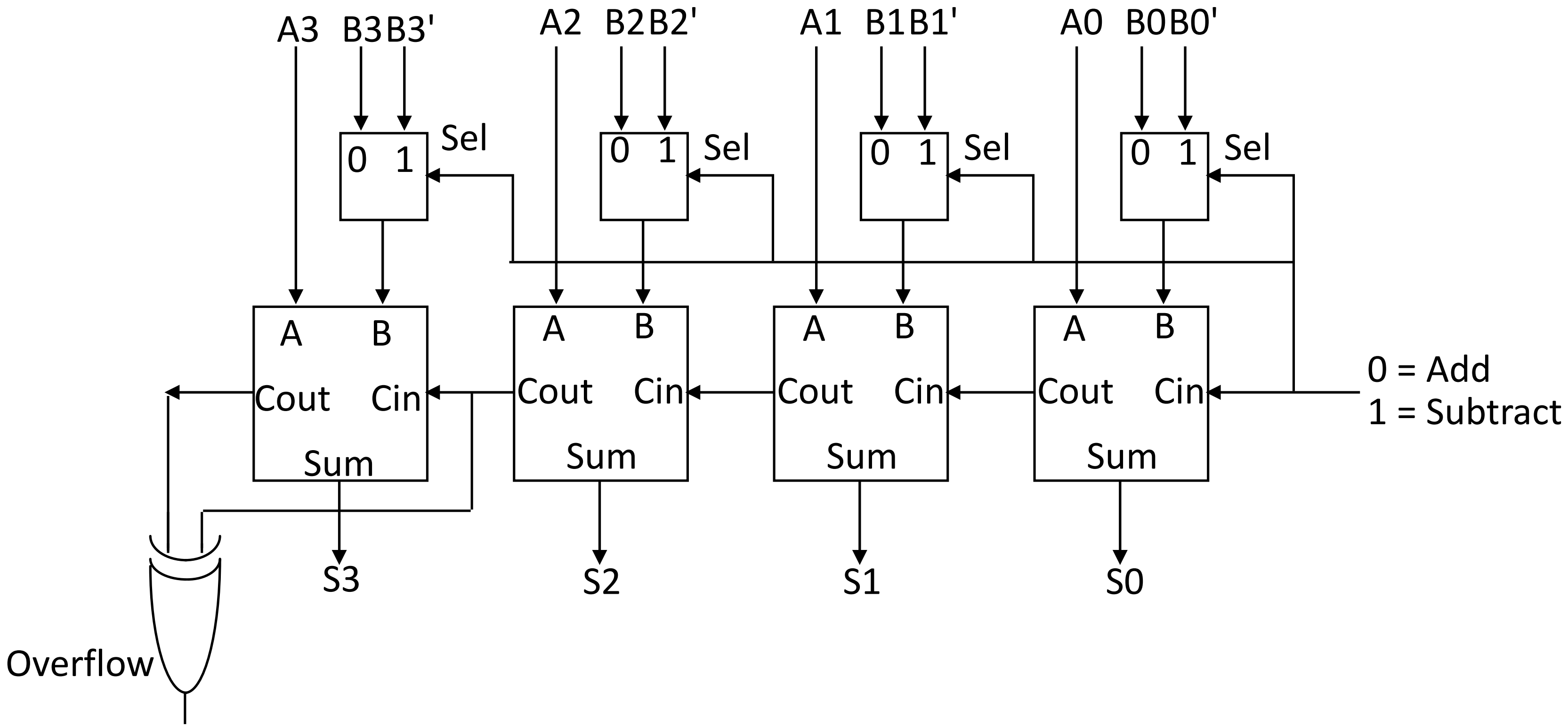
Review: Ripple-Carry Adder

The Ripple-Carry Adder is a piece of hardware that can add or subtract two numbers that are represented using 2’s complement representation of binary values. It can also detect overflow.



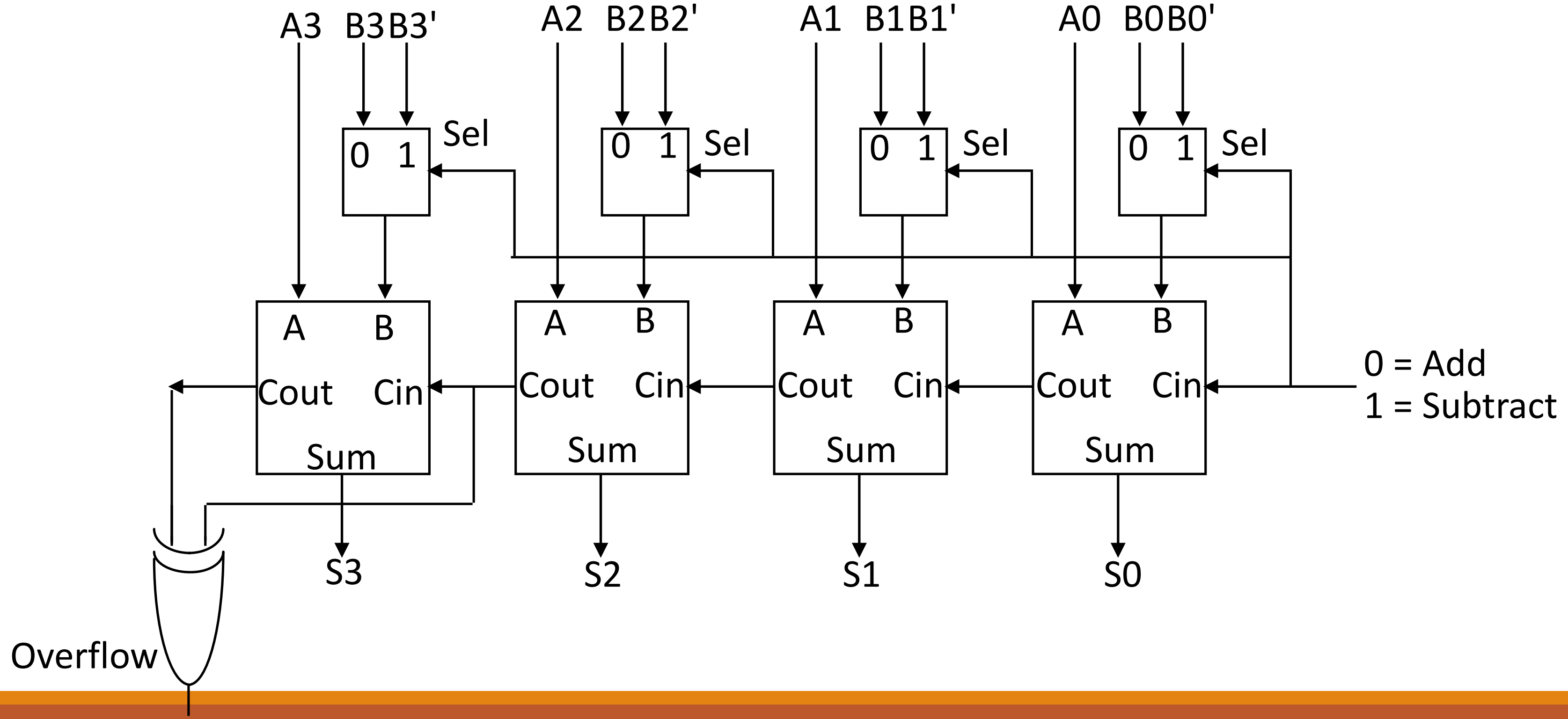
Review: Ripple-Carry Adder

We have to send a signal to the Ripple-Carry Adder to tell it which action to perform: either addition or subtraction. Addition will select A and B to be added and the initial carry-in will be 0.



Review: Ripple-Carry Adder

A subtraction signal will select the inverse of B. Each bit of B is inverted. B and B’ are sent to a 2:1 multiplexor that will choose between them. The subtraction signal also sets the initial carry-in to 1. This accomplishes both steps of calculating –B: the bitwise inverse and adding 1.



Review: Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is the brawn of the computer

- Performs arithmetic operations (addition, subtraction)
- Performs logical operations (AND, OR, NOR)
- Performs logical comparisons (Less Than, Equal To)

The ALU takes part in all multiplication and division algorithms as well

Inputs to the Arithmetic Logic Unit (ALU)

- 2 binary values (signed
- Performs logical operations (AND, OR, NOR)
- Performs logical comparisons (Less Than, Equal To)

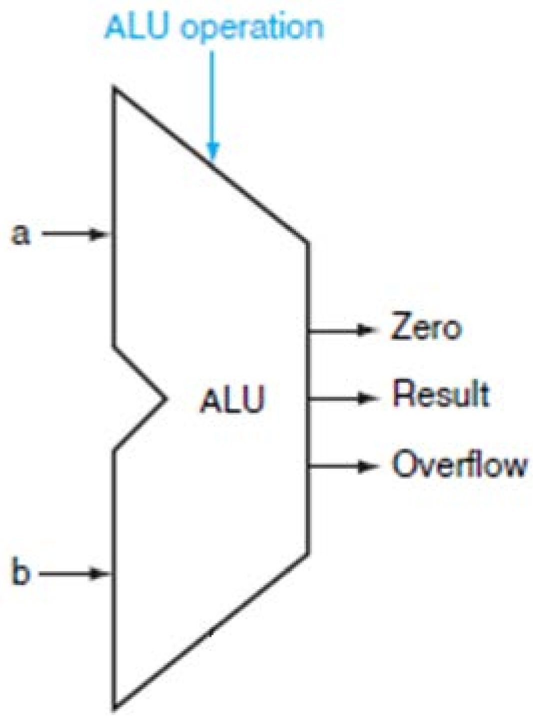
Review: Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) has three inputs:

- The two register inputs are commonly called A and B
- A and B must be the same width.
- The ALU operation determines which operation's result will be output
- ALU operation is 2-4 bits.

