HW#7 (CSC390) **Due: 03/23/2018 by 11:00PM on Blackbaord**

SOLUTION

Q1. (Read chapter#3 (pages 178-182) of your text book before you solve the following questions)

(a) MIPS assembler generates an exception error when overflow occurs in mathematical operation of signed numbers. Write a MIPS assembly language program that will check overflows during the element by element addition between the array elements (signed numbers) of two vectors A and B and store the results in C vector. If overflow occurs the code will store $0xFFFFFFFF_{hex}$ in the corresponding memory location of the C vector, otherwise store the actual addition results.

Consider the following vectors A and B and store the results C vectors in your coding:

A = [1000000000, 2000000000, 2000000000, -1000000000, -2000000000];

B= [1000000000, -1000000000, 1000000000, -1000000000, -1000000000];

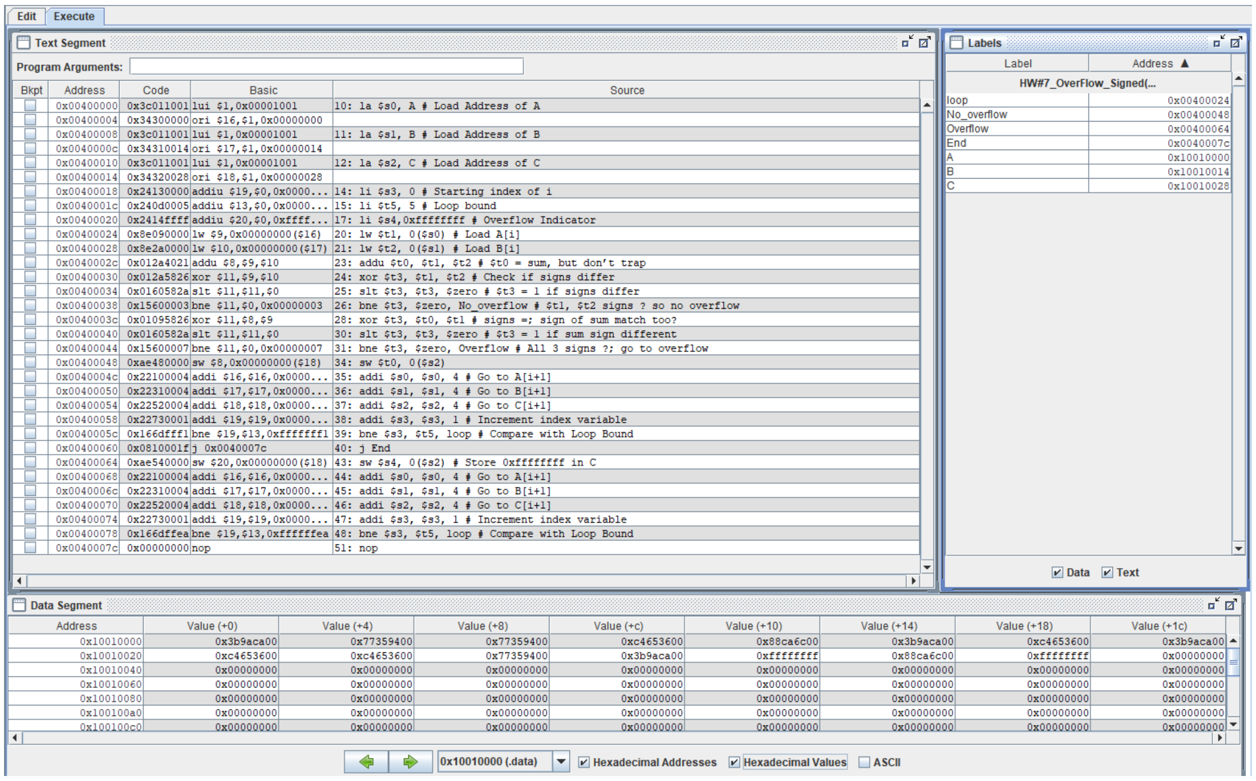
C=[. . . ..];

SOLUTION:

```
#Detecting Overflow in signed addition
# HW#7(Q#2_a)

.data
A: .word 1000000000, 2000000000, 2000000000, -1000000000, -2000000000
B: .word 1000000000, -1000000000, 1000000000, -1000000000, -1000000000
C: .space 20 #allocting spce for storing results

.text
la $s0, A # Load Address of A
la $s1, B # Load Address of B
la $s2, C # Load Address of C

li $s3, 0 # Starting index of i
li $t5, 5 # Loop bound

li $s4,0xffffffff # Overflow Indicator

loop:
lw $t1, 0($s0) # Load A[i]
lw $t2, 0($s1) # Load B[i]

addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor $t3, $t1, $t2 # Check if signs differ
slt $t3, $t3, $zero # $t3 = 1 if signs differ
bne $t3, $zero, No_overflow # $t1, $t2 signs ? so no overflow

xor $t3, $t0, $t1 # signs =; sign of sum match too?
```

```
# $t3 negative if sum sign different
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow # All 3 signs ?; go to overflow

No_overflow:
sw $t0, 0($s2)
addi $s0, $s0, 4 # Go to A[i+1]
addi $s1, $s1, 4 # Go to B[i+1]
addi $s2, $s2, 4 # Go to C[i+1]
addi $s3, $s3, 1 # Increment index variable
bne $s3, $t5, loop # Compare with Loop Bound
j End

Overflow:
sw $s4, 0($s2) # Store 0xffffffff in C
addi $s0, $s0, 4 # Go to A[i+1]
addi $s1, $s1, 4 # Go to B[i+1]
addi $s2, $s2, 4 # Go to C[i+1]
addi $s3, $s3, 1 # Increment index variable
bne $s3, $t5, loop # Compare with Loop Bound

End:
nop
```

