CS-47 Project 1

*Programming a calculator by assembley language

Yulong Ran

yulong.ran@sjsu.edu

San Jose State University

***Abstract*—The report consists of diagrams, code snippet, screen shots of testing results and explanation of four logic operations: Addition, Subtraction, Multiplication and Division.**

I. Introduction

The objective of this project is to simulate four basic calculator operations: Addition, Subtraction, Multiplication, and Division with assembly language. The calculator consists of both arithmetic operations and logic operations. The goal of this project is to enhance understanding of how computers perform mathematics through the implementation of logic operations.

II. Requirement

A. Normal operations

- The au_normal procedure uses normal math operations of MIPS to compute the result such as add, sub, mul and div.
- Input: The au_normal takes three arguments as $a0 (first operand); $a1 (second operand); $a2 (operation code ‘+’, ‘-‘, ‘*’, ‘/’ – ASCII code).
- Return: The au_normal returns result in $v0 and $v1; for multiplication $v1 will contain HI, for division $v1 will contain remainder.

B. Logic operations

- The au_logical procedure only uses MIP logic operations; the implementation follows the digital algorithm in the hardware.
- Input: The au_logical takes three arguments as $a0 (first operand); $a1 (second operand); $a2 (operation code ‘+’, ‘-‘, ‘*’, ‘/’ – ASCII code).
- Return: The au_logical returns result in $v0 and $v1; for multiplication $v1 will contain HI, for division $v1 will contain remainder.

III. Design and Implementation

A. Normal operations

```
#include "./cs47_proj_macro.asm"
.text
.globl au_normal
# TBD: Complete your project procedures
# Needed skeleton is given
#####################################################################
# Implement au_normal
# Argument:
#       $a0: First number
#       $a1: Second number
#       $a2: operation code ('+':add, '-':sub, '*':mul, '/':div)
# Return:
#       $v0: ($a0+$a1) | ($a0-$a1) | ($a0*$a1):LO | ($a0 / $a1)
#       $v1: ($a0 * $a1):HI | ($a0 % $a1)
# Notes:
#####################################################################
au_normal:

# Caller RTE store:
        addi $sp, $sp, -24
        sw $fp, 24($sp)
        sw $ra, 20($sp)
        sw $a0, 16($sp)
        sw $a1, 12($sp)
        sw $a2, 8 ($sp)
        addi $fp, $sp, 24

# au_normal:
        beq $a2, 0x2B , Addition
        beq $a2, 0x2D, Subtraction
        beq $a2, 0x2A, Multiplication
        beq $a2, 0x2F , Division
```

The normal implementations takes two argument $a0 as first number and $a1 as second number, and return value at register $v0 for addition $a0+$a1, $a0-$a1, LO 32 bit of $a0 * $a1, $a0 / $a0; register $v1 returns HI 32 bit of result of $a0 * $a1 and $a0/ $a1.

```
Addition:
        add $v0, $a0, $a1  # Add first number+ second number, store result to v0
        j RTE_restore

Subtraction:
        sub $v0 $a0, $a1  # Sub first number- second number, store result to v0
        j RTE_restore

Multiplication:
        mul $v0, $a0, $a1  # Mul first number * second number, store result to v0
        mfhi $v1           # $v1 will contains HI
        j RTE_restore

Division:
        div $a0, $a1  # Div first number / second number, store result to v0
        mflo $v0
        mfhi $v1                  # $v1 will contains LO
        j RTE_restore


# Caller RTE restore

RTE_restore:
        lw $fp, 24($sp)
        lw $ra, 20($sp)
        lw $a0, 16($sp)
        lw $a1, 12($sp)
        lw $a2, 8 ($sp)
        addi $fp, $sp, 24

# TBD: Complete it
        jr      $ra
```

Alu_normal consists of four operations: Addition, Subtraction, Multiplication and Division. All four operations are implement diretly using MIPS instructions “add”, “sub”, ““div”, “mul”. The implementation include Run-Time_Environment store and restore.