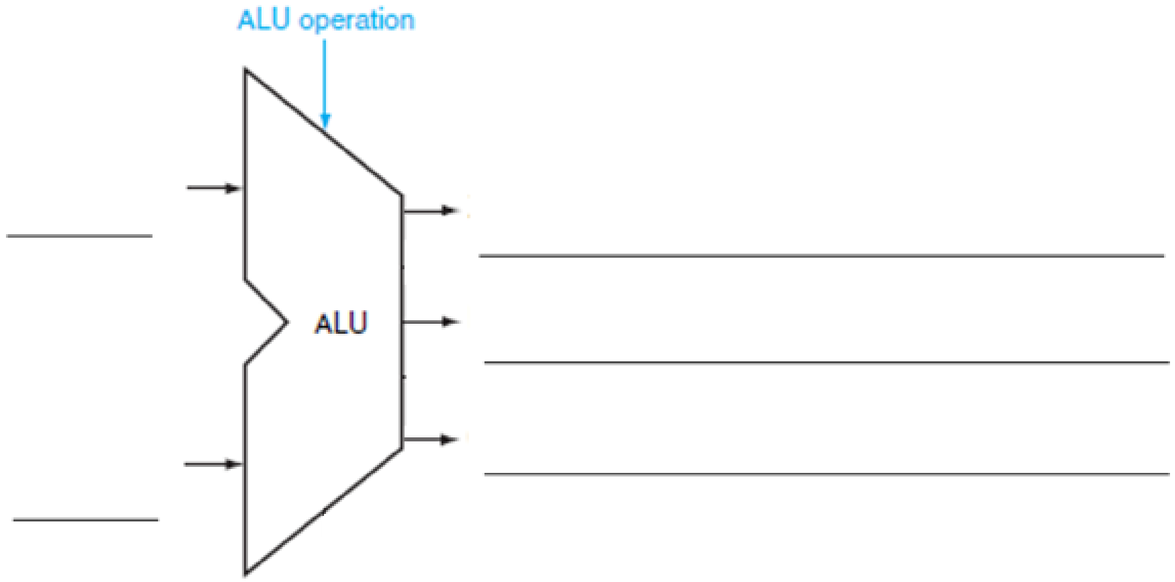
The ALU has three primary outputs

- The Result must be the same width as A and B. In a 32-bit ALU this is 32 bits. In a 16-bit ALU this will be 16 bits, etc.
- Zero is 1-bit. It will be 1 if all of the bits of Result are 0s. It will be 0 if any bit of Result is not 0.
- Overflow is also a 1-bit signal. It will be 1 if overflow is detected.



Example: Arithmetic Logic Unit

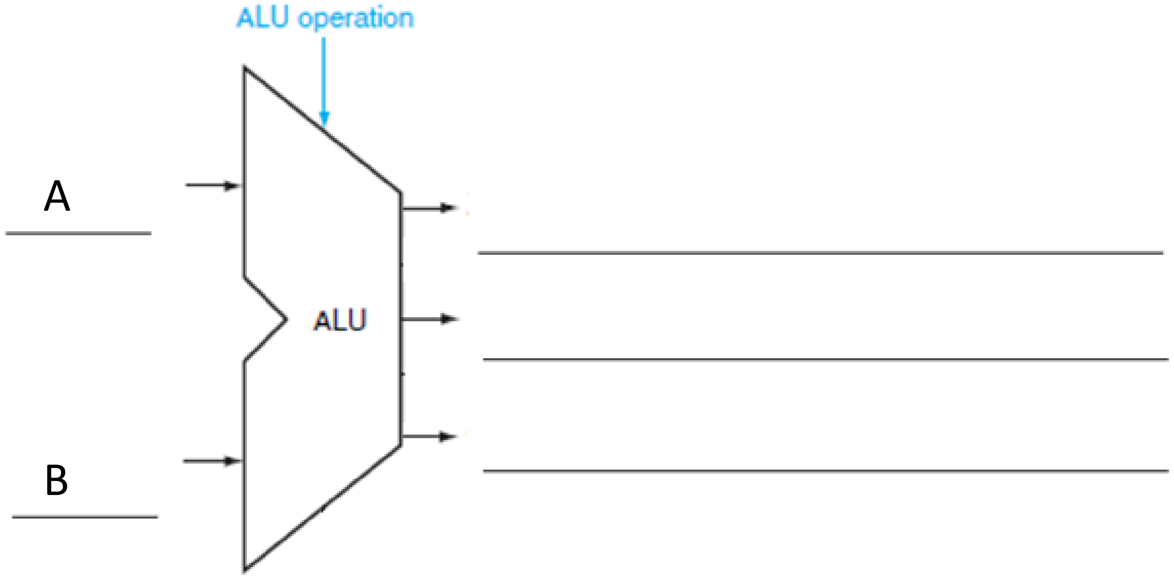
Given: Label the inputs and outputs of the following 32 bit Arithmetic Logic Unit. Identify how many bits would be used to represent each value.



Example: Arithmetic Logic Unit

Given: Label the inputs and outputs of the following 32 bit Arithmetic Logic Unit. Identify how many bits would be used to represent each value.

Partial Credit 1: Since the ALU is shown in isolation, we can give the inputs generic placeholders like A and B, X and Y, or Input 1 and Input 2. The only thing to keep in mind is that our labels should indicate they two inputs are distinct. There are two separate input buses.

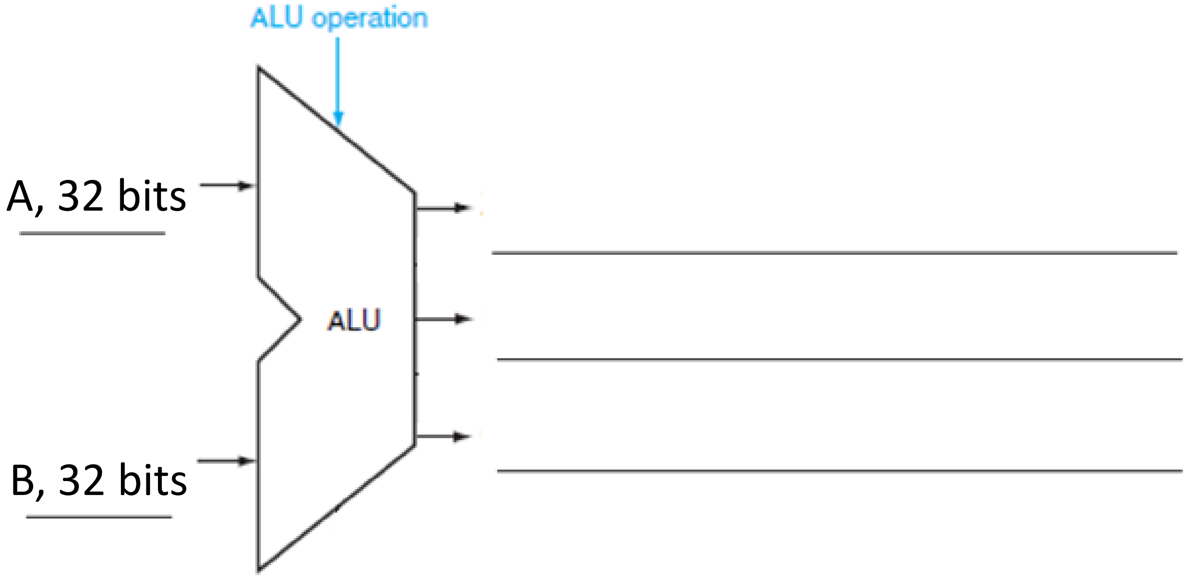


Solution 1:

Example: Arithmetic Logic Unit

Given: Label the inputs and outputs of the following 32 bit Arithmetic Logic Unit. Identify how many bits would be used to represent each value.

Partial Credit 2: The number of bits used to represent A and B are referred to as the "width" of those buses. The width of the input buses has to match the width of the ALU itself. In this case, we're told that this is a 32-bit ALU.

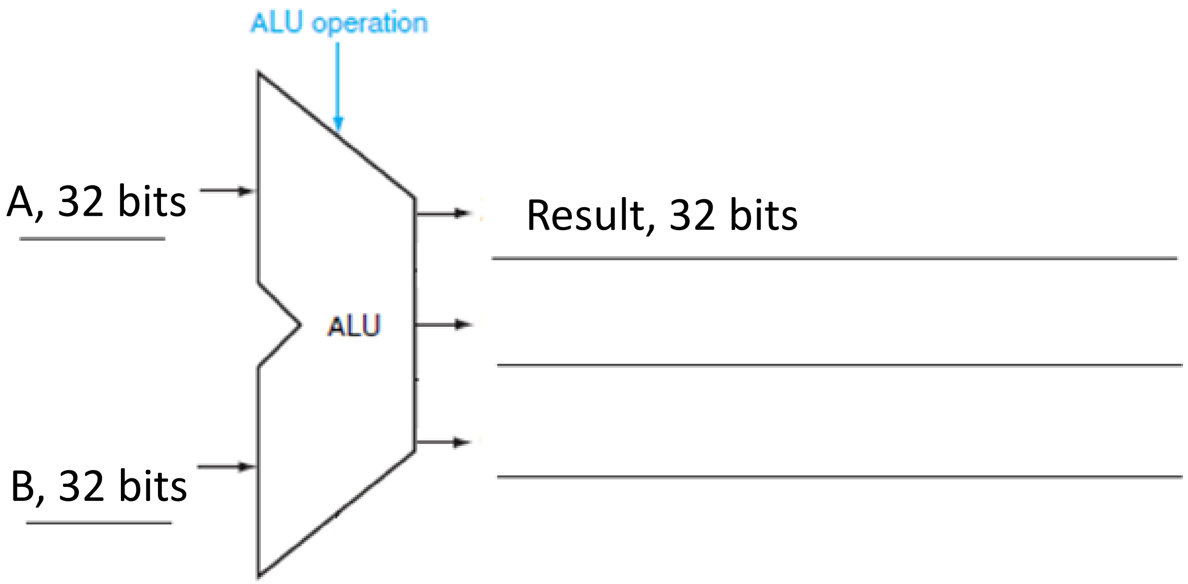


Solution 2:

Example: Arithmetic Logic Unit

Given: Label the inputs and outputs of the following 32 bit Arithmetic Logic Unit. Identify how many bits would be used to represent each value.

Partial Credit 3: There are three outputs we want to represent in our ALU diagram. The first is the result. Remember, the ALU will calculate the result of several operations at once, but only one will be sent to the output bus. This result is also the same width as the ALU.