Output Print Screen:



Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	10: la $s0, A # Load Address of A
☐	0x00400004	0x34300000	ori $16,$1,0x00000000	
☐	0x00400008	0x3c011001	lui $1,0x00001001	11: la $s1, B # Load Address of B
☐	0x0040000c	0x34310014	ori $17,$1,0x00000014	
☐	0x00400010	0x3c011001	lui $1,0x00001001	12: la $s2, C # Load Address of C
☐	0x00400014	0x34320028	ori $18,$1,0x00000028	
☐	0x00400018	0x24130000	addiu $19,$0,0x0000...	14: li $s3, 0 # Starting index of i
☐	0x0040001c	0x240d0005	addiu $13,$0,0x0000...	15: li $t5, 5 # Loop bound
☐	0x00400020	0x2414ffff	addiu $20,$0,0xffff...	17: li $s4,0xffffffff # Overflow Indicator
☐	0x00400024	0x8e090000	lw $9,0x00000000($16)	20: lw $t1, 0($s0) # Load A[i]
☐	0x00400028	0x8e2a0000	lw $10,0x00000000($17)	21: lw $t2, 0($s1) # Load B[i]
☐	0x0040002c	0x012a4021	addu $8,$9,$10	23: addu $t0, $t1, $t2 # $t0 = sum, but don't trap
☐	0x00400030	0x012a5826	xor $11,$9,$10	24: xor $t3, $t1, $t2 # Check if signs differ
☐	0x00400034	0x0160582a	slt $11,$11,$0	25: slt $t3, $t3, $zero # $t3 = 1 if signs differ
☐	0x00400038	0x15600003	bne $11,$0,0x00000003	26: bne $t3, $zero, No_overflow # $t1, $t2 signs ? so no overflow
☐	0x0040003c	0x01095826	xor $11,$8,$9	28: xor $t3, $t0, $t1 # signs =; sign of sum match too?
☐	0x00400040	0x0160582a	slt $11,$11,$0	30: slt $t3, $t3, $zero # $t3 = 1 if sum sign different
☐	0x00400044	0x15600007	bne $11,$0,0x00000007	31: bne $t3, $zero, Overflow # All 3 signs ?; go to overflow
☐	0x00400048	0xae480000	sw $8,0x00000000($18)	34: sw $t0, 0($s2)
☐	0x0040004c	0x22100004	addi $16,$16,0x0000...	35: addi $s0, $s0, 4 # Go to A[i+1]
☐	0x00400050	0x22310004	addi $17,$17,0x0000...	36: addi $s1, $s1, 4 # Go to B[i+1]
☐	0x00400054	0x22520004	addi $18,$18,0x0000...	37: addi $s2, $s2, 4 # Go to C[i+1]
☐	0x00400058	0x22730001	addi $19,$19,0x0000...	38: addi $s3, $s3, 1 # Increment index variable
☐	0x0040005c	0x166dfff1	bne $19,$13,0xfffffff1	39: bne $s3, $t5, loop # Compare with Loop Bound
☐	0x00400060	0x0810001f	j 0x0040007c	40: j End
☐	0x00400064	0xae540000	sw $20,0x00000000($18)	43: sw $s4, 0($s2) # Store 0xffffffff in C
☐	0x00400068	0x22100004	addi $16,$16,0x0000...	44: addi $s0, $s0, 4 # Go to A[i+1]
☐	0x0040006c	0x22310004	addi $17,$17,0x0000...	45: addi $s1, $s1, 4 # Go to B[i+1]
☐	0x00400070	0x22520004	addi $18,$18,0x0000...	46: addi $s2, $s2, 4 # Go to C[i+1]
☐	0x00400074	0x22730001	addi $19,$19,0x0000...	47: addi $s3, $s3, 1 # Increment index variable
☐	0x00400078	0x166dffea	bne $19,$13,0xffffffea	48: bne $s3, $t5, loop # Compare with Loop Bound
☐	0x0040007c	0x00000000	nop	51: nop

Label	Address ▲
HW#7_OverFlow_Signed(...	
loop	0x00400024
No_overflow	0x00400048
Overflow	0x00400064
End	0x0040007c
A	0x10010000
B	0x10010014
C	0x10010028

☑ Data ☑ Text

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x3b9aca00	0x77359400	0x77359400	0xc4653600	0x88ca6c00	0x3b9aca00	0xc4653600	0x3b9aca00
0x10010020	0xc4653600	0xc4653600	0x77359400	0x3b9aca00	0xffffffff	0x88ca6c00	0xffffffff	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

(b) Write a MIPS assembly language program that will check overflows during the element by element addition between the array elements (unsigned numbers) of two vectors A and B and store the results in C vector. If overflow occurs the code will store 0x00000000 in the corresponding memory location of the C vector, otherwise store the actual addition results.

