Arithmetic for Computers

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

Addition and Subtraction

- MIPS uses two's complement
- Addition:
  - `a + b`: Straightforward
  - (4-bit) $3_{10} + (-5_{10}) = 0011_2 + 1011_2 = 1110_2 = -2_{10}$
- Subtraction:
  - $a - b = a + (-b) = a + (\bar{b} + 1)$
  - Invert `b` and set `CarryIn` of the least significant bit ALU to 1
  - (4-bit) $5_{10} - 3_{10} = 5_{10} + (-3_{10}) = 0101_2 + (1100_2 + 0001_2) = 0010_2 = 2_{10}$
- Need to catch overflows

Overflow Examples

- Consider 4-bit representations:
  - $5_{10} + 4_{10}$
    - $5_{10} + 4_{10} = 0101_2 + 0100_2 = 1001_2$
    - But 1001_2 in two's complement is -7_{10}
    - The answer should be 9_{10}
  - $(-3_{10}) - 7_{10}$
    - $(-3_{10}) - 7_{10} = (-3_{10}) + (-7_{10}) = 1101_2 + 1001_2 = 0110_2$
    - But 0110_2 is equal to 6_{10}
    - The answer should be -10_{10}
- Can we generate an overflow if we try to add two operand with different sign?
  - $0_{10} + (-8_{10}) = 0000_2 + 1000_2 = 1000_2 = -8_{10}$ no overflow
  - $7_{10} + (-1_{10}) = 0111_2 + 1111_2 = 0110_2 = 6_{10}$ no overflow

Overflow in Addition and Subtraction

- Adding two operands with different signs, no overflow
- Subtracting two operands with same signs, no overflow
  - Both positive: `a - b` = `a + (-b)`. Similar to addition with different signs
  - Both negative: `-a - (-b)` = `-a + b`. Similar to addition with different signs
- When overflow occurs, the sign bit represents value, not sign
- Example (4-bit representation)
  - $7 + 2 = 0111_2 + 0002_2 = 1001_2$. The actual result is 9 not -7
  - If we use 5 bit representation, $01001_2 = 9$
  - $-4 + -5 = 1100_2 + 1011_2 = 0111_2$. The actual result is -9 to 7
  - If we use 5 bit representation, $10111_2 = -9$

Overflow in Addition and Subtraction

- Addition: Overflow occurs in the following situations:
  - add two positive numbers and the sum is negative
  - add two negative numbers and the sum is positive
- Subtraction: Overflow occurs in the following situation
  - subtract a negative number from a positive number and get a negative result
  - subtract a positive number from a negative number and get a positive result

Operation	Operand A	Operand B	Result (overflow)
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Overflow in MIPS

- In MIPS, an exception occurs when it detects an overflow
  - `add`, `addi`, and `sub`
- No exception for unsigned operations (`addu`, `addiu`, and `subu`) (C compiler uses these)
- When MIPS detects an overflow with an exception
  - Save the address of instruction that causes overflow in a register called *exception program counter* (EPC)
  - Jump to predefined address to run a routine
  - To copy the value of EPC to a general purpose register, uses `mfc0` (Move From system Control) instruction

Testing Overflow without Exception

- Complier can check overflow without using exception
- For signed addition:
  - check whether all 3 signs (two operands and sum) are the same

```
addu $t0, $t1, $t2      # $t0 = $t1 + $t2, no exception
xor  $t3, $t1, $t2      # $t3 = 1x..x if sings are different
slt  $t3, $t3, $zero    # $t3 = 1 if signs are different
bne  $t3, $zero, noOvF # Sign are different, no overflow
xor  $t3, $t0, $t1      # $t3 = 1x..x if signs are different
slt  $t3, $t3, $zero    # $t3 = 1 if signs are different
bne  $t3, $zero, OvF    # signs are different, overflow
```

Testing Overflow without Exception

- For unsigned addition:
  - Check whether result is greater than $2^{32} - 1$

```
addu $t0, $t1, $t2    # $t0 = $t1 + $t2, no exception
nor  $t3, $t1, $zero  # $t3 = Not of $t1 ((-1 x $t1) - 1)
sltu $t3, $t3, $t2    # $t3 = 1 if (2^32 - $t1 - 1) < $t2
                      # (2^32 - 1) < ($t2 + $t1)
bne  $t3, $zero, OvF  # go to overflow
```