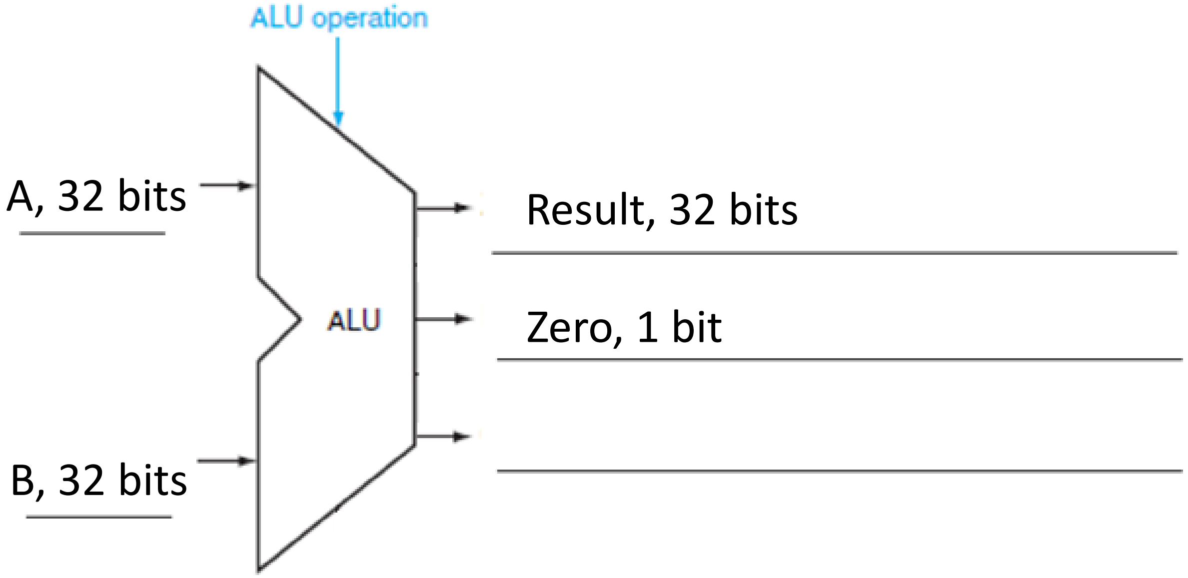


Solution 3:

Example: Arithmetic Logic Unit

Given: Label the inputs and outputs of the following 32 bit Arithmetic Logic Unit. Identify how many bits would be used to represent each value.

Partial Credit 4: Our other two outputs are Overflow and Zero. Zero indicates if the result is equal to zero or not.
This output only needs 1 bit as it will either be asserted or deasserted.

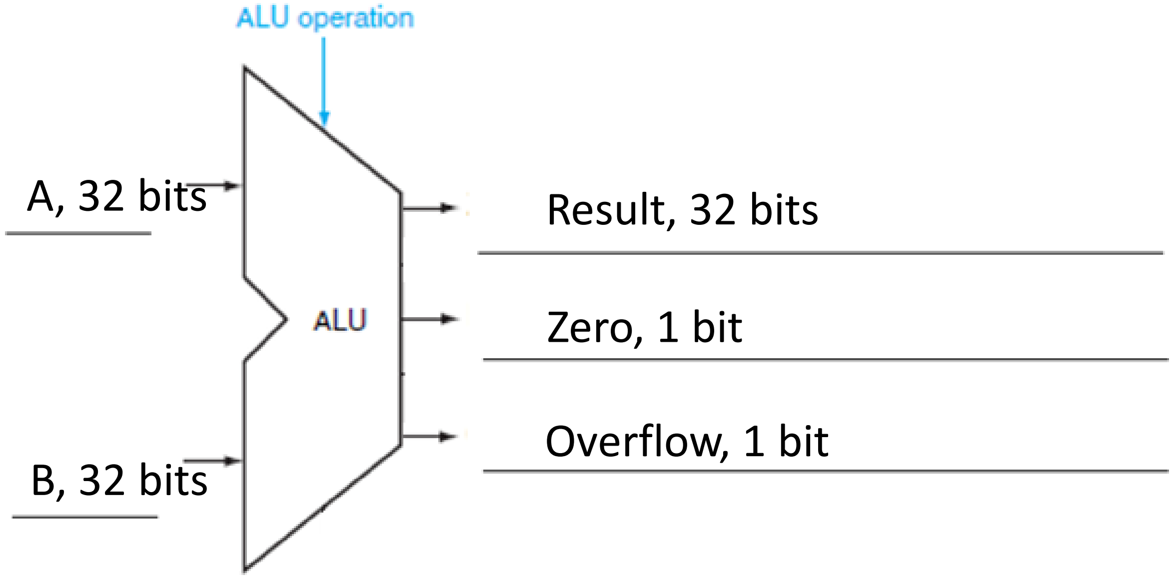


Solution 4:

Example: Arithmetic Logic Unit

Given: Label the inputs and outputs of the following 32 bit Arithmetic Logic Unit. Identify how many bits would be used to represent each value.

Partial Credit 5: Our other two outputs are Overflow and Zero. Overflow indicates if there was an overflow error in the arithmetic. This output only needs 1 bit as it will either be asserted or deasserted.



Solution 5:

Example: Arithmetic Logic Unit

<u>Given</u>: Assume we have a 8-bit Arithmetic Logic Unit. List the inputs and outputs in binary for the ALU if we are using it to determine if A = 25_{10} < B = 28_{10}. Remember: the ALU has three inputs and three outputs. The selection value will be $(11)_2$ for Set on Less Than. Use 8 bits or 1 bit to represent the remaining inputs and outputs as appropriate.

Example: Arithmetic Logic Unit

Given: Assume we have a 8-bit Arithmetic Logic Unit. List the inputs and outputs in binary for the ALU if we are using it to determine if $A = 25_{10} < B = 28_{10}$. Remember: the ALU has three inputs and three outputs. The selection value will be $(11)_2$ for Set on Less Than. Use 8 bits or 1 bit to represent the remaining inputs and outputs as appropriate.

Partial Credit 1: This question asks us to follow the same process as the ALU for completing the Set on Less Than operation. In this operation we determine if A is less than B by subtracting B from A and checking the sign. Our first step is to show A and B in 2's complement binary using 8 bits.

Solution 1: A = 0001 1001

B = 0001 1100

Example: Arithmetic Logic Unit

Given: Assume we have a 8-bit Arithmetic Logic Unit. List the inputs and outputs in binary for the ALU if we are using it to determine if A = 25_{10} < B = 28_{10}. Remember: the ALU has three inputs and three outputs. The selection value will be $(11)_2$ for Set on Less Than. Use 8 bits or 1 bit to represent the remaining inputs and outputs as appropriate.

Partial Credit 2: Now we can calculate A-B. This is the same as A + -B, so we should invert B and add 1.

Solution 2:

B = 0001 1100

B' = 1110 0011

Note that the bitwise inverse of B is not the same as B * -1.

-B = 1110 0100

Example: Arithmetic Logic Unit

Given: Assume we have a 8-bit Arithmetic Logic Unit. List the inputs and outputs in binary for the ALU if we are using it to determine if $A = 25_{10} < B = 28_{10}$. Remember: the ALU has three inputs and three outputs. The selection value will be $(11)_2$ for Set on Less Than. Use 8 bits or 1 bit to represent the remaining inputs and outputs as appropriate.

Partial Credit 3: Now we can calculate A-B. This is the same as A + -B:

Solution 3:

```
       0 0000 0000     <- This line shows the carries for each column. We can use this to
A    = 0001 1001          set the overflow output. Since the carry-in and the carry-out for
-B   = 1110 0100          the most significant bit is the same, there is no overflow.
A-B  = 1111 1101          Overflow = 0
```

Example: Arithmetic Logic Unit

Given: Assume we have a 8-bit Arithmetic Logic Unit. List the inputs and outputs in binary for the ALU if we are using it to determine if A = 25_{10} < B = 28_{10}. Remember: the ALU has three inputs and three outputs. The selection value will be $(11)_2$ for Set on Less Than. Use 8 bits or 1 bit to represent the remaining inputs and outputs as appropriate.

Partial Credit 4: Based on the sign bit of the result of A-B we can set the result of the ALU. Everything but the least significant bit is set to 0. The least significant bit is the same as the sign bit from A-B. Remember the result field should have the same number of bits as the ALU.

Solution 4: A-B = 1111 1101
Result = 0000 0001

Example: Arithmetic Logic Unit

Given: Assume we have a 8-bit Arithmetic Logic Unit. List the inputs and outputs in binary for the ALU if we are using it to determine if A = 25_{10} < B = 28_{10}. Remember: the ALU has three inputs and three outputs. The selection value will be $(11)_2$ for Set on Less Than. Use 8 bits or 1 bit to represent the remaining inputs and outputs as appropriate.

Partial Credit 5: Finally, we can set the Zero output. We can calculate it by NOR'ing all of the bits of the result. If any bit in the result is equal to 1, an 8 fan-in NOR gate will produce a 0. Only if all the bits of the result are 0 will an 8 fan-in NOR gate produce a 1.

Solution 5: Result = 0000 0001
Zero = 0 NOR 0 NOR 0 NOR 0 NOR 0 NOR 0 NOR 0 NOR 1
Zero = 0

Example: Arithmetic Logic Unit

Given: Assume we have a 8-bit Arithmetic Logic Unit. List the inputs and outputs in binary for the ALU if we are using it to determine if $A = 25_{10} < B = 28_{10}$. Remember: the ALU has three inputs and three outputs. The selection value will be $(11)_2$ for Set on Less Than. Use 8 bits or 1 bit to represent the remaining inputs and outputs as appropriate.

Partial Credit 6: As a final step, we can list each of these values so they're easy to identify visually.

Solution 6:

$A = (0001\ 1001)_2$
$B = (0001\ 1100)_2$
$\text{Result} = (0000\ 0001)_2$
$\text{Overflow} = (0)_2$
$\text{Zero} = (0)_2$

Review: Binary Multiplication

In class we discussed several methods of multiplying binary values. The paper and pencil method follows the mathematical algorithm we use for decimal integers. This is the foundation of our multiplication algorithm. We were able to reduce the hardware cost by tweaking the algorithm in small ways, combining registers, and reducing the size of some registers.

The algorithms we want to remember are the final version of this method and Booth's algorithm.