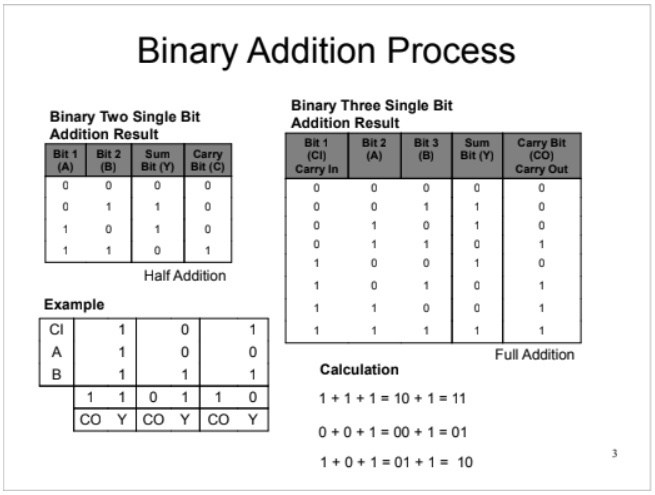
B. Logic operations

I. Au_logical:

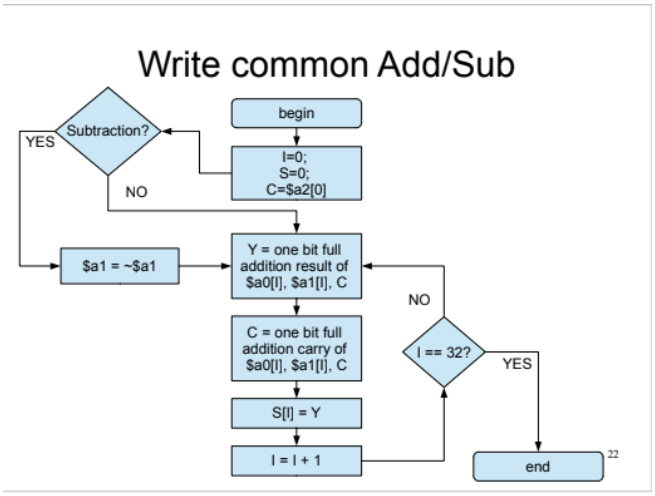
```
.include "./cs47_proj_macro.asm"
.include "./cs47_common_macro.asm"
.text
.globl au_logical
# TBD: Complete your project procedures
# Needed skeleton is given
#####################################################################
# Implement au_logical
# Argument:
#       $a0: First number
#       $a1: Second number
#       $a2: operation code ('+':add, '-':sub, '*':mul, '/':div)
# Return:
#       $v0: ($a0+$a1) | ($a0-$a1) | ($a0*$a1):LO | ($a0 / $a1)
#       $v1: ($a0 * $a1):HI | ($a0 % $a1)
# Notes:
#####################################################################
au_logical:

# au_logical:

        beq $a2, 0x2B , Addition   # ASCII for + is Hex: 2B
        beq $a2, 0x2D, Subtraction  # ASCII for - is Hex: 2D
        beq $a2, 0x2A, Multiplication  # ASCII for * is Hex: 2A
        beq $a2, 0x2F , Division  # ASCII for / is Hex: 2F
```

At beginning of the program, check $a2 operation code then go to each procedure. The operation code is follow by ASCII table: Addition-0x2b; Subtraction-0x2D; Multiplication-0x2A; Division- 0x2F.

II. Addition:



Each binary addition of two single bit produces two bits of result: carry and sum bit. The graph shows binary addition result chat.

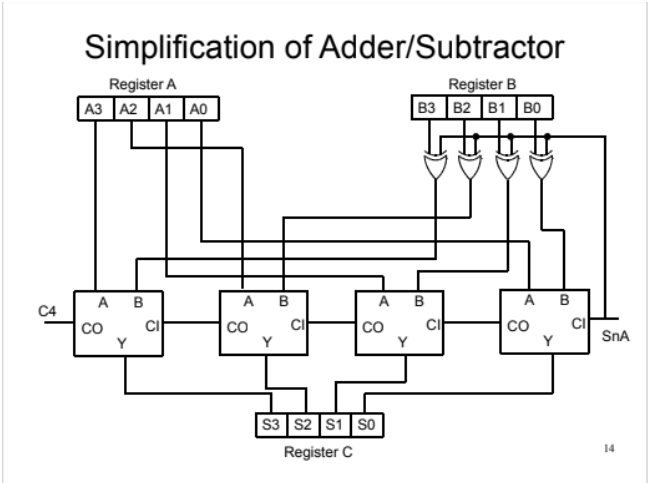


The implementation of addition follows the given chat. I is the index from 0 to 31. S is the result register. C is the carry bit for addition. For each bit position, compute sum bit using bit at I-th position in first and second number. Compute the next carry bit c, and insert the sum bit into Ith position of register result. Loop terminate when all 32 bits of result have been process.

```
        addi $sp, $sp, -60
        sw $fp, 60($sp)
        sw $ra, 56($sp)
        sw $a0, 52($sp)
        sw $a1, 48($sp)
        sw $a2, 44($sp)
        sw $a3, 40($sp)
        sw $s0, 36($sp)
        sw $s1, 32($sp)
        sw $s2, 28($sp)
        sw $s3, 24($sp)
        sw $s4, 20($sp)
        sw $s5, 16($sp)
        sw $s6, 12($sp)
        sw $s7, 8($sp)
        addi $fp, $sp, 60
        li  $s2, 0              # Set bit index  i at 0
        li  $s7, 32             # Loop run 32 times
        li  $s6, 0              # Set starting carry at 0
        jal Addition_loop
Addition_loop:

        extract_nth($s2, $a0, $s0)    # Extract bit value at i for first number
        extract_nth($s2, $a1, $s1)    # Extract bit value at i for second number
                                      # extract_nth($regNthBit, $regInput, $regResult)
        xor $s4, $s6, $s0             #  Starting carry xor  a
        and $s3, $s6, $s0             # Carry for c and a
        and $s6, $s4,  $s1            # Second carry for (c xor a) and b
        or  $s6, $s6, $s3             # Carry result= first carry or second carry
        xor $s4, $s4, $s1             # Result bit = Result bit xor second bit
        insert_to_nth_bit ($s2, $v0, $s4, $s5)     # Insert the result back to the $V0
        addi $s2, $s2, 1              # Increment i
        blt $s2, $s7, Addition_loop   # i < 32 loop
```

The picture shows the implementation of the logic addition. The loop uses extract_nth macro to get nth bit from both first and second number. Preform xor for sum bit and and for carry bit. Insert_to_nth_bit macro inserts the result bit into result register ($v0). The loop will run 32 times for 32-bit operation.

