6 Integer Multiplication and Division

6.1 Objectives

After completing this lab, you will:

- Understand binary multiplication and division
- Understand the MIPS multiply and divide instructions
- Write MIPS programs that use integer multiplication and division

6.2 Binary Multiplication

Sequential binary multiplication is a simple but slow form of multiplication. It is performed using addition and shift operations as illustrated in Figure 6.1. The multiplication of two 32-bit integers is a 64-bit product stored in two registers: HI and LO.

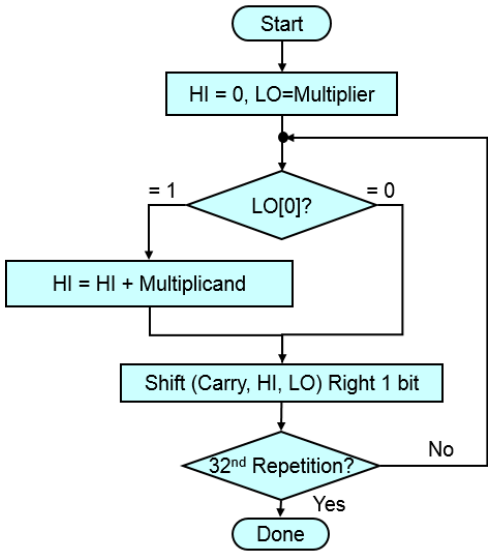

Figure 6.1: Sequential Binary Multiplication Algorithm

Register HI is initialized with the value 0, and LO is loaded with the value of the multiplier. The sequential algorithm is repeated 32 times for each bit of the multiplier. Finally, the product is computed in two registers HI and LO.

Figure 6.2 shows a simple sequential binary multiplier for unsigned integers. It uses a 32-bit ALU, a register for storing the multiplicand, a shift register (HI and LO) for storing the final product, and simple control. Instead of shifting the multiplicand to the left, the product is shifted to the right. It has the same net effect and computes the same result. The control examines each bit of the multiplier LO[0]. If a bit of the multiplier is 1, then an addition is done: HI = HI + Multiplicand. Then the HI and LO registers are shifted to the right and the carry bit is inserted. This is repeats 32 times to compute the 64-bit product in HI and LO.

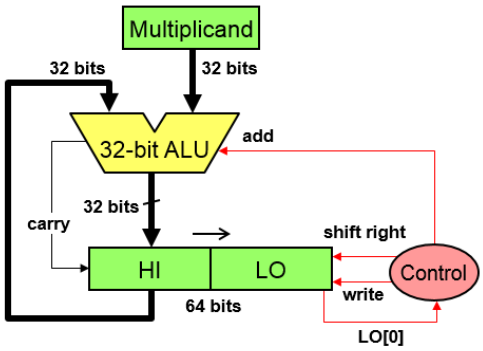

Figure 6.2: Sequential Binary Multiplier

Signed multiplication can also be performed using the same sequential binary multiplier, but with minor modification. When adding (HI + Multiplicand) the proper sign bit of the result is computed, instead of the carry bit. When shifting the HI and LO registers to the right, the sign-bit is inserted to the left of the product. Additions are used for the first 31 steps. However, the last step should use subtraction (rather than addition), if the sign-bit of the multiplier is negative.

Sequential binary multiplication is slow because it requires one cycle for each bit of the multiplier. Faster binary multiplication can be done in hardware, as shown in Figure 6.3. The cost of the binary multiplier increases because it uses many adders, instead of just one used as in Figure 6.2.