Review: Binary Multiplication

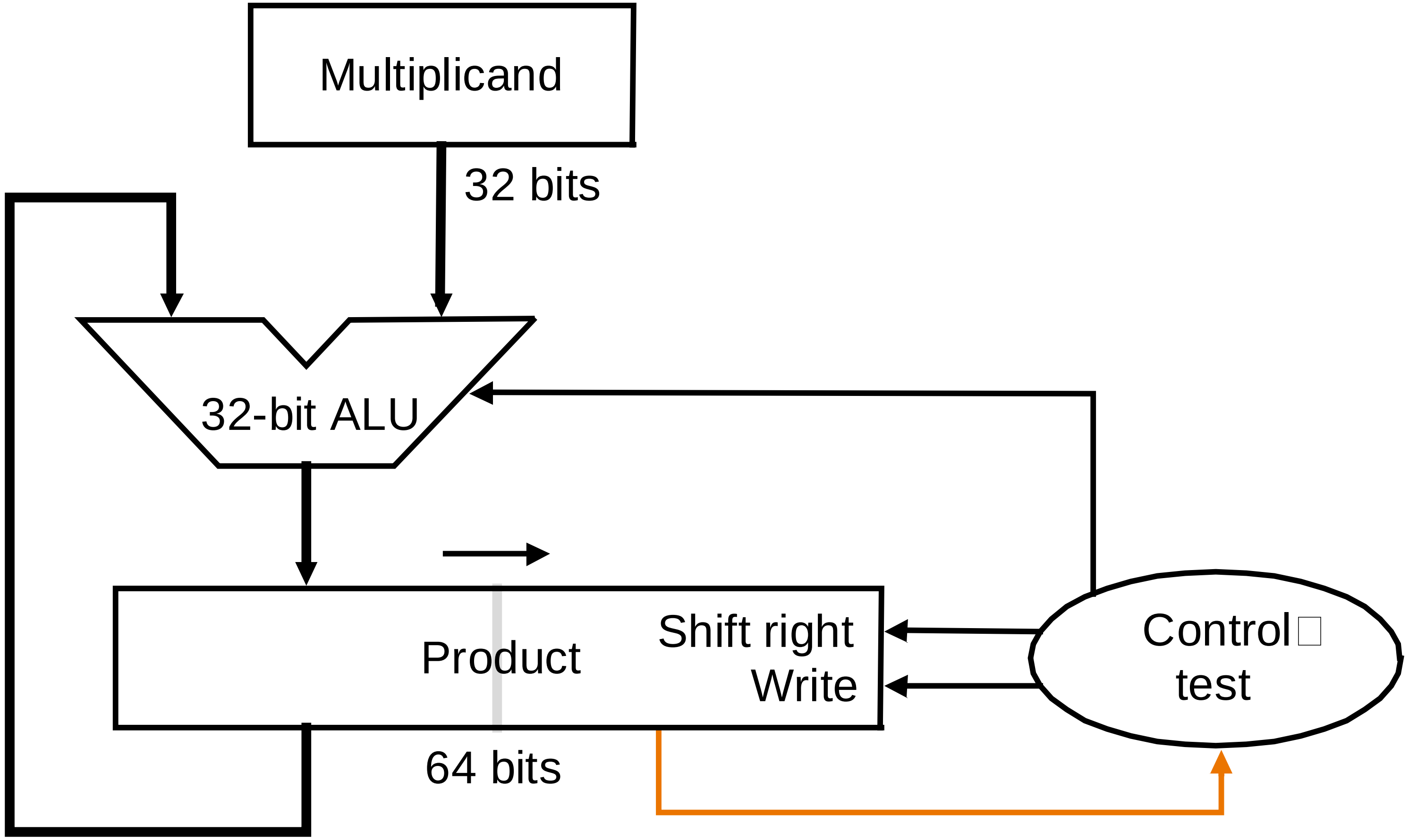
In the course material this is referred to as Multiplication Algorithm Version 3. Our hardware consists of a 32-bit multiplicand register, a 32-bit arithmetic logic unit, and a 64-bit product register.

We start by placing the multiplier in the lower half of the product register. Our action for each iteration is determined by the least significant bit of the multiplier in the product register. We will do the same number of iterations as the width of the multiplicand register. If the least significant bit is 1 we add the multiplicand to the upper half of the product register and place the result back in the upper half of the product register.

If the least significant bit is 0, we do not perform an arithmetic operation. Regardless of the value of the bit, we will shift the product register "down" or to the right one bit. This will discard the current least significant bit and a new bit will move into that position.

Review: Binary Multiplication

Hardware diagram:



Example: Binary Multiplication

Given: Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 1: We want to represent both the hardware we're using and the steps we're taking. Since we have 8 bit numbers, we should assume we will have an 8-bit multiplicand register and a 16-bit product register. The product register is always double the size of the multiplicand register. We will perform 8 iterations of the algorithm. The number of iterations is equal to the number of bits in the multiplicand register.

Solution 1:

Iter.	Step	Product	Multiplicand	Action

Example: Binary Multiplication

Given: Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 2: To initialize our hardware, we place the multiplicand in the multiplicand register. In the problem statement we are told which value to use as the multiplicand. The multiplier is placed in the lower half of the product register. The upper half of the product register is initialized to zero.

Solution 2:

Iter.	Step	Product	Multiplicand	Action
0	0	0000 0000 0110 1111	0001 1101	Initialize

Example: Binary Multiplication

Given: Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 3: Now we can start the first iteration. Each iteration has two steps. In step one we may add or do nothing based on the least significant bit of the product register. In step two we shift the product register to the right one bit.

Since the least significant bit is 1, we add the multiplicand to the upper half of the produce register and place the result in the upper half of the product register. In other words, we add 0001 1101 and 0000 000 and place 0001 1101 in the upper half of the product register.

Solution 3:

Iter.	Step	Product	Multiplicand	Action
0	0	0000 0000 0110 1111	0001 1101	Initialize
1	1a.	0001 1101 0110 1111	0001 1101	Add
1	2	0000 1110 1011 0111	0001 1101	Shift

Example: Binary Multiplication

Given: Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Solution 4:

Iter.	Step	Product	Multiplicand	Action
0	0	0000 0000 0110 1111	0001 1101	Initialize
1	1a.	0001 1101 0110 1111	0001 1101	Add
1	2	0000 1110 1011 0111	0001 1101	Shift
2	1a.	0010 1011 1011 0111	0001 1101	Add
2	2	0001 0101 1101 1011	0001 1101	Shift

Partial Credit 4: In the second iteration, the least significant bit is again 1.

We need to add the multiplicand to the upper half of the product register:

0000 1110
<u>0001 1101</u>
0010 1011

Example: Binary Multiplication

Given: Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Solution 5:

Iter.	Step	Product	Multiplicand	Action
0	0	0000 0000 0110 1111	0001 1101	Initialize
1	1a.	0001 1101 0110 1111	0001 1101	Add
1	2	0000 1110 1011 0111	0001 1101	Shift
2	1a.	0010 1011 1011 0111	0001 1101	Add
2	2	0001 0101 1101 1011	0001 1101	Shift
3	1a.	0011 0010 1101 1011	0001 1101	Add
3	2	0001 1001 0110 1101	0001 1101	Shift

Partial Credit 5:

In the third iteration, the least significant bit is again 1.

We need to add the multiplicand to the upper half of the product register:

0001 0101
<u>0001 1101</u>
0011 0010

Example: Binary Multiplication

Given: Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 6:

In the 4th iteration, the least significant bit is again 1.

We need to add the multiplicand to the upper half of the product register:

0001 1001
<u>0001 1101</u>
0011 0110

Solution 6:

Iter.	Step	Product	Multiplicand	Action
0	0	0000 0000 0110 1111	0001 1101	Initialize
1	1a.	0001 1101 0110 1111	0001 1101	Add
1	2	0000 1110 1011 0111	0001 1101	Shift
2	1a.	0010 1011 1011 0111	0001 1101	Add
2	2	0001 0101 1101 1011	0001 1101	Shift
3	1a.	0011 0010 1101 1011	0001 1101	Add
3	2	0001 1001 0110 1101	0001 1101	Shift
4	1a.	0011 0110 0110 1101	0001 1101	Add
4	2	0001 1011 0011 0110	0001 1101	Shift

Example: Binary Multiplication

Given: Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Solution 7:

Iter.	Step	Product	Multiplicand	Action
0	0	0000 0000 0110 1111	0001 1101	Initialize
1	1a.	0001 1101 0110 1111	0001 1101	Add
1	2	0000 1110 1011 0111	0001 1101	Shift
2	1a.	0010 1011 1011 0111	0001 1101	Add
2	2	0001 0101 1101 1011	0001 1101	Shift
3	1a.	0011 0010 1101 1011	0001 1101	Add
3	2	0001 1001 0110 1101	0001 1101	Shift
4	1a.	0011 0110 0110 1101	0001 1101	Add
4	2	0001 1011 0011 0110	0001 1101	Shift
5	1a.	0001 1011 0011 0110	0001 1101	No action
5	2	0000 1101 1001 1011	0001 1101	Shift

Partial Credit 7:

In the 5th iteration, the least significant bit is 0.

We do not need to do any arithmetic in this case and we can proceed to the shift step.

Example: Binary Multiplication

Given:

Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 8:

In the 6th iteration, the least significant bit is 1, so we need to add.

0000 1101
0001 1101
0010 1010

Solution 8:

Iter.	Step	Product	Multiplicand	Action
0	0	0000 0000 0110 1111	0001 1101	Initialize
1	1a.	0001 1101 0110 1111	0001 1101	Add
1	2	0000 1110 1011 0111	0001 1101	Shift
2	1a.	0010 1011 1011 0111	0001 1101	Add
2	2	0001 0101 1101 1011	0001 1101	Shift
3	1a.	0011 0010 1101 1011	0001 1101	Add
3	2	0001 1001 0110 1101	0001 1101	Shift
4	1a.	0011 0110 0110 1101	0001 1101	Add
4	2	0001 1011 0011 0110	0001 1101	Shift
5	1a.	0001 1011 0011 0110	0001 1101	No action
5	2	0000 1101 1001 1011	0001 1101	Shift
6	1a.	0010 1010 1001 1011	0001 1101	Add
6	2	0001 0101 0100 1101	0001 1101	Shift

Example: Binary Multiplication

Solution 9:

Given:

Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 9:

In the 7th iteration, the least significant bit is 1, so we need to add.

0001 0101
<u>0001 1101</u>
0011 0010

Iter.	Step	Product	Multiplicand	Action
0	0	0000 0000 0110 1111	0001 1101	Initialize
1	1a.	0001 1101 0110 1111	0001 1101	Add
1	2	0000 1110 1011 0111	0001 1101	Shift
2	1a.	0010 1011 1011 0111	0001 1101	Add
2	2	0001 0101 1101 1011	0001 1101	Shift
3	1a.	0011 0010 1101 1011	0001 1101	Add
3	2	0001 1001 0110 1101	0001 1101	Shift
4	1a.	0011 0110 0110 1101	0001 1101	Add
4	2	0001 1011 0011 0110	0001 1101	Shift
5	1a.	0001 1011 0011 0110	0001 1101	No action
5	2	0000 1101 1001 1011	0001 1101	Shift
6	1a.	0010 1010 1001 1011	0001 1101	Add
6	2	0001 0101 0100 1101	0001 1101	Shift
7	1a.	0011 0010 0100 1101	0001 1101	Add
7	2	0001 1001 0010 0110	0001 1101	Shift

Example: Binary Multiplication

Given: Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 10: In the 8th iteration, the least significant bit is 0.

We do not need to do any arithmetic in this case and we can proceed to the shift step.