III. Subtraction

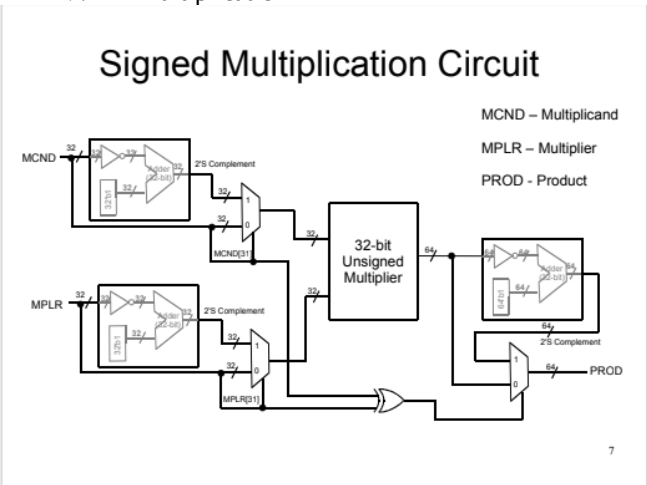


The diagram shows a circuit design for the subtraction logic. For subtraction, there is no need for additional subtraction circuit, since the subtraction is simply addition with the inversion of negative number. The XOR gate with one input as the control signal will invert the register B if the control signal pass in 1.

```
        li  $s2, 0              # Set bit index  i at 0
        li  $s7, 32             # Loop run 32 times
        li  $s6, 1              # Set starting carry to 1 for subtraction

        not $a1, $a1
        jal Subtraction_loop

Subtraction_loop:

        extract_nth($s2, $a0, $s0)   # Extract bit value at i for first number
        extract_nth($s2, $a1, $s1)   # Extract bit value at i for second number
                                     # extract_nth($regNthBit, $regInput, $regResult)
        xor $s4, $s6, $s0            # Carry xor  a
        and $s3, $s6, $s0            # Carry for c and a
        and $s6, $s4,  $s1           # Second carry for (c xor a) and b
        or  $s6, $s6, $s3            # Carry result= first carry or second carry
        xor $s4, $s4, $s1            # Result bit = Result bit xor second bit
        insert_to_nth_bit ($s2, $v0, $s4, $s5)     # Insert the result back to the $V0 (result)
        addi $s2, $s2, 1             # Increment i
        blt $s2, $s7, Subtraction_loop             # i<32, loop
        j RTE_restore
```

The implementation of subtraction follows the same diagram for add/sub chat., except invert the second number by using logic not.

IV. Multiplication



The picture shows a logic circuit for the Multiplication. Signed multiplication circuit is build onto of the unsigned multiplication circuit. At the start of the procedure, if MCND and MPLR are negative operand then they will be translated to positive value but the sign will be remembered. Then preform multiplication as unsigned operation, in the end, convert to product to corresponding sign.

- Utility Procedures:

I. Twos_complement

```
# Compute two's complement of $a0; return two's complement of $a0 at $v0
twos_complement:
# Caller RTE store:
        addi $sp, $sp, -60
        sw $fp, 60($sp)
        sw $ra, 56($sp)
        sw $a0, 52($sp)
        sw $a1, 48($sp)
        sw $a2, 44($sp)
        sw $a3, 40($sp)
        sw $s0, 36($sp)
        sw $s1, 32($sp)
        sw $s2, 28($sp)
        sw $s3, 24($sp)
        sw $s4, 20($sp)
        sw $s5, 16($sp)
        sw $s6, 12($sp)
        sw $s7, 8($sp)
        addi $fp, $sp, 60

# $a1 hold value of 1
        li $a1, 1
        not $a0, $a0    # Not $a0
        jal Addition    # ~$a1+1 = two's complement
        j RTE_restore
```

Procedure ‘two_complement’ takes a number and convert it to two’s complement by ‘~$a0+1’.

➢ Arguments:

$a0: Number of Which 2’s complement to be computed.

➢ Return:

$v0: Two’s complement of $a0.