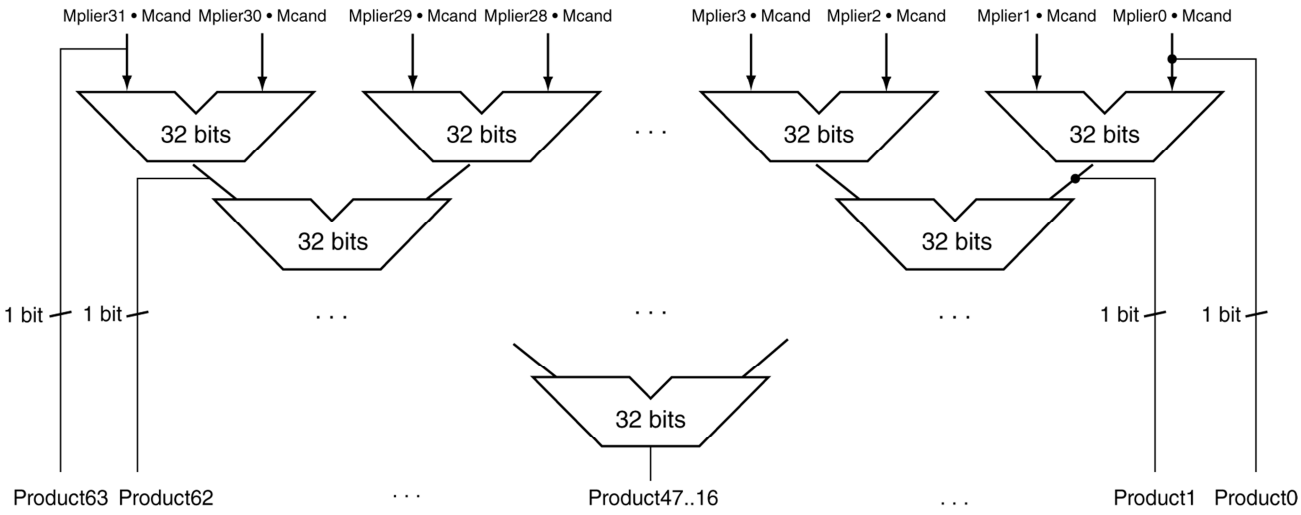

Figure 6.3: Faster Integer Multiplier

6.3 Binary Division

Sequential binary division can be performed using shift and subtract operations. Binary division produces a quotient and a remainder. It also uses two registers HI and LO. The quotient is computed in the LO register and the remainder is computed in the HI register. HI is initialized with zero and LO is initialized with the dividend. At each iteration step, registers HI and LO are shifted left by 1 bit. The difference: (HI – Divisor) is computed. If this difference is ≥ 0, the Remainder (HI register) is set equal to the difference and the least significant bit of the quotient (LO register) is set to 1. The sequential binary division algorithm is repeated 32 times, as shown in Figure 6.4.

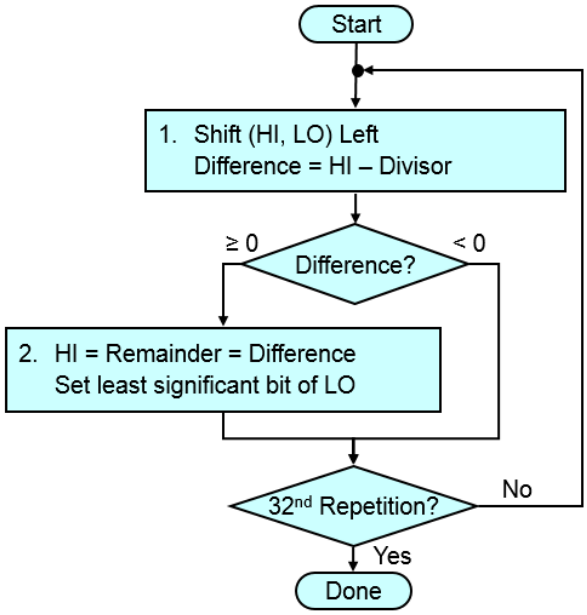

Figure 6.4: Binary Division Algorithm

A simple but slow sequential binary divider is shown in Figure 5.5. It uses a 32-bit ALU that does subtraction. It also uses HI and LO registers, a Divisor register, and simple control logic to shift left the HI and LO registers and set the least-significant bit of LO. The control logic repeats 32 times to compute each bit of the quotient in LO. The final remainder will be in the HI register.

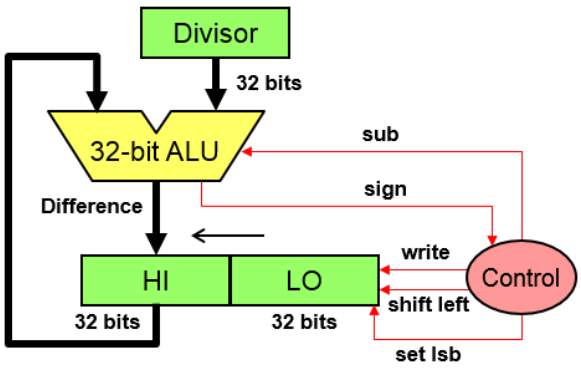

Figure 6.5: Sequential Binary Divider

6.4 MIPS Integer Multiply and Divide Instructions

Multiplication and division generate results that are larger than 32 bits. Multiplication produces a 64-bit product while division produces a 32-bit quotient and a 32-bit remainder. The MIPS architecture provides two special 32-bit registers that are the target for integer multiply and divide instructions. The source operands come from the general-purpose register file. However, the results are written into HI and LO registers**.** For multiplication, the HI and LO registers form a 64-bit product. For division, HI stores the 32-bit remainder, while LO stores the 32-bit quotient.