Solution 10:

Iter.	Step	Product	Multiplicand	Action
0	0	0000 0000 0110 1111	0001 1101	Initialize
1	1a.	0001 1101 0110 1111	0001 1101	Add
1	2	0000 1110 1011 0111	0001 1101	Shift
2	1a.	0010 1011 1011 0111	0001 1101	Add
2	2	0001 0101 1101 1011	0001 1101	Shift
3	1a.	0011 0010 1101 1011	0001 1101	Add
3	2	0001 1001 0110 1101	0001 1101	Shift
4	1a.	0011 0110 0110 1101	0001 1101	Add
4	2	0001 1011 0011 0110	0001 1101	Shift
5	1	0001 1011 0011 0110	0001 1101	No action
5	2	0000 1101 1001 1011	0001 1101	Shift
6	1a.	0010 1010 1001 1011	0001 1101	Add
6	2	0001 0101 0100 1101	0001 1101	Shift
7	1a.	0011 0010 0100 1101	0001 1101	Add
7	2	0001 1001 0010 0110	0001 1101	Shift
8	1	0001 1001 0010 0110	0001 1101	No action
8	2	0000 1100 1001 0011	0001 1101	Shift

Example: Binary Multiplication

Given:

Determine AxB using the third multiplication algorithm for the following 8-bit numbers. Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 10:

We have completed all the iterations.

Our final answer is that AxB = $(0000\ 1100\ 1001\ 0011)_2$

Solution 10:

Iter.	Step	Product	Multiplicand	Action
0	0	0000 0000 0110 1111	0001 1101	Initialize
1	1a.	0001 1101 0110 1111	0001 1101	Add
1	2	0000 1110 1011 0111	0001 1101	Shift
2	1a.	0010 1011 1011 0111	0001 1101	Add
2	2	0001 0101 1101 1011	0001 1101	Shift
3	1a.	0011 0010 1101 1011	0001 1101	Add
3	2	0001 1001 0110 1101	0001 1101	Shift
4	1a.	0011 0110 0110 1101	0001 1101	Add
4	2	0001 1011 0011 0110	0001 1101	Shift
5	1	0001 1011 0011 0110	0001 1101	No action
5	2	0000 1101 1001 1011	0001 1101	Shift
6	1a.	0010 1010 1001 1011	0001 1101	Add
6	2	0001 0101 0100 1101	0001 1101	Shift
7	1a.	0011 0010 0100 1101	0001 1101	Add
7	2	0001 1001 0010 0110	0001 1101	Shift
8	1	0001 1001 0010 0110	0001 1101	No action
8	2	0000 1100 1001 0011	0001 1101	Shift

Review: Booth's Algorithm

Notice how in the previous example both of our input values were positive. The multiplication algorithm only supports positive values. If we have a negative number we have to convert it to positive, perform the multiplication, and then adjust the sign of the result if needed.

Booth's Algorithm is a different method that works for negative as well as positive values. We need to identify when a series of 1's begins and ends (called a "run" of 1's). When a run begins, we will subtract the multiplicand. When a run ends, we will add the multiplicand. In the middle of a run we do not need to do any arithmetic. Similarly, in the middle of a series of zeros we do not need to do any arithmetic.

In our shift step we need to perform an arithmetic shift. We need to preserve the sign bit of the product register by shifting in a copy of the most significant bit (the sign bit).

This algorithm uses the same hardware as the previous one.

Example: Booth’s Algorithm

Solution 1:

Iter.	Step	Product	Previous	Action

Given:

Determine AxB using Booth’s algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 1:

The setup for Booth’s algorithm similar to the previous algorithm.

We also need to track the previous least significant bit so we identify the beginning and end of runs.

Example: Booth's Algorithm

Solution2:

Iter.	Step	Product	Previous	Action

Given:

Determine AxB using Booth's algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 2:

It's also helpful to keep track of the multiplicand and the result of multiplicand * -1.

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Note: only the multiplicand is stored in the hardware. The ALU is capable of calculating A-B without storing –B.

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth's Algorithm

Solution 3:

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize

Given:

Determine AxB using Booth's algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 3:

To identify runs we compare the least significant bit of the product register with the "previous" bit.

These two bits together form a pattern. In this case: 10.

This tells we are starting a run of 1's.

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth’s Algorithm

Solution 3:

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract

Given:

Determine AxB using Booth’s algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 3:

Since we are starting a run of 1’s we are going to subtract the multiplicand from the upper half of the product register and place the result in the upper half of the product register.

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth's Algorithm

Solution 3:

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract
1	2	1111 0001 1011 0111	1	Shift

Given:

Determine AxB using Booth's algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 3:

After the arithmetic, we need to perform an arithmetic shift. Since the most significant bit is 1, we need to shift in a 1.

The current least significant bit of the product is shifted into previous.

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth's Algorithm

Solution 4:

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract
1	2	1111 0001 1011 0111	1	Shift
2	1.11	1111 0001 1011 0111	1	No action

Given:

Determine AxB using Booth's algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 4:

To start the 2nd iteration, compare the least significant bit of the product with previous. Since they are both 1, we are in the middle of a run and do not need to do any arithmetic.

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth’s Algorithm

Solution 4:

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract
1	2	1111 0001 1011 0111	1	Shift
2	1.11	1111 0001 1011 0111	1	No action
2	2	1111 1000 1101 1011	1	Shift

Given:

Determine AxB using Booth’s algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 4:

In our shift step, we need to again shift in a 1 because the most significant bit of the product is a 1.

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth's Algorithm

Solution 5:

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract
1	2	1111 0001 1011 0111	1	Shift
2	1.11	1111 0001 1011 0111	1	No action
2	2	1111 1000 1101 1011	1	Shift
3	1.11	1111 1000 1101 1011	1	No action
3	2	1111 1100 0110 1101	1	Shift

Given: Determine AxB using Booth's algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 5: Our 3rd iteration is similar. We are still in the middle of a run of 1's. So we can proceed directly to the shift step. Remember to shift in a copy of the most significant bit of the product.

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth's Algorithm

Solution 6:

Given:

Determine AxB using Booth's algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract
1	2	1111 0001 1011 0111	1	Shift
2	1.11	1111 0001 1011 0111	1	No action
2	2	1111 1000 1101 1011	1	Shift
3	1.11	1111 1000 1101 1011	1	No action
3	2	1111 1100 0110 1101	1	Shift
4	1.11	1111 1100 0110 1101	1	No action
4	2	1111 1110 0011 0110	1	Shift

Partial Credit 6:

Our 4th iteration is similar. We are still in the middle of a run of 1's. So we can proceed directly to the shift step. Remember to shift in a copy of the most significant bit of the product.

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth's Algorithm

Given:

Determine AxB using Booth's algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 7:

Our 5th iteration introduces a new patter: 01. This tells us that we are ending a run of 1's and need to add:

1111 1110
0001 1101
0001 1011

Then we can shift in a copy of the new most significant bit of the product register.

Solution 7:

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract
1	2	1111 0001 1011 0111	1	Shift
2	1.11	1111 0001 1011 0111	1	No action
2	2	1111 1000 1101 1011	1	Shift
3	1.11	1111 1000 1101 1011	1	No action
3	2	1111 1100 0110 1101	1	Shift
4	1.11	1111 1100 0110 1101	1	No action
4	2	1111 1110 0011 0110	1	Shift
5	1.01	0001 1011 0011 0110	1	Add
5	2	0000 1101 1001 1011	0	Shift

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth's Algorithm

Solution 8:

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract
1	2	1111 0001 1011 0111	1	Shift
2	1.11	1111 0001 1011 0111	1	No action
2	2	1111 1000 1101 1011	1	Shift
3	1.11	1111 1000 1101 1011	1	No action
3	2	1111 1100 0110 1101	1	Shift
4	1.11	1111 1100 0110 1101	1	No action
4	2	1111 1110 0011 0110	1	Shift
5	1.01	0001 1011 0011 0110	1	Add
5	2	0000 1101 1001 1011	0	Shift
6	1.10	1111 0000 1001 1011	0	Subtract
6	2	1111 1000 0100 1101	1	Shift

Given: Determine AxB using Booth's algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 8: In the 6th iteration we start a new run, so we subtract:

0000 1101
<u>1110 0011</u>
1111 0000

Then we can shift in a copy of the new most significant bit of the product register.

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth’s Algorithm

Given: Determine AxB using Booth’s algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 9: In the 7th iteration we continue our run and shift in a copy of the most significant bit.

Solution 9:

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract
1	2	1111 0001 1011 0111	1	Shift
2	1.11	1111 0001 1011 0111	1	No action
2	2	1111 1000 1101 1011	1	Shift
3	1.11	1111 1000 1101 1011	1	No action
3	2	1111 1100 0110 1101	1	Shift
4	1.11	1111 1100 0110 1101	1	No action
4	2	1111 1110 0011 0110	1	Shift
5	1.01	0001 1011 0011 0110	1	Add
5	2	0000 1101 1001 1011	0	Shift
6	1.10	1111 0000 1001 1011	0	Subtract
6	2	1111 1000 0100 1101	1	Shift
7	1.11	1111 1000 0100 1101	1	No action
7	2	1111 1100 0010 0110	1	Shift

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth's Algorithm

Given:

Determine AxB using Booth's algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 10:

In the 8th iteration we finish our run and need to add:

1111 1100
0001 1101
0001 1001

Then we can shift in a copy of the new most significant bit of the product register.

Solution 10:

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract
1	2	1111 0001 1011 0111	1	Shift
2	1.11	1111 0001 1011 0111	1	No action
2	2	1111 1000 1101 1011	1	Shift
3	1.11	1111 1000 1101 1011	1	No action
3	2	1111 1100 0110 1101	1	Shift
4	1.11	1111 1100 0110 1101	1	No action
4	2	1111 1110 0011 0110	1	Shift
5	1.01	0001 1011 0011 0110	1	Add
5	2	0000 1101 1001 1011	0	Shift
6	1.10	1111 0000 1001 1011	0	Subtract
6	2	1111 1000 0100 1101	1	Shift
7	1.11	1111 1000 0100 1101	1	No action
7	2	1111 1100 0010 0110	1	Shift
8	1.01	0001 1001 0010 0110	1	Add
8	2	0000 1100 1001 0011	0	Shift

Multiplicand = 0001 1101
-Multiplicand = 1110 0011

Example: Booth's Algorithm

Solution 11:

Given: Determine AxB using Booth's algorithm for the following 8-bit numbers.
Use A = $(0110\ 1111)_2$ for the multiplier and B = $(0001\ 1101)_2$ for the multiplicand.

Partial Credit 11: We have completed all the iterations.