II. Twos_complement_if_neg

```
# Compute two's complement of $a0; return two's complement of $a0 at $v0
twos_complement_if_neg:

# Caller RTE store:
        addi $sp, $sp, -60
        sw $fp, 60($sp)
        sw $ra, 56($sp)
        sw $a0, 52($sp)
        sw $a1, 48($sp)
        sw $a2, 44($sp)
        sw $a3, 40($sp)
        sw $s0, 36($sp)
        sw $s1, 32($sp)
        sw $s2, 28($sp)
        sw $s3, 24($sp)
        sw $s4, 20($sp)
        sw $s5, 16($sp)
        sw $s6, 12($sp)
        sw $s7, 8($sp)
        addi $fp, $sp, 60

        bge $a0, 0, remain_same     # Postive number, remain same
        jal twos_complement         # Negative go to two's complement
        j RTE_restore

remain_same:
        addi $v0, $a0, 0          # value at a0 remain same but move to $v0
        j RTE_restore
```

Procedure twos_complement_if_neg test $a0 value less than 0 and use twos_complement procedure to convert, if the $a0 value is equal or greater than 0 remain same.

➢ Arguments:

$a0: Number of Which 2’s complement to be computed.

➢ Return:

$v0: Two’s complement of $a0 of $a0 is negative.

III. Twos_complement_64bit:

```
twos_complement_64bit:
# Caller RTE store:
        addi $sp, $sp, -60
        sw $fp, 60($sp)
        sw $ra, 56($sp)
        sw $a0, 52($sp)
        sw $a1, 48($sp)
        sw $a2, 44($sp)
        sw $a3, 40($sp)
        sw $s0, 36($sp)
        sw $s1, 32($sp)
        sw $s2, 28($sp)
        sw $s3, 24($sp)
        sw $s4, 20($sp)
        sw $s5, 16($sp)
        sw $s6, 12($sp)
        sw $s7, 8($sp)
        addi $fp, $sp, 60

        # Invert both $a0, $a1
        not $a0, $a0
        not $a1, $a1

         #Use add_logical to add 1 to $a0
        move $t6, $a1    # Store second number $a1 to $t1
        li  $a1, 1
        jal Addition            #  Addition with $a0 and $a1 (1)
        move $t2, $v0           # Store Lo part of 2's in to t2
        move $a0, $t6    # Move original $a1 value back
        move $a1, $v1    # Set second number to the value of the carry
        jal Addition            # Addition of carry with $a1
        move $v1, $v0   # Store High part 2's in $v1
        move $v0, $t2    # Store Lo part 2's in $v0
        j RTE_restore
```

Procedure ‘twos_complement_64bit takes two registers $a0, and $a1 and convert it into 2’s complemented 64 bits. The implementation follows the step: Invert both $a0 and $a1, use add_logical to add 1 to $a0, use add logical add carry from previous step to $a1.

➢ Arguments:

$a0: Lo of the number
$a1: Hi of the number

➢ Return:

$v0: Lo part of 2's complemented 64 bits.
$v1: Hi part of 2's complemented 64 bits.

IV. bit_replicator:

```
# Return $v0 of value 0x00000000 if $a0 =0x0; value FFFFFFFF if $a0 = 0x1
bit_replicator:

# Caller RTE store:
        addi $sp, $sp, -60
        sw $fp, 60($sp)
        sw $ra, 56($sp)
        sw $a0, 52($sp)
        sw $a1, 48($sp)
        sw $a2, 44($sp)
        sw $a3, 40($sp)
        sw $s0, 36($sp)
        sw $s1, 32($sp)
        sw $s2, 28($sp)
        sw $s3, 24($sp)
        sw $s4, 20($sp)
        sw $s5, 16($sp)
        sw $s6, 12($sp)
        sw $s7, 8($sp)
        addi $fp, $sp, 60


        ble $a0, $zero, Assign_zero        # If a0 =0x0; assign v0=0x00000000
        bgt $a0, $zero, Assign_one         # If a0 =0x1; assign v0=0xffffffff
        j RTE_restore

Assign_zero:
        add $v0, $zero, 0x00000000
        j RTE_restore
Assign_one:
        add $v0, $zero, 0xffffffff
        j RTE_restore
```

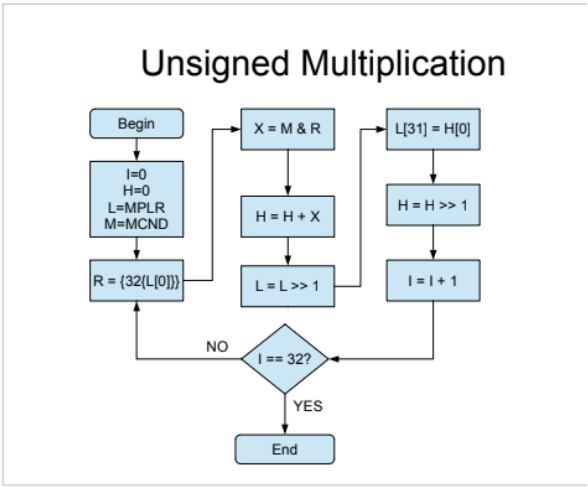
The procedure 'bit_replicator' takes a bit value 0x1 or 0x0 and return a 32-bit value 0x00000000 or 0xffffffff.