Consider the following vectors A and B and store the results C vectors in your coding:

A = [1000000000, 2000000000, 2000000000, 1000000000, 2000000000];

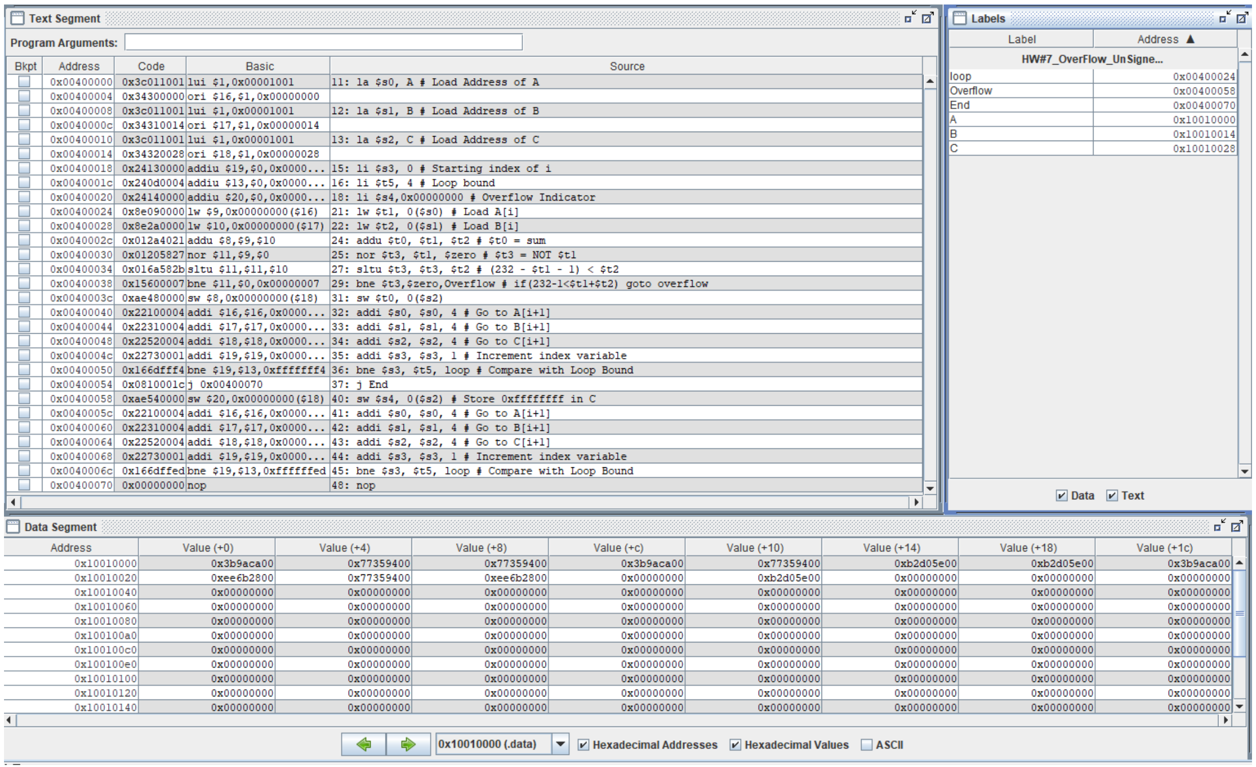
B = [3000000000, 3000000000, 1000000000, 4000000000, 2000000000];

C=[. . .   ….   ….   …   ];

SOLUTION:

```
#Detecting Overflow in unsigned addition
# HW#7_Q1(b)

.data
A: .word 1000000000, 2000000000, 2000000000, 1000000000, 2000000000
B: .word 3000000000, 3000000000, 1000000000, 4000000000, 2000000000
C: .space 20 #allocting spce for storing results


.text
la $s0, A # Load Address of A
la $s1, B # Load Address of B
la $s2, C # Load Address of C

li $s3, 0 # Starting index of i
li $t5, 4 # Loop bound

li $s4,0x00000000 # Overflow Indicator

loop:
lw $t1, 0($s0) # Load A[i]
lw $t2, 0($s1) # Load B[i]

addu $t0, $t1, $t2 # $t0 = sum
nor $t3, $t1, $zero # $t3 = NOT $t1
             # (2’s comp – 1: 232 – $t1 – 1)
sltu $t3, $t3, $t2 # (232 – $t1 – 1) < $t2
            # ? 232 – 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(232–1<$t1+$t2) goto overflow

sw $t0, 0($s2)
addi $s0, $s0, 4 # Go to A[i+1]
addi $s1, $s1, 4 # Go to B[i+1]
addi $s2, $s2, 4 # Go to C[i+1]
addi $s3, $s3, 1 # Increment index variable
bne $s3, $t5, loop # Compare with Loop Bound
j End

Overflow:
sw $s4, 0($s2) # Store 0xffffffff in C
addi $s0, $s0, 4 # Go to A[i+1]
addi $s1, $s1, 4 # Go to B[i+1]
addi $s2, $s2, 4 # Go to C[i+1]
addi $s3, $s3, 1 # Increment index variable
bne $s3, $t5, loop # Compare with Loop Bound

End:
nop
```

Output Print Screen



Text Segment

Program Arguments:

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x3c011001	lui $1,0x00001001	11: la $s0, A # Load Address of A
☐	0x00400004	0x34300000	ori $16,$1,0x00000000	
☐	0x00400008	0x3c011001	lui $1,0x00001001	12: la $s1, B # Load Address of B
☐	0x0040000c	0x34310014	ori $17,$1,0x00000014	
☐	0x00400010	0x3c011001	lui $1,0x00001001	13: la $s2, C # Load Address of C
☐	0x00400014	0x34320028	ori $18,$1,0x00000028	
☐	0x00400018	0x24130000	addiu $19,$0,0x0000...	15: li $s3, 0 # Starting index of i
☐	0x0040001c	0x240d0004	addiu $13,$0,0x0000...	16: li $t5, 4 # Loop bound
☐	0x00400020	0x24140000	addiu $20,$0,0x0000...	18: li $s4,0x00000000 # Overflow Indicator
☐	0x00400024	0x8e090000	lw $9,0x00000000($16)	21: lw $t1, 0($s0) # Load A[i]
☐	0x00400028	0x8e2a0000	lw $10,0x00000000($17)	22: lw $t2, 0($s1) # Load B[i]
☐	0x0040002c	0x012a4021	addu $8,$9,$10	24: addu $t0, $t1, $t2 # $t0 = sum
☐	0x00400030	0x01205827	nor $11,$9,$0	25: nor $t3, $t1, $zero # $t3 = NOT $t1
☐	0x00400034	0x016a582b	sltu $11,$11,$10	27: sltu $t3, $t3, $t2 # (232 - $t1 - 1) < $t2
☐	0x00400038	0x15600007	bne $11,$0,0x00000007	29: bne $t3,$zero,Overflow # if(232-1<$t1+$t2) goto overflow
☐	0x0040003c	0xae480000	sw $8,0x00000000($18)	31: sw $t0, 0($s2)
☐	0x00400040	0x22100004	addi $16,$16,0x0000...	32: addi $s0, $s0, 4 # Go to A[i+1]
☐	0x00400044	0x22310004	addi $17,$17,0x0000...	33: addi $s1, $s1, 4 # Go to B[i+1]
☐	0x00400048	0x22520004	addi $18,$18,0x0000...	34: addi $s2, $s2, 4 # Go to C[i+1]
☐	0x0040004c	0x22730001	addi $19,$19,0x0000...	35: addi $s3, $s3, 1 # Increment index variable
☐	0x00400050	0x166dfff4	bne $19,$13,0xfffffff4	36: bne $s3, $t5, loop # Compare with Loop Bound
☐	0x00400054	0x0810001c	j 0x00400070	37: j End
☐	0x00400058	0xae540000	sw $20,0x00000000($18)	40: sw $s4, 0($s2) # Store 0xffffffff in C
☐	0x0040005c	0x22100004	addi $16,$16,0x0000...	41: addi $s0, $s0, 4 # Go to A[i+1]
☐	0x00400060	0x22310004	addi $17,$17,0x0000...	42: addi $s1, $s1, 4 # Go to B[i+1]
☐	0x00400064	0x22520004	addi $18,$18,0x0000...	43: addi $s2, $s2, 4 # Go to C[i+1]
☐	0x00400068	0x22730001	addi $19,$19,0x0000...	44: addi $s3, $s3, 1 # Increment index variable
☐	0x0040006c	0x166dffed	bne $19,$13,0xffffffed	45: bne $s3, $t5, loop # Compare with Loop Bound
☐	0x00400070	0x00000000	nop	48: nop

Labels

Label	Address ▲
HW#7_OverFlow_UnSigne...	
loop	0x00400024
Overflow	0x00400058
End	0x00400070
A	0x10010000
B	0x10010014
C	0x10010028

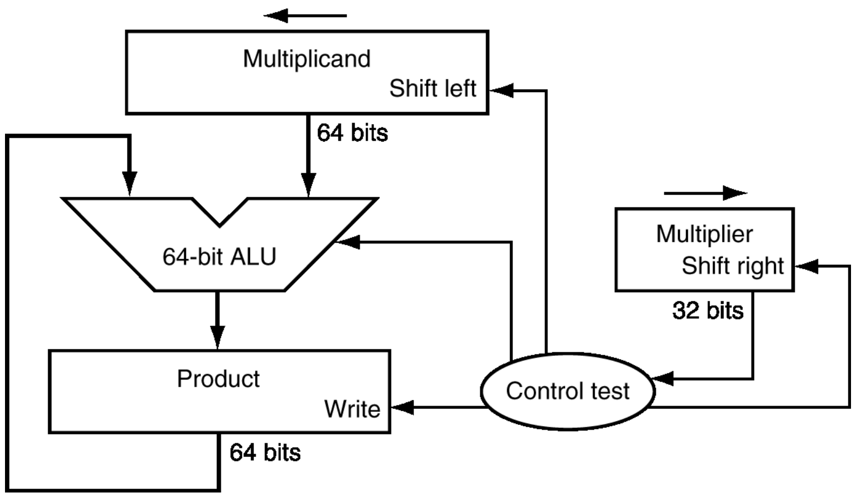
☑ Data ☑ Text

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x3b9aca00	0x77359400	0x77359400	0x3b9aca00	0x77359400	0xb2d05e00	0xb2d05e00	0x3b9aca00
0x10010020	0xee6b2800	0x77359400	0xee6b2800	0x00000000	0xb2d05e00	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

Q2. The following Circuit performs the multiplication of two 32-bit unsigned binary numbers and produces a 64-bit result. Figure 1 shows the three steps multiplication algorithm that the circuit performs to produce the result. Suppose, you are assigned to design a 5-bit multiplier circuit which will perform the multiplication of the two 5-bit unsigned numbers A=$(11011)_2$ and B=$(10011)_2$ and produce a 10-bit result. Draw the necessary hardware and show the results produced in each iteration of your algorithm. Hints: see figure 3.6 (page 187) of our text book.