Multimedia Support

- Perform tasks in parallel
- In 32-bit wide ALU, Suppose we can select whether
  1. the `CarryIn` of ALU8 should be 0, 1, or `CarryOut` of ALU7
  2. the `CarryIn` of ALU16 should be 0, 1, or `CarryOut` of ALU15
  3. the `CarryIn` of ALU24 shluld be 0, 1, or `CarryOut` of ALU 23
- Called *Partitioning ALU*
- Allow us to perform
  - Add or subtract four sets of 8-bit operands at the same time
  - Add or subtract two sets of 16-bit operands at the same time

Multiplication

- How to perform binary multiplication
- $a \times b = c$
  - a: multiplicand
  - b: multiplier
  - c: product (result)
- The size of the result is the sum of the sizes of operands
- Elementary/Middle School paper and pencil method can be used
- How to handle sign of operands?
  1. If operands have the same sign, result is positive. Otherwise, negative
  2. Use sign extension

Examples (4-bit Representation)

$5_{10} \times 6_{10} = 0101_2 \times 0110_2$ with sign extension to 8 bits $00000101_2 \times 00000110_2$ discard anything beyond 8-bit representation

$$
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \quad \times \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & \\
0 & 0 & 0 & 1 & 0 & 1 & & \quad + \\
0 & 0 & 0 & 0 & 0 & & & \\
0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & & & & & \\
0 & 0 & & & & & & \\
0 & & & & & & & \\
\hline
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
\hline
\end{array}
$$

which is 30_{10}

Examples (4-bit Representation)

$5_{10} \times -6_{10} = 0101_2 \times 1010_2$ with sign extension to 8 bits
$00000101_2 \times 11111010_2$ discard anything beyond 8-bit representation

$$
\begin{array}{cccccccc|c}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & \times \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & + \\
0 & 0 & 1 & 0 & 1 & & & & \\
0 & 1 & 0 & 1 & & & & & \\
1 & 0 & 1 & & & & & & \\
0 & 1 & & & & & & & \\
1 & & & & & & & & \\
\hline
1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & \\
\hline
\end{array}
$$

which is -30_{10}

Examples (4-bit Representation)

$-5_{10} \times 6_{10} = 1011_2 \times 0110_2$ with sign extension to 8 bits $11111011_2 \times 00000110_2$ discard anything beyond 8-bit representation

1	1	1	1	1	0	1	1	
0	0	0	0	0	1	1	0	$\times$
0	0	0	0	0	0	0	0	
1	1	1	1	0	1	1		
1	1	1	0	1	1			$+$
0	0	0	0	0				
0	0	0	0					
0	0	0						
0	0							
0								
1	1	1	0	0	0	1	0	

which is -30_{10}

Examples (4-bit Representation)

$-5_{10} \times -6_{10} = 1011_2 \times 1010_2$ with sign extension to 8 bits $11111011_2 \times 11111010_2$ discard anything beyond 8-bit representation

$$
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \quad \times \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & \\
0 & 0 & 0 & 0 & 0 & 0 & & \quad + \\
1 & 1 & 0 & 1 & 1 & & & \\
1 & 0 & 1 & 1 & & & & \\
0 & 1 & 1 & & & & & \\
1 & 1 & & & & & & \\
1 & & & & & & & \\
\hline
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
\hline
\end{array}
$$