➢ Arguments:

$a0: 0x0 or 0x1; the bit value to be replicated

➢ Return:
$v0, a 32-bit value 0x00000000 or 0xffffffff depends on the $a0.

- Unsigned Multiplication:



The Unsigned Multiplication diagrams shows the procedures of unsinged multiplication.

```
# $a0: First number
# $a1: Second number
# $a2: operation code ('+':add, '-':sub, '*':mul, '/':div)
# $v0: ($a0+$a1) | ($a0-$a1) | ($a0*$a1):LO | ($a0 / $a1)
# $v1: ($a0 * $a1):HI | ($a0 % $a1)
# $s0:  L[0]
# $s1:  copy of MPLR / L
# $s2: bit index
# $s3: R
# $s4: X
# $s5: maskRegister / copy of MCND / M
# $s6: r
# $s7: 32 times
# $t1, Hold value of 31
# $t2, Hold value of H[0]
# $t3, Mask Register for insert_to_nth_bit
# $t4, H

Unsigned_Multiplication:
# Caller RTE store:
        addi $sp, $sp, -60
        sw $fp, 60($sp)
        sw $ra, 56($sp)
        sw $a0, 52($sp)
        sw $a1, 48($sp)
        sw $a2, 44($sp)
        sw $a3, 40($sp)
        sw $s0, 36($sp)
        sw $s1, 32($sp)
        sw $s2, 28($sp)
        sw $s3, 24($sp)
        sw $s4, 20($sp)
```

The picture shows the usage of each register for the unsigned multiplication

```
        li $t1, 31              # Hold value of 31 for getting 31th bit
        move $s1, $a1     # L= second number
        move $s5, $a0     # M = first number

        jal  Unsigned_Multiplication_loop

        j RTE_restore


Unsigned_Multiplication_loop:

        extract_nth($zero,$s1, $a0)     # extract_nth($regNthBit, $regInput, $regResult)
        jal bit_replicator
        move $s3, $v0           # Move result of bit replicator to R
        and $s4, $s5, $s3               # X= M & R


        move $a0, $t4           # Move H to $a0 for addition
        move $a1, $s4                   # Move x to $a1 for adition
        jal Addition                    # Addition
        move $t4, $v0           # H = Hnx
        srl  $s1, $s1, 1                # L= L>>1
        extract_nth($zero, $t4, $t2)    # extract_nth($regNthBit, $regInput, $regResult)
        insert_to_nth_bit($t1, $s1  ,$t2, $t5)  #insert_to_nth_bit ($regNthBit, $regInput, $regValue, $maskRegister)

        srl $t4, $t4, 1                 # H= H>>1
        addi $s2, $s2, 1                # Increment i

        blt  $s2, $s7, Unsigned_Multiplication_loop


        move $v0, $s1           # End result of Lo
        move $v1, $t4           # End result of H
        j RTE_restore
```

The Unsigned Multiplication initialize the L as MPLR and M as MCND. Then use replication procedure to replicate single bit value of L [0] to determine R. X=M & R to determine the result bit of the multiplication and add X to the result register. Right shift 1 bit L and insert H [0] bit into L [31] bit, then right shift 1 bit of H register.

The procedure 'Unsigned_multiplicatin' preforms the bit multiplication.

- ○ Arguments:

  $a0: Multiplicand
  $a1: Multiplier

- ○ Return:

  $v0: Lo part of result
  $v1: Hi part of result

- Signed Multiplication:

```
# $a0: First number
# $a1: Second number
# $a2: operation code ('+':add, '-':sub, '*':mul, '/':div)
# $v0: ($a0+$a1) | ($a0-$a1) | ($a0*$a1):LO | ($a0 / $a1)
# $v1: ($a0 * $a1):HI | ($a0 % $a1)
# $s0: copy of MPLR
# $s1: N2
# $s2: bit index
# $s3: R
# $s4: X
# $s5: N1
# $s6: r
# $s7: H
# $t1, Hold value of 31
# $t2, Hold value of L[31]
# $t3, Mask Register for insert_to_nth_bit
# $t4, $a0[31]
# $t5, $a1[31]
# $t6, S
# $t7, 1

Signed_Multiplication:

        # Caller RTE store:
        addi $sp, $sp, -60
        sw $fp, 60($sp)
        sw $ra, 56($sp)
        sw $a0, 52($sp)
        sw $a1, 48($sp)
        sw $a2, 44($sp)
        sw $a3, 40($sp)
        sw $s0, 36($sp)
```