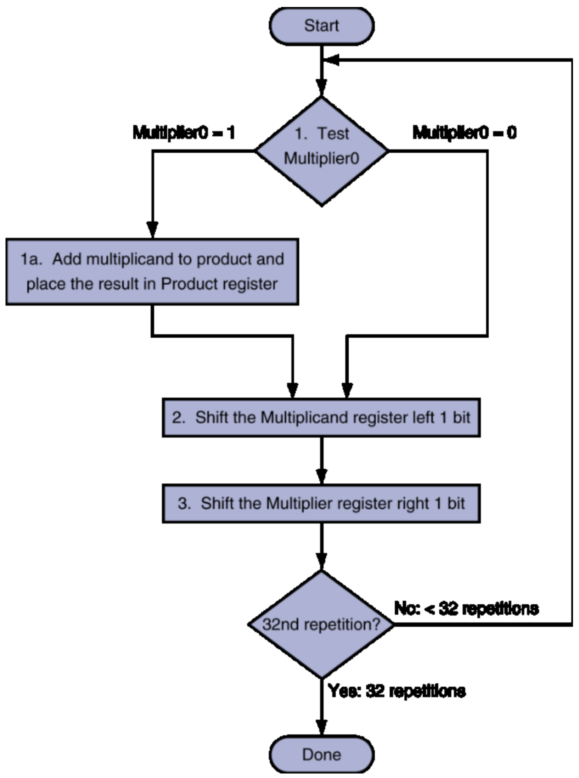
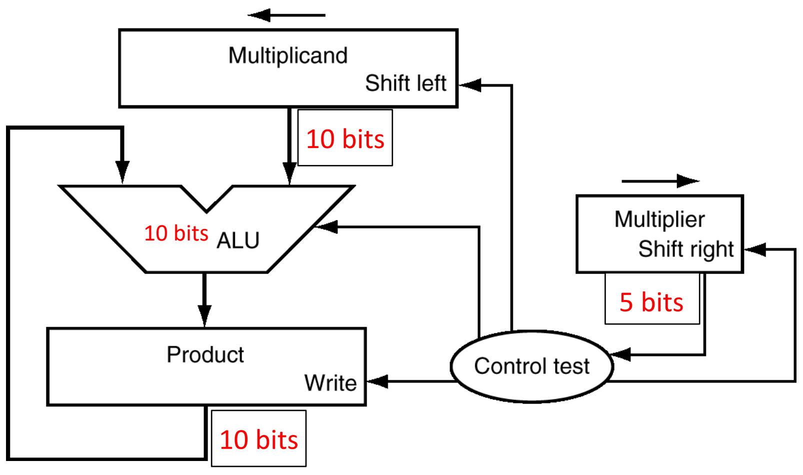
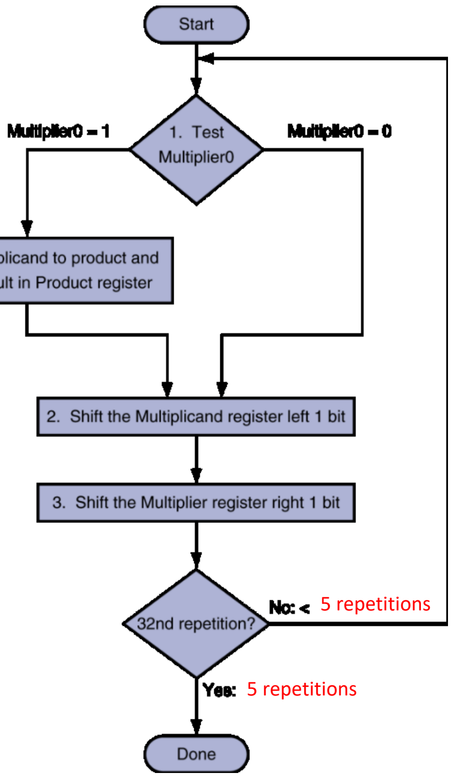

Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
No: < 32 repetitions
32nd repetition?
Yes: 32 repetitions
Done

Figure 1

SOLUTION:





Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 5 repetitions
Yes: 5 repetitions
Done

Figure 2

Iteration	Step	Multiplier	Multiplicand	Product
0	**Initial Value**	**10011**	**0000011011**	**0000000000**
1	1a: 1=> prod=prod+Mcand	**10011**	**0000011011**	**0000011011**
	2: Shift left Multiplicand	**10011**	**0000110110**	**0000011011**
	3: Shift right Multiplier	**01001**	**0000110110**	**0000011011**
2	1a: 1=> prod=prod+Mcand	**01001**	**0000110110**	**0001010001**
	2: Shift left Multiplicand	**01001**	**0001101100**	**0001010001**
	3: Shift right Multiplier	**00100**	**0001101100**	**0001010001**
3	1: 0 => no operation	**00100**	**0001101100**	**0001010001**
	2: Shift left Multiplicand	**00100**	**0011011000**	**0001010001**
	3: Shift right Multiplier	**00010**	**0011011000**	**0001010001**
4	1: 0 => no operation	**00010**	**0011011000**	**0001010001**
	2: Shift left Multiplicand	**00010**	**0110110000**	**0001010001**
	3: Shift right Multiplier	**00001**	**0110110000**	**0001010001**
5	1a: 1=> prod=prod+Mcand	**00001**	**0110110000**	**1000000001**
	2: Shift left Multiplicand	**00001**	**1101100000**	**1000000001**
	3: Shift right Multiplier	**00000**	**1101100000**	$\boxed{\textbf{1000000001}}$

11011 → 27

10011 →19

27x19 = 513 → 1000000001

Q3 (a). The optimized version of the hardware and its corresponding algorithm is shown in figure 2. Note that the right half of the product register is now initialized with the multiplier.