which is 30_{10}

Two's Complement Multiplication

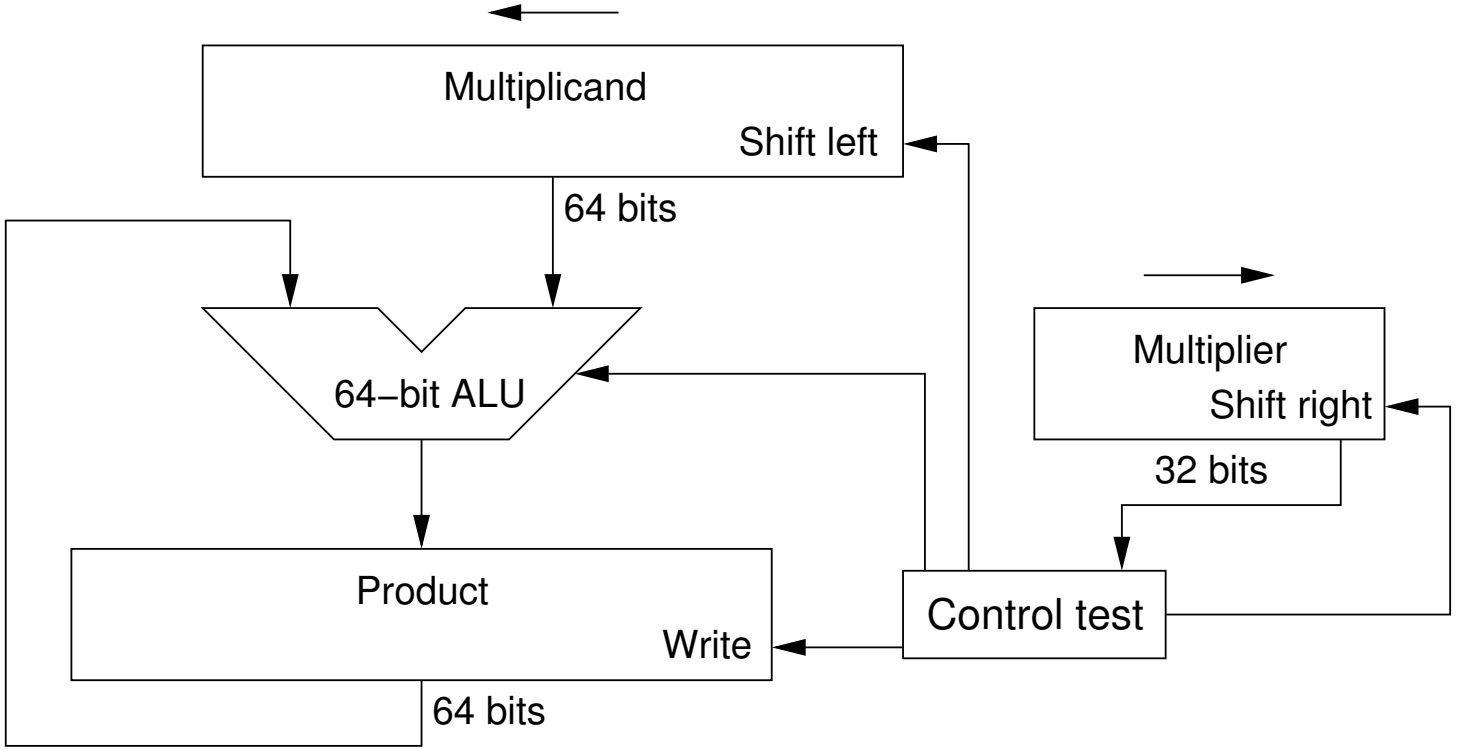
Method 1

- Sign extension to the result number of bit representation
- 32-bit representation, needs 64-bit result. Thus, extend to 64-bit representation
- Perform multiplication similar to paper and pencil method