The picture shows the usage of each register in the implementation of signed multiplication.

```
        li $t1, 31

        extract_nth($t1, $a0, $t4) # extract_nth($regNthBit, $regInput, $regResult)
        extract_nth($t1, $a1, $t5)

        move $s5, $a0           # N1= $a0
        move $s1, $a1           # N2= $a1
        move $a0, $s5           # Move N1 to $a0 for two's complement caller
        jal twos_complement_if_neg
        move $s5, $v0    # N1= $v0
        move $a0, $s1    # Move N2 to $a0 for two's complement caller
        jal twos_complement_if_neg  # Move N2 to $a0 for two's complement caller
        move $s1, $v0    # Move two's complement back to $s1
        move $a0, $s5   # Move N1 to $a0, for Unsigned_Mul
        move $a1, $s1    # Move N2 to $a0, for Unsigned_Mul
        xor $t6, $t4, $t5
        jal Unsigned_Multiplication

        move $a0, $v0      # For twos_complement_64 bit
        move $a1, $v1      # For twos_complement_64 bit

        bgt $t6, $zero, twos_complement_64bit
        #jal  twos_complement_64bit #twos_complement_64bit
        j RTE_restore
```

The Signed Multiplication initialize N1 = $a0, and N2= $a1. Then check N1 and N2, if the value is smaller than 0 then make two's complement. Call Unsigned Multiplication using N1 and N2 two's complement. Determine the sign S of result by xor $a0[31], $a1[31], if the result is 1, use the twos_complement_64bit to determine two complement form of 64-bit number.

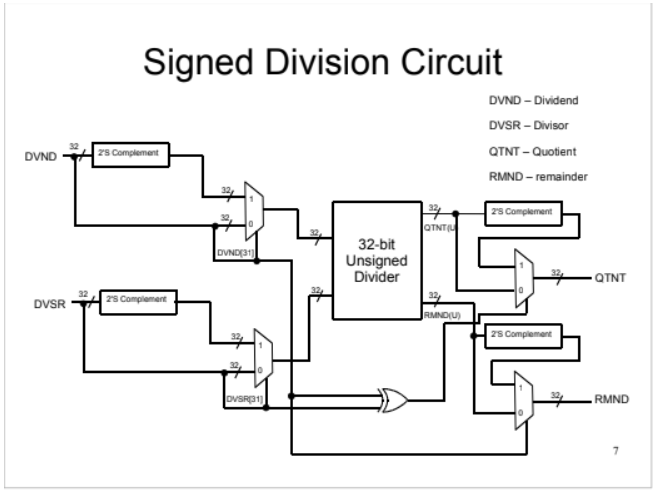
- Arguments:

  $a0: Multiplicand
  $a1: Multiplier

- Return:

  $v0: Lo part of result
  $v1: Hi part of result

V. Division



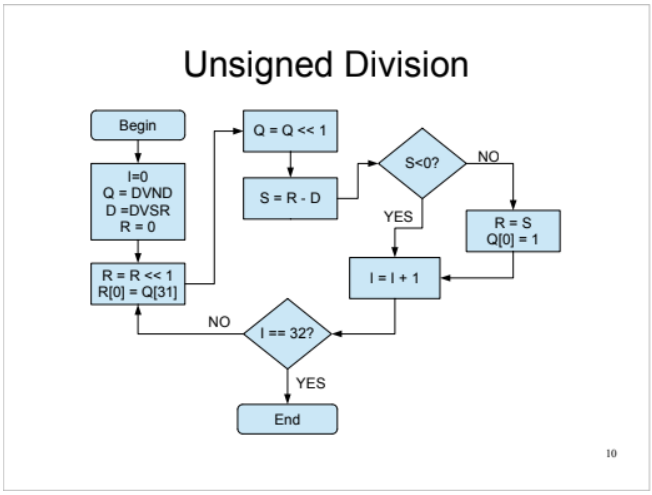
The diagram shows a signed division circuit. If both DVND and DVSR have the same sign, the result is positive, if DVND and DVSR have different sign, the result will the negative sign. The remainder will follow the sign of the DVND.

- Division:

```
Division:

        # Caller RTE store:
        addi $sp, $sp, -60
        sw $fp, 60($sp)
        sw $ra, 56($sp)
        sw $a0, 52($sp)
        sw $a1, 48($sp)
        sw $a2, 44($sp)
        sw $a3, 40($sp)
        sw $s0, 36($sp)
        sw $s1, 32($sp)
        sw $s2, 28($sp)
        sw $s3, 24($sp)
        sw $s4, 20($sp)
        sw $s5, 16($sp)
        sw $s6, 12($sp)
        sw $s7, 8($sp)
        addi $fp, $sp, 60

        li $s2, 0               # Bit index i =0
        li $s7, 32

        jal Signed_Dvision
        j RTE_restore
```

The picture shows the RTE_store for the division, and we initialize the ith bit to 0 and maximum loop times to 32.