Suppose, you are assigned to design a 5-bit multiplier circuit which will perform the multiplication of the two 5-bit signed numbers A=$(11011)_2$ and B=$(01011)_2$ and produce a 10-bit result. Draw the necessary hardware and show the results produced in each iteration of your algorithm. Hints: see slide 13 of the Chapter_03.ppt posted on the Blackboard.

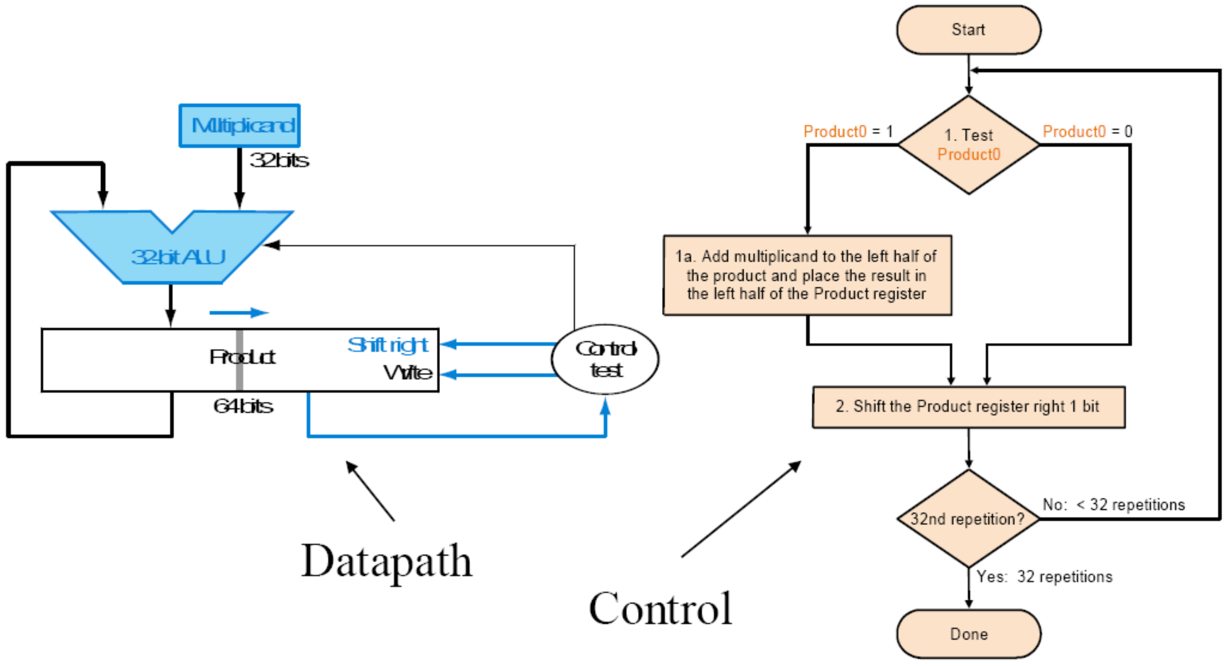


Figure 3

SOLUTION: Next Page

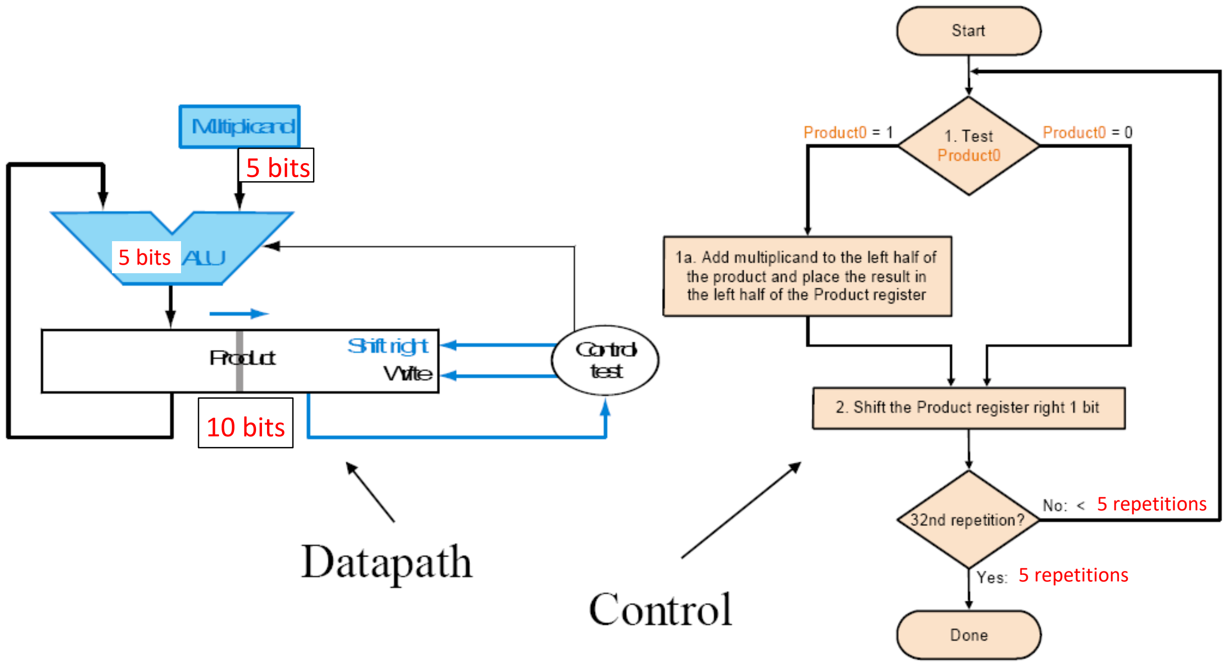


Figure 4

Steps	Multiplicand	Product
Initial Value	11011	0000001011
1a	11011	1101101011
2 , 3	11011	1110110101
1a	11011	1100010101
2 , 3	11011	1110001010
1	11011	1110001010
2, 3	11011	1111000101
1a	11011	1100100101
2 , 3	11011	1110010010
1	11011	1110010010
2, 3	11011	**1111001001**

11011 → -5 ; 01011 → 11 ; -5x11 = -55 → 1111001001

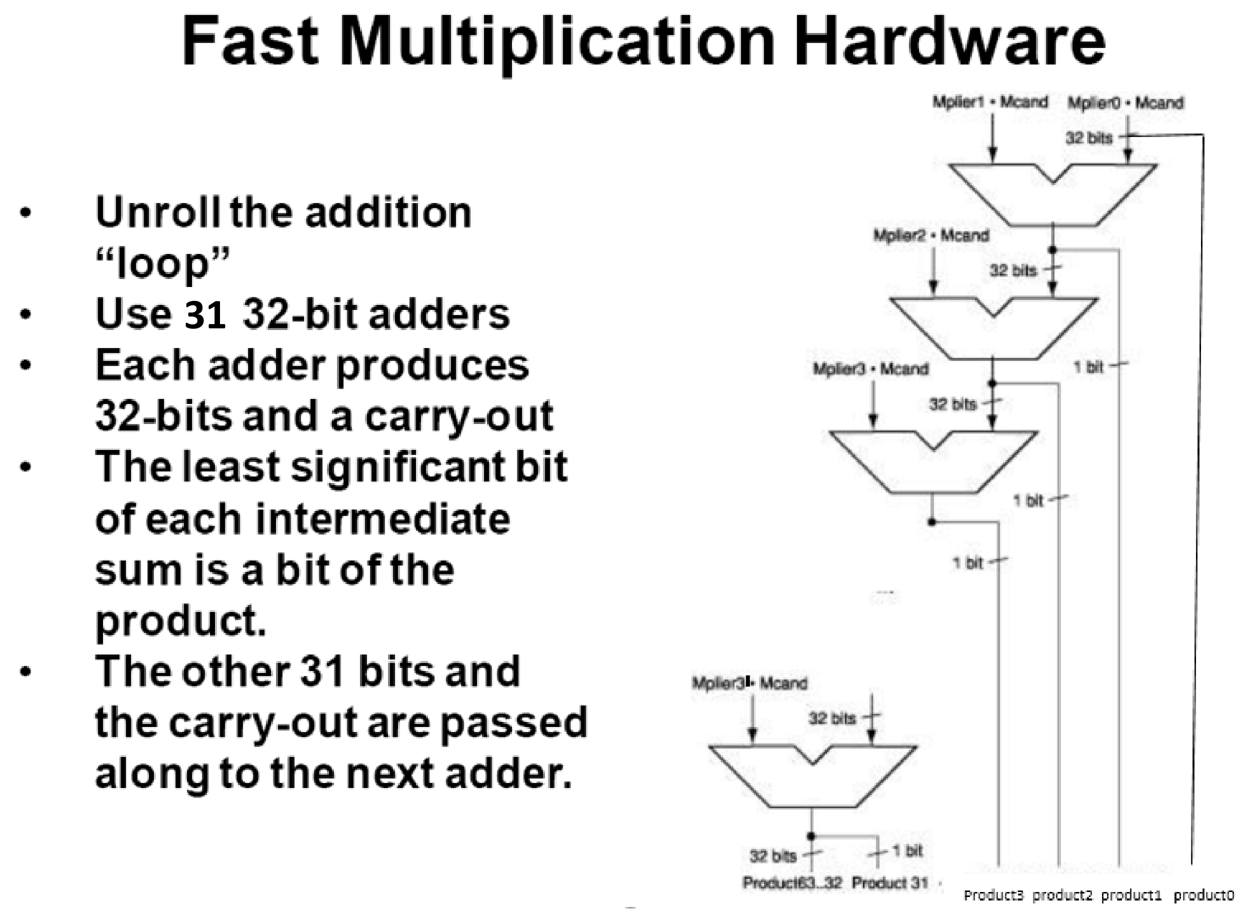
Q3 (b). Verify that the above can also be used to perform the multiplication of the two 5-bit **unsigned** numbers A=$(11011)_2$ and B=$(01011)_2$ and produce a 10-bit result. Show the results produced in each iteration of your algorithm

SOLUTION:

Step	Multiplican	Product
	11011	00000 01011
1a	11011	11011 01011
2,3		01101 10101
1a	11011	01000 10101
2,3		10100 01010
1a	11011	10100 01010
2,3		01010 00101
1a	11011	00101 00101
2,3		10010 10010
1a	11011	10010 10010
2,3		01001 01001

The result is 0100101001_2

Q4. To improve the execution speed of the above multiplier circuits, a Fast Multiplication Hardware, as shown in figure 3, is designed using (n-1) adder circuits, which perform the multiplication of two n-bit numbers and produces the 2n-bit result. Suppose, you are assigned to design a 5-bit Fast Multiplication Hardware circuit which will perform the multiplication of the two 5-bit unsigned numbers A=$(11011)_2$ and B=$(10011)_2$ and produce a 10-bit result. Draw the necessary hardware and show the results produced in step of the multiplication process.