Our final answer is that AxB = $(0000\ 1100\ 1001\ 0011)_2$

Iter.	Step	Product	Previous	Action
0	0	0000 0000 0110 1111	0	Initalize
1.	1.10	1110 0011 0110 1111	0	Subtract
1	2	1111 0001 1011 0111	1	Shift
2	1.11	1111 0001 1011 0111	1	No action
2	2	1111 1000 1101 1011	1	Shift
3	1.11	1111 1000 1101 1011	1	No action
3	2	1111 1100 0110 1101	1	Shift
4	1.11	1111 1100 0110 1101	1	No action
4	2	1111 1110 0011 0110	1	Shift
5	1.01	0001 1011 0011 0110	1	Add
5	2	0000 1101 1001 1011	0	Shift
6	1.10	1111 0000 1001 1011	0	Subtract
6	2	1111 1000 0100 1101	1	Shift
7	1.11	1111 1000 0100 1101	1	No action
7	2	1111 1100 0010 0110	1	Shift
8	1.01	0001 1001 0010 0110	1	Add
8	2	0000 1100 1001 0011	0	Shift

Example: Booth’s Algorithm

Compare and contrast these two examples for multiplying A and B. Which algorithm uses more arithmetic operations? Which algorithm is more efficient for AxB? Recall that shifts are more efficient than adds.

Note how we arrived at the same answer with both algorithms. Go back and count how many arithmetic operations are performed for each. These are just the addition and subtraction operations. The algorithm with fewer arithmetic operations will ultimately perform less steps and is considered more efficient.

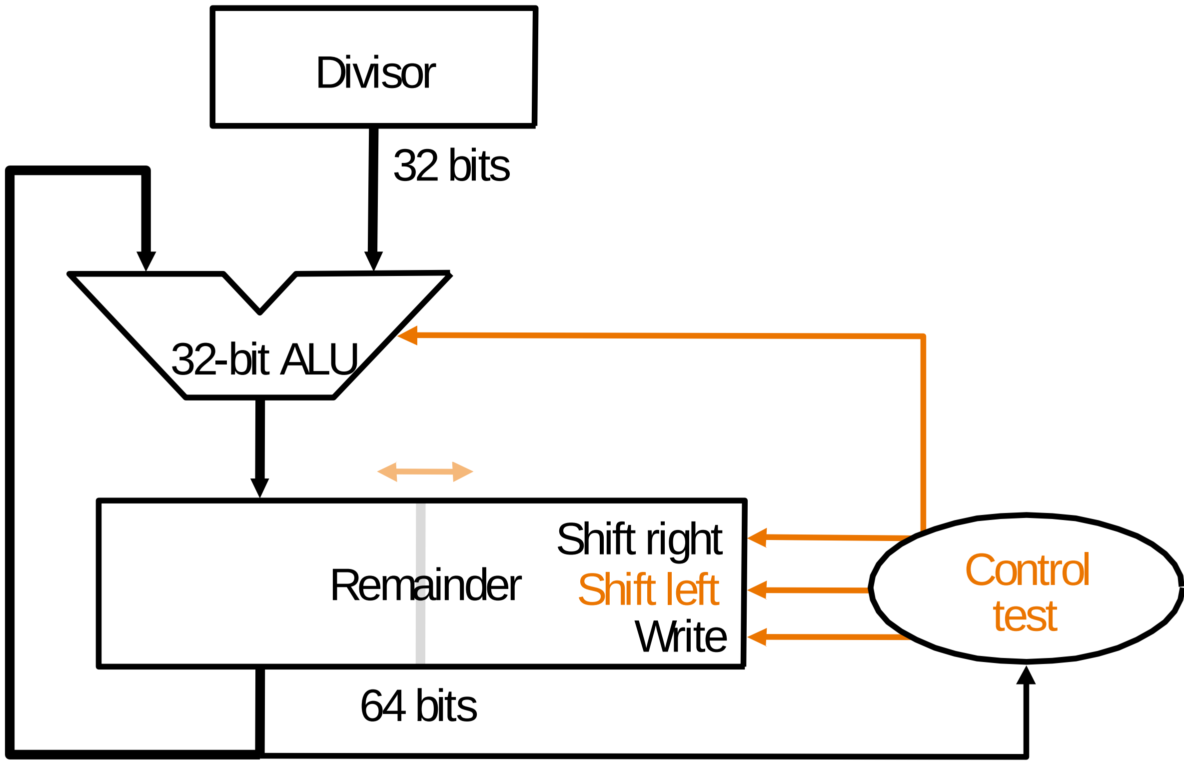
The efficiency of Booth’s algorithm is dependent on the multiplier. A long series of 1’s or 0’s can be dealt with very efficiently. But a numerical pattern like 0101 will not be very efficient. However, Booth’s algorithm works with both positive and negative numbers so it is the preferred algorithm for multiplication.

Review: Binary Division

Unlike multiplication, there is only one division algorithm. It uses the same hardware as multiplication, which means we do not need any additional hardware we just have to use the existing hardware a bit differently.

Even though we are using the same hardware, it might be helpful to relabel the pieces so we can see how they are used in the division algorithm.

The multiplicand register is repurposed as the divisor register. The product register is now labeled the remainder register.



Review: Binary Division

We initialize the registers by placing the dividend in the lower half of the remainder register and then shifting it to the left once.

Then, in each iteration, we subtract the divisor from the upper half of the remainder register and place the result in the upper half of the remainder register. We then need to check the sign of this result.

If it is negative we need to restore the previous value. The hardware can do this by adding back the same value we just subtracted (the divisor). We then shift the remainder to the left and shift in a 0.

If it positive, we shift the remainder to the left and shift in a 1.

If the divisor register is a 32-bit register we complete 32 iterations. After all the iterations are complete, we need to shift just the upper half of the remainder register to the right one bit.

Example: Binary Division

Solution 1:

Iter.	Step	Remainder	Divisor	Action

Given:

Determine A/B if
A = $(0110\ 1111)_2$ and
B = $(0001\ 1101)_2$.

Partial Credit 1:

We want to represent both the hardware we're using and the steps we're taking. Since we have 8 bit numbers, we should assume we will have an 8-bit divisor register and a 16-bit remainder register. The number of iterations is equal to the number of bits in the divisor register.

Example: Binary Division

Solution 1:

Iter.	Step	Remainder	Divisor	Action
0	0	0000 0000 1101 1110	0001 1101	Initialize

Given:

Determine A/B if
A = $(0110\ 1111)_2$ and
B = $(0001\ 1101)_2$.

Partial Credit 1:

We place A in the lower half of the remainder register and then shift the remainder register to the left 1 bit. This is a logical shift: we shift in a 0.

B is the divisor. Since we will be subtracting B, we should also calculate –B:

1110 0011

Example: Binary Division

Solution 2:

Iter.	Step	Remainder	Divisor	Action
0	0	0000 0000 1101 1110	0001 1101	Initialize
1	1	1110 0011 1101 1110	0001 1101	Subtract
1	2b	0000 0001 1011 1100	0001 1101	Shift 0

Given:

Determine A/B if
A = $(0110\ 1111)_2$ and
B = $(0001\ 1101)_2$.

Partial Credit 2:

The first step of the iteration is to subtract B from the left half of the remainder register:

1110 0011

Since the result is negative, we need to restore the previous value and shift left, place a 0 into the least significant position.

Example: Binary Division

Solution 3:

Iter.	Step	Remainder	Divisor	Action
0	0	0000 0000 1101 1110	0001 1101	Initialize
1	1	1110 0011 1101 1110	0001 1101	Subtract
1	2b	0000 0001 1011 1100	0001 1101	Shift 0
2	1	1110 0100 1011 1100	0001 1101	Subtract
2	2b	0000 0011 0111 1000	0001 1101	Shift 0

Given:

Determine A/B if
$A = (0110\ 1111)_2$ and
$B = (0001\ 1101)_2$.

Partial Credit 3:

The first step of the iteration is to subtract B from the left half of the remainder register:

0000 0001
<u>1110 0011</u>
1110 0100

Since the result is negative, we need to restore the previous value and shift left, place a 0 into the least significant position.

Example: Binary Division

Solution 4:

Iter.	Step	Remainder	Divisor	Action
0	0	0000 0000 1101 1110	0001 1101	Initialize
1	1	1110 0011 1101 1110	0001 1101	Subtract
1	2b	0000 0001 1011 1100	0001 1101	Shift 0
2	1	1110 0100 1011 1100	0001 1101	Subtract
2	2b	0000 0011 0111 1000	0001 1101	Shift 0
3	1	1110 0110 0111 1000	0001 1101	Subtract
3	2b	0000 0110 1111 0000	0001 1101	Shift 0

Given:

Determine A/B if
A = $(0110\ 1111)_2$ and
B = $(0001\ 1101)_2$.

Partial Credit 4:

The first step of the iteration is to subtract B from the left half of the remainder register:

0000 0011
<u>1110 0011</u>
1110 0110

Since the result is negative, we need to restore the previous value and shift left, place a 0 into the least significant position.

Example: Binary Division

Given:

Determine A/B if
A = $(0110\ 1111)_2$ and
B = $(0001\ 1101)_2$.

Partial Credit 5:

The first step of the iteration is to subtract B from the left half of the remainder register:

0000 0110
<u>1110 0011</u>
1110 1001

Since the result is negative, we need to restore the previous value and shift left, place a 0 into the least significant position.

Solution 5:

Iter.	Step	Remainder	Divisor	Action
0	0	0000 0000 1101 1110	0001 1101	Initialize
1	1	1110 0011 1101 1110	0001 1101	Subtract
1	2b	0000 0001 1011 1100	0001 1101	Shift 0
2	1	1110 0100 1011 1100	0001 1101	Subtract
2	2b	0000 0011 0111 1000	0001 1101	Shift 0
3	1	1110 0110 0111 1000	0001 1101	Subtract
3	2b	0000 0110 1111 0000	0001 1101	Shift 0
4	1	1110 1001 1111 0000	0001 1101	Subtract
4	2b	0000 1101 1110 0000	0001 1101	Shift 0

Example: Binary Division

Solution 6:

Iter.	Step	Remainder	Divisor	Action
0	0	0000 0000 1101 1110	0001 1101	Initialize
1	1	1110 0011 1101 1110	0001 1101	Subtract
1	2b	0000 0001 1011 1100	0001 1101	Shift 0
2	1	1110 0100 1011 1100	0001 1101	Subtract
2	2b	0000 0011 0111 1000	0001 1101	Shift 0
3	1	1110 0110 0111 1000	0001 1101	Subtract
3	2b	0000 0110 1111 0000	0001 1101	Shift 0
4	1	1110 1001 1111 0000	0001 1101	Subtract
4	2b	0000 1101 1110 0000	0001 1101	Shift 0
5	1	1111 0000 1110 0000	0001 1101	Subtract
5	2b	0001 1011 1100 0000	0001 1101	Shift 0

Given:

Determine A/B if
$A = (0110\ 1111)_2$ and
$B = (0001\ 1101)_2$.