- Division_unsigned:



The diagram shows the procedures of Unsigned Division.

```
# $a0: First number
# $a1: Second number
# $a2: operation code ('+':add, '-':sub, '*':mul, '/':div)
# $v0: Quotient
# $v1: Remainder
# $s0: Q = DVND
# $s1: D = DVSR
# $s2: bit index
# $s3: R
# $s4: Hold Value of 31
# $s5: S
# $s6: maskRegister for insert bit
# $s7: Maximum bit 32
# $t0: bit value at $v1 after mul
# $t2: Hold value of 1
# $t3: MaskRegister for Q[0]=1
# $t4: Q[31]
div_unsigned:

# Caller RTE store:
        addi $sp, $sp, -60
        sw $fp, 60($sp)
        sw $ra, 56($sp)
        sw $a0, 52($sp)
        sw $a1, 48($sp)
        sw $a2, 44($sp)
```

The picture indicates the usage of each register for the division_unsigned.

```
        move $s0, $a0               # Q= DVND
        move $s1, $a1               # D= DVSR
        li $s3, 0                   # R=0
        li $s6, 0                   # Empty the mask register
        li $s5, 0
        li $t0, 0
        li $t2, 0                   # Empty the temporary register
        li $t3, 0                   # Empty the mask register
        li $t4, 0                   # Q[31], empty the temporary register
        jal div_unsigned_loop
        j RTE_restore


div_unsigned_loop:
        sll  $s3, $s3, 1            # R= R<<1
        li $s4, 31                  # 31 Value for extract 31th bit
        extract_nth($s4,$s0,$t4)   # extract_nth($regNthBit, $regInput, $regResult)
        insert_to_nth_bit($zero, $s3, $t4,$s6)      #insert_to_nth_bit ($regNthBit, $regInput, $regValue, $maskRegister)
        sll $s0, $s0, 1             # Q=Q <<1
        move $a0, $s3               # Move R to $a0 for sub
        move $a1, $s1               # Move D to $a1 for sub
        jal Subtraction
        move $s5, $v0              # Set S to the sub result

        blt $s5, $zero, increment_i
        #addi $s2, $s2, 1           # Increment i
        #blt  $s2, $s7, div_unsigned_loop      # i== 32?
        move $s3, $s5              # R=s
        li $t2, 1
        insert_to_nth_bit($zero, $s0, $t2,$t3)      #insert_to_nth_bit ($regNthBit, $regInput, $regValue, $maskRegister)
        #jal increment_i
```

```
        #jal increment_i

increment_i:

        addi $s2, $s2, 1            # Increment i
        bne     $s2, $s7, div_unsigned_loop      # i== 32?
        move $v0, $s0
        move $v1, $s3
        j RTE_restore
```

The two pictures show the implementation of the division_unsigned. We first take DVND and assign to register to Q; assign DVSR to D. Then left right R by 1 and insert Q [31] to R [0]. Then left shift Q by 1. Compute S by using the sub_logical we implemented before. Check if S>0, if so we make R=S and Q [0] =1. Then increment I and iterate the loop 32 times.

- Arguments:
  - $a0: Dividend
  - $a1: Divisor
- Return:
  - $v0: Quotient
  - $v1: Remainder

➢ Signed Division:

```
# $a0: First number
# $a1: Second number
# $a2: operation code ('+':add, '-':sub, '*':mul, '/':div)
# $v0: Quotient
# $v1: Remainder
# $s0: N1 = DVND
# $s1: N2 = DVSR
# $s2: bit index
# $s3: R
# $s4: Hold Value of 31
# $s5: a0[31]
# $s6: a1[31]
# $s7: Maximum bit 32
# $t0: a0[31]
# $t2: a1[31]
# $t3: MaskRegister for Q[0]=1
# $t4: a0[31]
# $t5: Q
# $t6: R
# $t7: S
Signed_Dvision:

        # Caller RTE store:
        addi $sp, $sp, -60
        sw $fp, 60($sp)
        sw $ra, 56($sp)
        sw $a0, 52($sp)
        sw $a1, 48($sp)
        sw $a2, 44($sp)
        sw $a3, 40($sp)
        sw $s0, 36($sp)
```

The picture shows the usage of each register for the signed Division.

```
        li $s4, 31          # Hold value of 31
        extract_nth($s4,$a0, $s5)   # extract_nth($regNthBit, $regInput, $regResult)
        extract_nth($s4,$a1, $s6)   # extract_nth($regNthBit, $regInput, $regResult)
        move $s0, $a0       # N1= $a0
        move $s1, $a1       # N2= $a1
        move $a0, $s0       # Move N1 to $a0 for two's complement caller
        jal twos_complement_if_neg
        move $s0, $v0       # N1= $v0
        move $a0, $s1       # Move N2 to $a0 for two's complement caller
        jal twos_complement_if_neg
        move $s1, $v0       # N1= $v0

        move $a0, $s0       # Move N1 to $a0 for unsigned div caller
        move $a1, $s1       # Move N2 to $a1 for unsigned div caller

        xor $t7, $s5, $s6   # Sign of X

        jal div_unsigned
        move $t1, $v0       # Q= $v0
        move $t6, $v1       # R= $v1


        # Determine sign of Q and R
        jal Change_sign
        j RTE_restore
```

```
# Change sign of R
Change_sign:
        beq $s5, $zero, Change_sign_Q    # a[31] is zero, dont change R, else change Q to twos_complement
        move $a0, $t6
        jal twos_complement
        move $t6, $v0
        jal Change_sign_Q

# Change sign of Q
Change_sign_Q:

        beq $t7, $zero, finished        # S is zero, finished
        move $a0, $t1
        jal  twos_complement
        move $t1, $v0
        jal finished

finished:
        move $v0, $t1               # Move Q to v0
        move $v1, $t6
        j RTE_restore               # Moce R to v1


# Caller RTE restore

RTE_restore:

        lw $fp, 60($sp)
        lw $ra, 56($sp)
        lw $a0, 52($sp)
```

The two picture shows the implementation of the signed Division. We first assign $a0 to N1 register and $a1to N2 register. Then we make DVND and DVSR two's complement if negative. Called unsigned Division using N1 and N2 and store results Q and R. Then Determine the sign of Q and R. If a0[31] xor a1[31] is 1 then the Q is negative, otherwise positive. The sign of remainder is determined by the a0[31].