MIPS defines two multiply instructions: **`mult`** for signed multiplication and **`multu`** for unsigned multiplication. Both multiply instructions produce a 64-bit product without overflow. There is also a third **`mul`** instruction that computes the same 64-bit product as a **`mult`** instruction in HI and LO registers. However, the **`mul`** instruction also copies the LO register into destination register **`Rd`**. The **`mul`** instruction is useful when the product is small and can fit in a 32-bit destination register.

Instruction	Meaning	Note
`mult Rs, Rt`	`[HI, LO] = Rs × Rt`	Signed multiplication
`multu Rs, Rt`	`[HI, LO] = Rs × Rt`	Unsigned multiplication
`mul Rd, Rs, Rt`	`[HI, LO] = Rs × Rt; Rd = LO`	**`Rd`** = Lower 32-bit of the product

Table 6.1: MIPS Integer Multiply Instructions

In addition, MIPS defines two integer divide instructions: **`div`** for signed division and **`divu`** for unsigned division. The quotient of the integer division is saved in the LO register, while the remainder is saved in the HI register as shown in Table 6.2.

Instruction	Meaning	Note
`div Rs, Rt`	`LO = Rs / Rt, HI = Rs % Rt`	Signed division
`divu Rs, Rt`	`LO = Rs / Rt, HI = Rs % Rt`	Unsigned division

Table 6.2: MIPS Integer Divide Instructions

If the divisor in register **`Rt`** is zero, then the MIPS divide instructions do not compute any result in the HI and LO registers. Division by zero is ignored and no exception is produced. The MIPS programmer must ensure that the divisor is non-zero when using the integer divide instruction.

Special instructions are used to move data between the HI and LO registers and general-purpose registers. These are listed in Table 6.3.

Instruction	Meaning	Instruction	Meaning
`mfhi Rd`	`Rd = HI`	`mthi Rs`	`HI = Rs`
`mflo Rd`	`Rd = LO`	`mtlo Rs`	`LO = Rs`

Table 6.3: Move Instructions for the HI and LO registers

6.5 Applications of Integer Multiply and Divide Instructions

Raising an integer number **x** to a power **n** (x^n), may be computed using successive multiplications. The following code uses integer multiplication to implement the power function. Registers **$a0** and **$a1** are used to store **x** and **n**, while **$v0** contains the result.

```
   li     $a0, 7              # number x
   li     $a1, 5              # number n
   li     $v0, 1              # $v0 = 1
pow:
   mul    $v0, $v0, $a0       # $v0 = $v0, $a0
   addiu  $a1, $a1, -1        # decrement n
   bnez   $a1, pow            # loop if (n != 0)
```

The greatest common divisor can be computed as follows:

```
   gcd(a, 0) = a
   gcd(a, b) = gcd(b, a % b)     where % is the remainder operator
```

For example,

```
   gcd(30, 18) = gcd(18, 30%18) =
   gcd(18, 12) = gcd(12, 18%12) =
   gcd(12, 6)  = gcd(6,  12%6)  = gcd(6, 0) = 6
```

The following MIPS loop computes the **gcd** of two numbers stored in registers **$a0** and **$a1**. The final result is computed in register **$a0**.

```
   li     $a0, 30             # number a
   li     $a1, 18             # number b
gcd:
   div    $a0, $a1            # HI = remainder, LO = quotient
   move   $a0, $a1            # $a0 = number b
   mfhi   $a1                 # $a1 = a % b (remainder)
   bnez   $a1, gcd            # loop if (b != 0)
```

A string is an array of bytes stored in memory. For example, **str** is defined as a string of digits in the data segment. The **.asciiz** directive is used to define an ASCII string stored in memory and terminated with a NULL byte.

```
.data
str:  .asciiz    "512943687"
```