Partial Credit 6:

The first step of the iteration is to subtract B from the left half of the remainder register:

0000 1101
<u>1110 0011</u>
1111 0000

Since the result is negative, we need to restore the previous value and shift left, place a 0 into the least significant position.

Example: Binary Division

Given:

Determine A/B if
$A = (0110\ 1111)_2$ and
$B = (0001\ 1101)_2$.

Partial Credit 7:

The first step of the iteration is to subtract B from the left half of the remainder register:

$$\begin{array}{l} 0001\ 1011 \\ \underline{1110\ 0011} \\ 1111\ 1110 \end{array}$$

Since the result is negative, we need to restore the previous value and shift left, place a 0 into the least significant position.

Solution 7:

Iter.	Step	Remainder	Divisor	Action
0	0	0000 0000 1101 1110	0001 1101	Initialize
1	1	1110 0011 1101 1110	0001 1101	Subtract
1	2b	0000 0001 1011 1100	0001 1101	Shift 0
2	1	1110 0100 1011 1100	0001 1101	Subtract
2	2b	0000 0011 0111 1000	0001 1101	Shift 0
3	1	1110 0110 0111 1000	0001 1101	Subtract
3	2b	0000 0110 1111 0000	0001 1101	Shift 0
4	1	1110 1001 1111 0000	0001 1101	Subtract
4	2b	0000 1101 1110 0000	0001 1101	Shift 0
5	1	1111 0000 1110 0000	0001 1101	Subtract
5	2b	0001 1011 1100 0000	0001 1101	Shift 0
6	1	1111 1110 1100 0000	0001 1101	Subtract
6	2b	0011 0111 1000 0000	0001 1101	Shift 0

Example: Binary Division

Solution 8:

Iter.	Step	Remainder	Divisor	Action
0	0	0000 0000 1101 1110	0001 1101	Initialize
1	1	1110 0011 1101 1110	0001 1101	Subtract
1	2b	0000 0001 1011 1100	0001 1101	Shift 0
2	1	1110 0100 1011 1100	0001 1101	Subtract
2	2b	0000 0011 0111 1000	0001 1101	Shift 0
3	1	1110 0110 0111 1000	0001 1101	Subtract
3	2b	0000 0110 1111 0000	0001 1101	Shift 0
4	1	1110 1001 1111 0000	0001 1101	Subtract
4	2b	0000 1101 1110 0000	0001 1101	Shift 0
5	1	1111 0000 1110 0000	0001 1101	Subtract
5	2b	0001 1011 1100 0000	0001 1101	Shift 0
6	1	1111 1110 1100 0000	0001 1101	Subtract
6	2b	0011 0111 1000 0000	0001 1101	Shift 0
7	1	0001 1010 1000 0000	0001 1101	Subtract
7	2a	0011 0101 0000 0001	0001 1101	Shift 1

Given: Determine A/B if
$A = (0110\ 1111)_2$ and
$B = (0001\ 1101)_2$.

Partial Credit 8: The first step of the iteration is to subtract B from the left half of the remainder register:

0011 0111
<u>1110 0011</u>
0001 1010

Since the result is positive, we leave the new upper half of the remainder register as it is and shift left, placing a 1 into the least significant position.

Example: Binary Division

Solution 9:

Given:

Determine A/B if
A = $(0110\ 1111)_2$ and
B = $(0001\ 1101)_2$.

Partial Credit 9:

The first step of the iteration is to subtract B from the left half of the remainder register:

$$\begin{array}{r} 0011\ 0101 \\ \underline{1110\ 0011} \\ 0001\ 1000 \end{array}$$

Since the result is positive, we leave the new upper half of the remainder register as it is and shift left, placing a 1 into the least significant position.

Iter.	Step	Remainder	Divisor	Action
0	0	0000 0000 1101 1110	0001 1101	Initialize
1	1	1110 0011 1101 1110	0001 1101	Subtract
1	2b	0000 0001 1011 1100	0001 1101	Shift 0
2	1	1110 0100 1011 1100	0001 1101	Subtract
2	2b	0000 0011 0111 1000	0001 1101	Shift 0
3	1	1110 0110 0111 1000	0001 1101	Subtract
3	2b	0000 0110 1111 0000	0001 1101	Shift 0
4	1	1110 1001 1111 0000	0001 1101	Subtract
4	2b	0000 1101 1110 0000	0001 1101	Shift 0
5	1	1111 0000 1110 0000	0001 1101	Subtract
5	2b	0001 1011 1100 0000	0001 1101	Shift 0
6	1	1111 1110 1100 0000	0001 1101	Subtract
6	2b	0011 0111 1000 0000	0001 1101	Shift 0
7	1	0001 1010 1000 0000	0001 1101	Subtract
7	2a	0011 0101 0000 0001	0001 1101	Shift 1
8	1	0001 1000 0000 0001	0001 1101	Subtract
8	2a	0011 0000 0000 0011	0001 1101	Shift 1

Example: Binary Division

Solution 10:

Iter.	Step	Remainder	Divisor	Action
0	0	0000 0000 1101 1110	0001 1101	Initialize
1	1	1110 0011 1101 1110	0001 1101	Subtract
1	2b	0000 0001 1011 1100	0001 1101	Shift 0
2	1	1110 0100 1011 1100	0001 1101	Subtract
2	2b	0000 0011 0111 1000	0001 1101	Shift 0
3	1	1110 0110 0111 1000	0001 1101	Subtract
3	2b	0000 0110 1111 0000	0001 1101	Shift 0
4	1	1110 1001 1111 0000	0001 1101	Subtract
4	2b	0000 1101 1110 0000	0001 1101	Shift 0
5	1	1111 0000 1110 0000	0001 1101	Subtract
5	2b	0001 1011 1100 0000	0001 1101	Shift 0
6	1	1111 1110 1100 0000	0001 1101	Subtract
6	2b	0011 0111 1000 0000	0001 1101	Shift 0
7	1	0001 1010 1000 0000	0001 1101	Subtract
7	2a	0011 0101 0000 0001	0001 1101	Shift 1
8	1	0001 1000 0000 0001	0001 1101	Subtract
8	2a	0011 0000 0000 0011	0001 1101	Shift 1

Given: Determine A/B if
$A = (0110\ 1111)_2$ and
$B = (0001\ 1101)_2$.

Partial Credit 10: Once all the iterations are complete, we need to take the upper half and shift to the right 1 bit: 0011 0000 >> 1

This is the remainder.

The lower half is the quotient.

$A/B = (0000\ 0011)_2$;
remainder $(0001\ 1000)_2$

Review: Real Numbers

In addition to integers, we have what is mathematically referred to as “real” numbers. Real numbers include: whole numbers, fractional numbers, and irrational numbers. They do not include imaginary numbers.

In programming these are called “float” values. We represent them in scientific notation and the decimal point or the binary point “floats” or changes position as we normalize the value.

Review: Real Numbers

We use the IEEE 754 standard for representing floats. We can use single (32bits) or double (64-bits) precision.

- Float values are represented in binary scientific notation — 8.5 = 1000.1
- These values are then normalized — $1.0001 * 2^3$
- A sign bit is determined — Sign = 0
- We calculate a biased exponent based on the level of precision — Exponent = 3+127 or 3+1023
- The 1 in front of the binary point is not stored — Mantissa = 0001
- The remaining bits will be zero, as trailing zeros will not affect the value

- Single precision: 8.5 = 0 10000010 000 1000 0000 0000 0000 0000
- Double precision: 8.5 = 0 10000000010 0001 0000 0000 0000 0000 0000
  0000 0000 0000 0000 0000 0000 0000

Example: Real Numbers

<u>Given</u>: Let A = 64.75. What is the representation of A in single precision IEEE-754 format?

<u>Partial Credit 1</u>: Our first step is to convert 64.75 to binary. Converting the whole number portion of the number is the same as converting an unsigned binary integer.

<u>Solution 1</u>:

64 = 2 * 32 + 0
32 = 2 * 16 + 0
16 = 2 * 8 + 0
8 = 2 * 4 + 0
4 = 2 * 2 + 0
2 = 2 * 1 + 0
1 = 2 * 0 + 1

Example: Real Numbers

Given: Let A = 64.75. What is the representation of A in single precision IEEE-754 format?

Partial Credit 2: Converting the fractional portion of the number uses a complementary process. Instead of dividing by two, we multiply by two. We need to multiply the fractional portion of our result until we reach an infinitely repeating pattern.

Solution 2:

.75 * 2 = 1.5
.5 * 2 = 1.0
.0 * 2 = 0.0 <- we can continue to multiply 0 by 2, but we will always get 0

The whole number portion of our results form the bits of the initial mantissa from top to bottom: .110

The rest of the mantissa is filled in with zeros.

Example: Real Numbers

Given: Let A = 64.75. What is the representation of A in single precision IEEE-754 format?

Partial Credit 3: Now we need to represent 64.75 in binary scientific notation.

Solution 3: We have calculated both portions so we simply place them together with a binary point between them:

$1000000.11 * 2^0$

Example: Real Numbers

Given: Let A = 64.75. What is the representation of A in single precision IEEE-754 format?

Partial Credit 4: Our next step is to normalize the value.

Solution 4: $1000000.11 * 2^0 = 1.00000011 * 2^6$

Example: Real Numbers

Given: Let A = 64.75. What is the representation of A in single precision IEEE-754 format?

Partial Credit 5: Now we can start filling the fields of A in single precision format. This format has 32 bits separated into three fields: 1-bit sign, 8-bit exponent, 23-bit mantissa.

Solution 5: The original number is positive, so the sign bit will be 0.

Example: Real Numbers

Given: Let A = 64.75. What is the representation of A in single precision IEEE-754 format?

Partial Credit 6: In the IEEE-754 standard the exponent we store is actually higher than the actual exponent. This allows us to store negative exponents without changing our representation. The difference between the actual exponent and the stored exponent is called the bias. In single precision, the bias is 127.

Solution 6: The original exponent is 6.
The stored exponent should be 6 + 127 = 133
We store this in 8 bits: 10000101

Example: Real Numbers

Given: Let A = 64.75. What is the representation of A in single precision IEEE-754 format?

Partial Credit 7: The mantissa field is 23 bits wide. However, we only store the values on the right side of the binary point.