- Arguments:
  - $a0: Dividend
  - $a1: Divisor
- Return:
  - $v0: Quotient
  - $v1: Remainder

C. Macros

Extract_nth_bit:

```
# Get nth bit of the register
# regResult:  contain 0x0 or 0x1 depending on nth bit being 0 or 1
# regNthBit: Bit position
# regInput: Source bit pattern
.macro extract_nth($regNthBit, $regInput, $regResult)

        addi $regResult, $zero, 1  # Set result to 1 for AND logic operation
        sllv $regResult, $regResult, $regNthBit # Left shift regInput by regNthBit of bit
        and $regResult, $regInput, $regResult  # Bitwise AND for the result and input ; save Bitwise AND result into regResult
        srlv $regResult, $regResult, $regNthBit
.end_macro
```

Utility Macros that extract nth bit from a bit pattern. This macro takes three registers $regNthBit, $regInput, $regResult.

- $regNthBit: Bit position n (0-31)
- $regInput: Source bit pattern
- $regResult: nth bit value, will contain 0x0 or 0x1

Insert_to_nth_bit:

```
# Insert bit 1 or 0 at nth bit to a bit pattern
# regValue: contain 0x0 or 0x1 depending on nth bit being 0 or 1
# regNthBit: position of the bit that we want to get
# regInput: Source register
# maskRegister: copy of the input
.macro insert_to_nth_bit ($regNthBit, $regInput, $regValue, $maskRegister)

        li   $maskRegister , 1     # Set maskRegister to value of 1
        sllv $maskRegister, $maskRegister, $regNthBit   # Left shift nth bit to match with the input
        not  $maskRegister, $maskRegister     # Invert maskRegister
        and  $regInput, $maskRegister, $regInput          # the number AND mask
        sllv $regValue, $regValue, $regNthBit          # Left shift the mask by n bit
        or   $regInput, $regValue, $regInput    # insert value to nth bit
        .end_macro
```

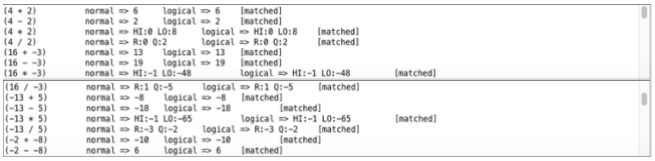
Utility Macros that insert bit 1 at nth bit to a bit pattern. This macro takes three registers $regNthBit, $regInput, $regValue, $maskRegister.

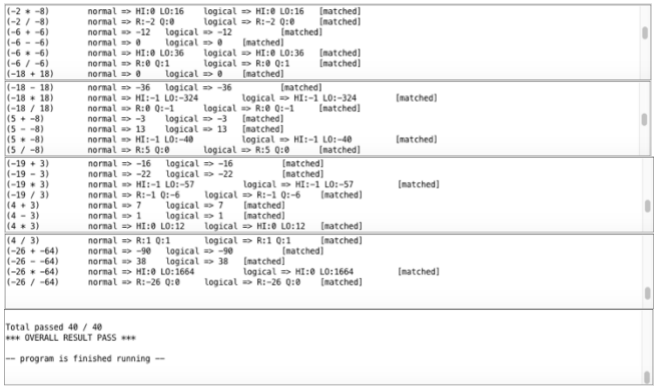
- $regNthBit: Bit position n (0-31)
- $regInput: Source bit pattern
- $regValue: Register that contains 0x1 or 0x0 (bit value to insert)
- maskRegister: Register to hold temporary mask

D. Some Problem faced during the implementation

- Infinite loop caused the program stuck
- Random errors that caused by the improperly store and restore frame.
- Testing macros
- Difficult to test code piece by piece

IV. Testing





The pictures show the testing results of the au_logical and au_normal computation.

V. Conclusion

The project practiced the concept of logical addition, subtraction, multiplication and division. Through the implementation of the code, the procedures demonstrate how are the logic circuits work. The project enhanced the skills of MIPS assembly language programming.

References

Patra, Kaushik. (2018). CS47-Lecture 18-20 [PPT]. Retrieved from the SJSU Canvas.

IEEE conference templates contain guidance text for composing and formatting conference papers. Please ensure that all template text is removed from your conference paper prior to submission to the conference. Failure to remove template text from your paper may result in your paper not being published.