Fast Multiplication Hardware

- Unroll the addition "loop"
- Use 31 32-bit adders
- Each adder produces 32-bits and a carry-out
- The least significant bit of each intermediate sum is a bit of the product.
- The other 31 bits and the carry-out are passed along to the next adder.

Figure 5

Solution: $\boxed{\text{\# Q4.}}$

$A = (11011)_2 \rightarrow (27)_{10}$

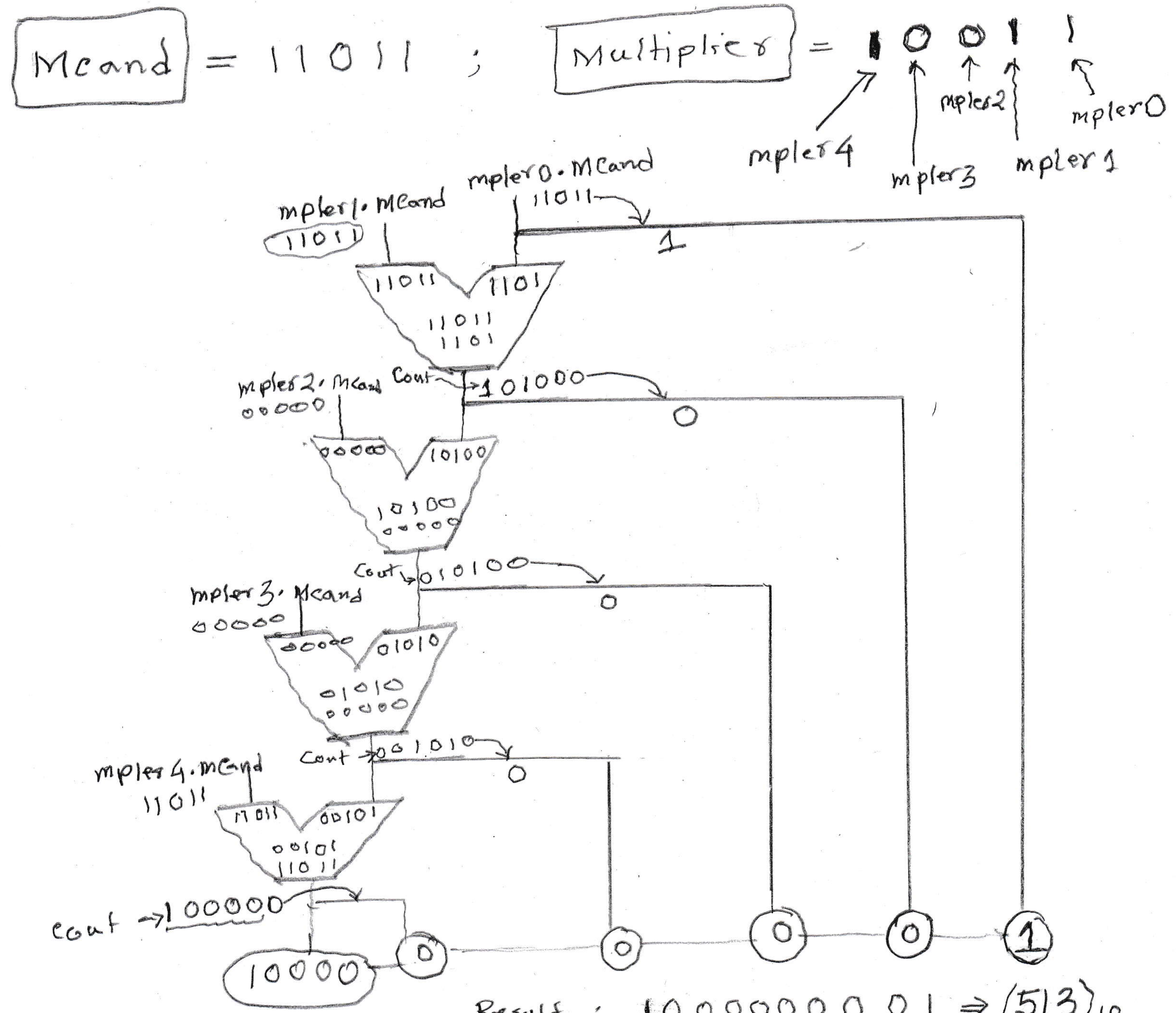
$B = (10011)_2 \rightarrow (19)_{10}$

$A \times B = (513)_{10}$

$(513)_{10} \Rightarrow (1000000001)_2$

5-bit multiplication, so we need four 5-bit adders

$\boxed{\text{Mcand}} = 11011$; $\boxed{\text{Multiplier}} = 10011$



Result : $1000000001 \Rightarrow (513)_{10}$