1.00000011

Solution 7: Mantissa: 000 0001 1000 0000 0000 0000

Example: Real Numbers

<u>Given</u>: Let A = 64.75. What is the representation of A in single precision IEEE-754 format?

<u>Partial Credit 8</u>: Our final step is to put the three fields together to form a full 32 bit number.

<u>Solution 8</u>: Sign: 0
Exponent: 10000101
Mantissa: 000 0001 1000 0000 0000 0000

A = 0 10000101 000 0001 1000 0000 0000 0000

Example: Real Numbers

<u>Given</u>: Let A = 1100 0010 0000 1010 0000 0000 0000 0000. If A is a single precision float value written in the IEEE 754 standard, what decimal value does it represent?

<u>Partial Credit 1</u>: We know that single precision has three fields so the first step is to identify those three fields in the bitstring we are given for A.

<u>Solution 1</u>:

The first bit is the sign bit.	1
The next 8 bits are the biased exponent.	10000100
The final 23 bits are the mantissa.	000 1010 0000 0000 0000 0000

Example: Real Numbers

Given: Let A = 1100 0010 0000 1010 0000 0000 0000 0000. If A is a single precision float value written in the IEEE 754 standard, what decimal value does it represent?

Partial Credit 2: Now we can translate each field into a normalized scientific binary value.

Solution 2: Since the sign bit is 1, the number is negative. The stored exponent is higher than the actual exponent, so we need to remove the bias. The actual exponent is 10000100 – 127 = 132 – 127 = 5. With the mantissa we need to add in the implied 1. that is not stored: 1.000101

Altogether A = -1.000101 * 2^5

Example: Real Numbers

<u>Given</u>: Let A = 1100 0010 0000 1010 0000 0000 0000 0000. If A is a single precision float value written in the IEEE 754 standard, what decimal value does it represent?

<u>Partial Credit 3</u>: Now that we have $A = -1.000101 * 2^5$ we can convert this to base 10. First, “denormalize” the value until the exponent is 0.

<u>Solution 3</u>: $A = -1.000101 * 2^5 = -100010.1 * 2^0$

Multiplying by 2^0 is the same as multiplying by 1, so this can be dropped.

Example: Real Numbers

Given: Let A = 1100 0010 0000 1010 0000 0000 0000 0000. If A is a single precision float value written in the IEEE 754 standard, what decimal value does it represent?

Partial Credit 4: Converting to base 10 after the number is denormalized is the same as converting an unsigned binary value to base 10.

Solution 4:

A = -100010.1

A = -1 * (1*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 0*2^0 + 1*2^{-1})

A = -1 * (32 + 2 + .5)

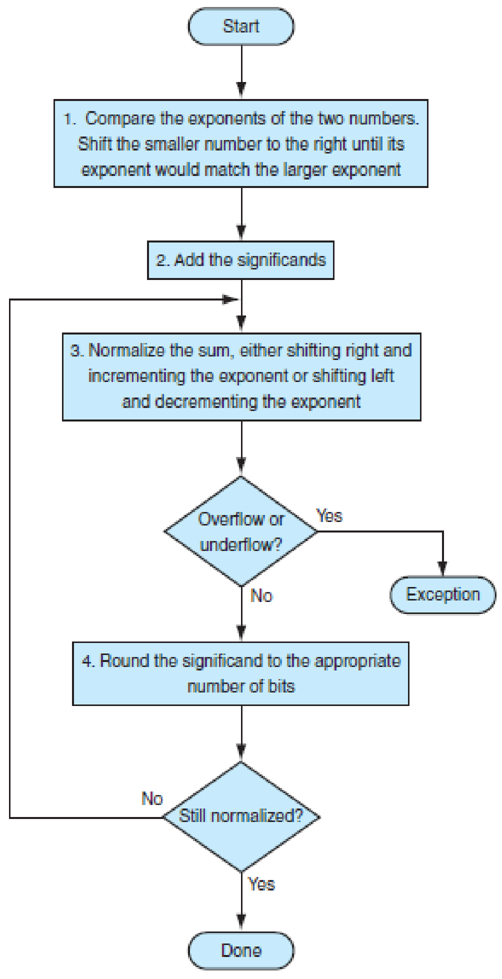
A = -34.5

Review: Floating Point Addition

Arithmetic with real number is a bit more complicated that integers. We get to use all of the same hardware, but we need to follow a different algorithm to ensure we get the correct answer.

The addition algorithm is shown in the activity diagram to the right.

Subtraction is handled with the same algorithm. Instead of adding in the second step we subtract the mantissas instead.



Example: Floating Point Addition

Given: Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X + Y and express the final answer in single precision IEEE 754-floating point representation.

Partial Credit 1: The easiest way to follow the algorithm is to first show X and Y in normalized scientific notation.

Solution1:

$X = 1.01111 * 2^{(01111111)}$

$X = 1.01111 * 2^{127}$

$Y = 1.1111 * 2^{(01110011)}$

$Y = 1.1111 * 2^{115}$

Example: Floating Point Addition

Given: Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X + Y and express the final answer in single precision IEEE 754-floating point representation.

Partial Credit 2: The first step of the algorithm is to align the binary points of both values. This is typically done by denomalizing the smaller value. In this case Y is smaller, so we will move the binary point of Y until it has the same exponent as X.

Solution2: $Y = 1.\ 1111 * 2^{115}$

$Y = 0.000\ 0000\ 0000\ 1111\ 1 * 2^{127}$

Example: Floating Point Addition

Given: Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X + Y and express the final answer in single precision IEEE 754-floating point representation.

Partial Credit 3: Now we can add X and Y together using binary addition:

X + Y =

Solution 3:

$$
\begin{array}{r}
1.\ 011\ 1100\ 0000\ 0000\ 0 * 2^{127} \\
\underline{0.\ 000\ 0000\ 0000\ 1111\ 1 * 2^{127}} \\
1.\ 011\ 1100\ 0000\ 1111\ 1 * 2^{127}
\end{array}
$$

Example: Floating Point Addition

Given: Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X + Y and express the final answer in single precision IEEE 754-floating point representation.

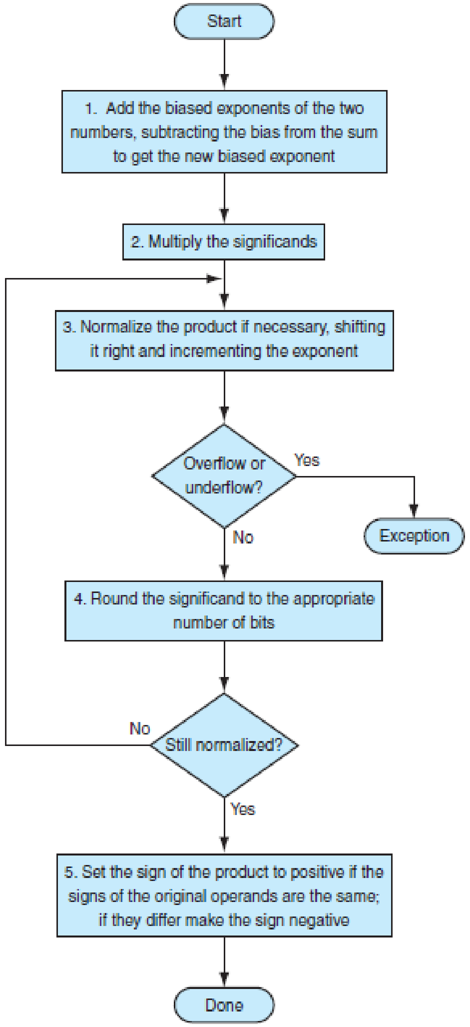
Partial Credit 4: 1. 011 1100 0000 1111 1 * 2^{127} This value is already normalized. So we just need to represent it in single precisión IEEE 754 format. The sign is positive and the exponent is the same as the original exponent for X.

Solution 4: X+Y = $(0011\ 1111\ 1011\ 1100\ 0000\ 1111\ 1000\ 0000)_2$

Review: Floating Point Multiplication

The multiplication algorithm is shown in the activity diagram to the right.

Division requires a few modifications. Instead of adding the exponents, we subtract. Instead of multiplying the mantissas we divide.



Example: Floating Point Multiplication

Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X * Y and express the final answer in single precision IEEE 754-floating point representation.

Example: Floating Point Multiplication

Given: Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X * Y and express the final answer in single precision IEEE 754-floating point representation.

Partial Credit 1: The easiest way to follow the algorithm is to first show X and Y in normalized scientific notation.

Solution1:

$X = 1.01111 * 2^{(01111111)}$
$X = 1.01111 * 2^{127}$

$Y = 1.1111 * 2^{(01110011)}$
$Y = 1.1111 * 2^{115}$

Example: Floating Point Multiplication

Given: Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X * Y and express the final answer in single precision IEEE 754-floating point representation.

Partial Credit 2: An easy method to handle decimal places in multiplication is to denormalize both values so that there are no 1's after the binary point.

Solution 2: $X = 1.01111 * 2^{127}$
$X = 101111 * 2^{122}$

$Y = 1.1111 * 2^{115}$
$Y = 11111 * 2^{111}$

Example: Floating Point Multiplication

Given: Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X * Y and express the final answer in single precision IEEE 754-floating point representation.

Partial Credit 3: The exponent of our result will be the two exponents of X and Y added together. These exponents are biased, so we need to deduct the bias from the answer. (If we had removed the bias earlier, there would be no need to subtract 127.)

Solution 3: Exponent = 122 + 111 – 127 = 106

Example: Floating Point Multiplication

Given: Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X * Y and express the final answer in single precision IEEE 754-floating point representation.

Partial Credit 4: Now we can multiply the mantissa using any multiplication process.

Solution 4:

```
        101111
      * 11111
      -------
       101111
      101111
     101111
    101111
   101111
   ----------
  10110110001
```

Example: Floating Point Multiplication

Given: Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X * Y and express the final answer in single precision IEEE 754-floating point representation.

Partial Credit 5: Our next step is to normalize the result.

Solution 5: $X * Y = 10110110001 * 2^{106}$

$X * Y = 1.0110110001 * 2^{116}$

Example: Floating Point Multiplication

Given: Suppose X = (0011 1111 1011 1100 0000 0000 0000 0000)2 and Y = (0011 1001 1111 1000 0000 0000 0000 0000)2 in single precision IEEE-754 floating point numbers. Determine X * Y and express the final answer in single precision IEEE 754-floating point representation.

Partial Credit 6: X * Y = 1.0110110001 * 2^{116}. Finally we need to represent this in single precision IEEE 754-floating point representation. The sign is positive and the exponent is 01110100.

Solution 6: X * Y = $(0011\ 1010\ 0011\ 0110\ 0010\ 0000\ 0000\ 0000)_2$