The string of digits can be read from memory and converted into a number. The **lb** (load byte) instruction can read each character of a string from memory into a register. The following MIPS loop converts the above string **str** into an integer computed in register **$v0**:

```
.text
main:
   la     $t0, str            # load address of str into $t0
   li     $v0, 0              # Initialize $v0 = 0
   li     $v1, 10             # Initialize $v1 = 10
   lb     $t1, ($t0)          # load byte: $t1 = Memory($t0)
str2int:
   addiu  $t1, $t1, -48       # convert character to a number
   mul    $v0, $v0, $v1       # $v0 = $v0 * 10
   addu   $v0, $v0, $t1       # $v0 = $v0 + digit
   addiu  $t0, $t0, 1         # point to next character in memory
   lb     $t1, ($t0)          # load byte: $t1 = Memory($t0)
   bnez   $t1, str2int        # loop back if not NULL character
done:
   . . .                      # $v0 = integer result
```

An unsigned integer can be converted to a string by successive divisions by **10**. The remainder is a digit between **0** and **9**. The remainder digits are computed backwards starting at the least significant digit. Each remainder is then converted to an ASCII character and stored in memory in a string. For example, consider converting the integer **5218** into a string. This can be done as follows:

```
5128 / 10 = 512,  5128 % 10 = 8   ➔ Convert 8 into character '8'
 512 / 10 = 51,    512 % 10 = 2   ➔ Convert 2 into character '2'
  51 / 10 = 5,      51 % 10 = 1   ➔ Convert 1 into character '1'
   5 / 10 = 0,       5 % 10 = 5   ➔ Convert 5 into character '0'
Stop when the quotient is zero
```

The following MIPS code converts an unsigned integer stored in register **$a0** into a string stored in the data segment in memory. The string is initialized with **10** space characters. The string has **10** characters only because a 32-bit unsigned integer can have at most **10** digits.

```
.data
str: .asciiz   "          "       # str = 10 space characters
.text
main:
   li     $a0, 5128               # $a0 = unsigned integer to convert
   la     $v0, str                # load address of str into $v0
   addiu  $v0, $v0, 11            # $v0 = pointer at end of str
   li     $a1, 10                 # Initialize $a1 = 10
int2str:
   divu   $a0, $a1                # divide $a0 by 10
   mflo   $a0                     # $a0 = quotient
   mfhi   $t0                     # $t0 = remainder (0 to 9)
   addiu  $t0, $t0, 48            # convert digit into a character
   addiu  $v0, $v0, -1            # point to previous space character
   sb     $t0, ($v0)              # store byte: Memory($v0) = $t0
   bnez   $a0, int2str            # loop back if quotient is not zero
done:
   . . .                          # $v0 = pointer to string in memory
```

6.6 In-Lab Tasks

1. Write MIPS code to perform the following integer multiplications. What is the value of the LO and HI registers?
   a) **98765 × 54321** using the **multu** instruction
   b) **-98765 × -54321** using the **mult** instruction

2. Write MIPS code to perform the following integer divisions. What is the value of the LO and HI registers?
   a) **98765 / 54321** using the **divu** instruction
   b) **-98765 / -54321** using the **div** instruction

3. Factorial Calculation: Using the **mul** instruction, write a MIPS program that computes the factorial of a number **n** input by the user, and display the result on the screen. Run your code and record the maximum 32-bit factorial value that can be computed without errors.

4. The string-to-integer program presented in Section 6.5 converts a string of decimal digits to an unsigned integer using successive multiplications by **10** and additions. It is also possible to convert a string of digits in any radix system to an integer, using successive multiplications by the radix value and additions. Rewrite the string-to-integer program asking the user to input a

radix value between **2** and **10** and a string of digits in the specified radix system. For example, if the radix value is **8** then the string can only have octal digit characters from **'0'** to **'7'**. Convert the string of digit characters into an unsigned integer and display the value of the unsigned integer.

5. The integer-to-string program presented in Section 6.5 converts an unsigned integer to string format using successive division by **10** and storing the remainder digit characters in a string. It is also possible to convert the unsigned integer to any radix using successive divisions by the radix value. Rewrite the integer-to-string program asking the user to input an unsigned integer and a radix value between **2** and **10**. Do the radix conversion and then print the string. Make sure that the string has sufficient space characters, especially when converting to radix **2**.

6. Fraction computation: Using successive integer multiplications and divisions, write a MIPS program that divides an integer ***x*** by another integer ***y*** that are read as input. The result of the division should be in the form: ***a.b***, where ***a*** is the integer part and ***b*** is the fractional part. Compute the fraction ***b*** with **8** digits after the decimal point. Display the result in the form ***a.b***.