Method 2

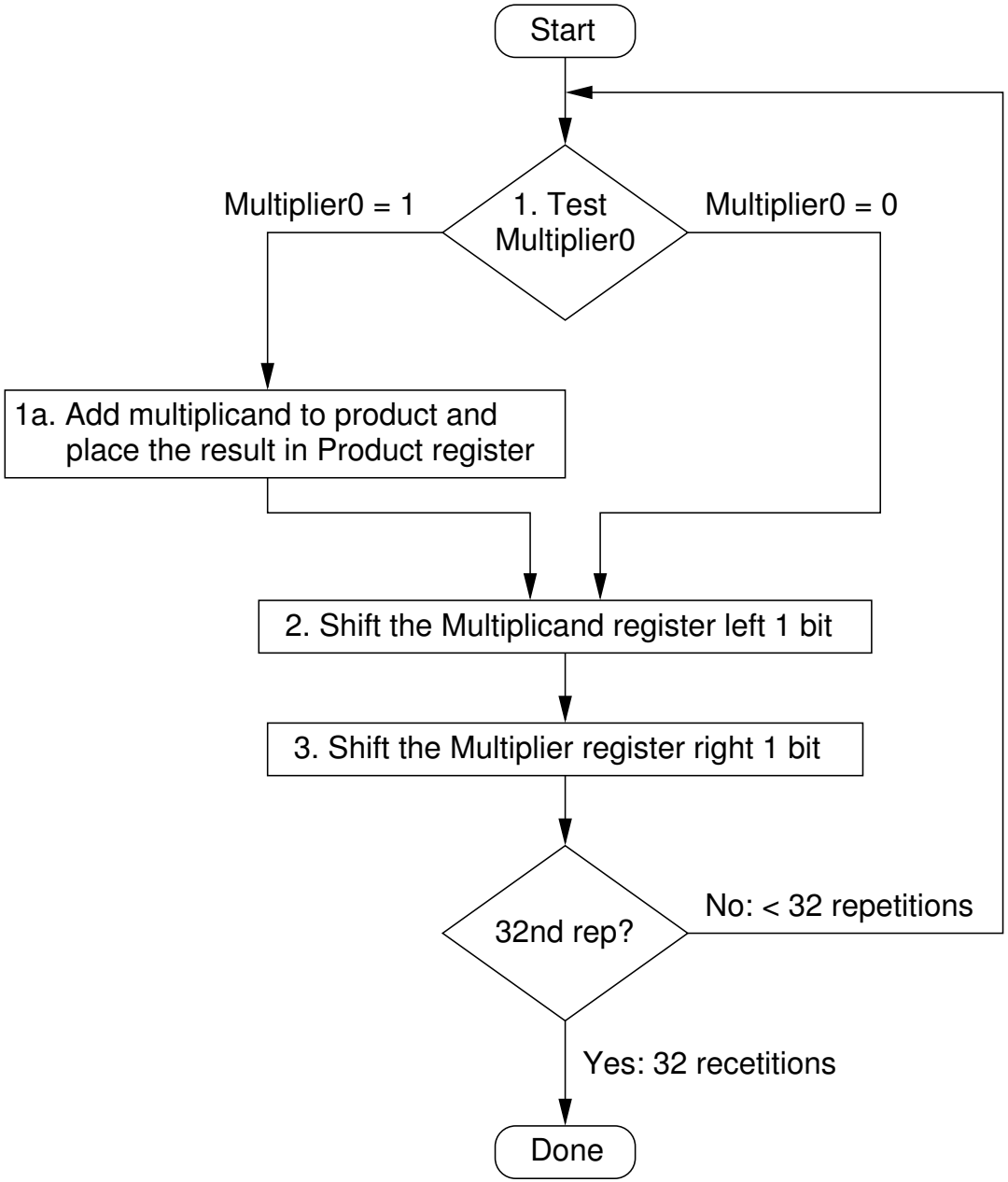
- Remember the signs of operands
- Convert operands to positive numbers if necessary
- Perform multiplication similar to paper and pencil method
- If signs are the same, the result is positive. Otherwise, negative

Sequential Version of Multiplication in Hardware



- Implemented from paper and pencil method
- Initialize 64-bit product register to 0
- Initialize 64-bit multiplicand register to 32-bit multiplicand in the right half and 0s in the left half

Sequential Version of Multiplication in Hardware



Sequential Version of Multiplication in Hardware

- Need almost 100 clock cycle to complete 32-bit multiplication
  - One for shifting multiplier
  - One for shifting multiplicand
  - One for adding multiplicand to the product

  Need 32 repetitions. Thus $32 \times 3 = 96$ clock cycles
- If done in parallel:
  - Shifting multiplier
  - Shifting multiplicand
  - Adding multiplicand to the product

  at the same time. Thus reduce the number of clock cycle by a third.

Example

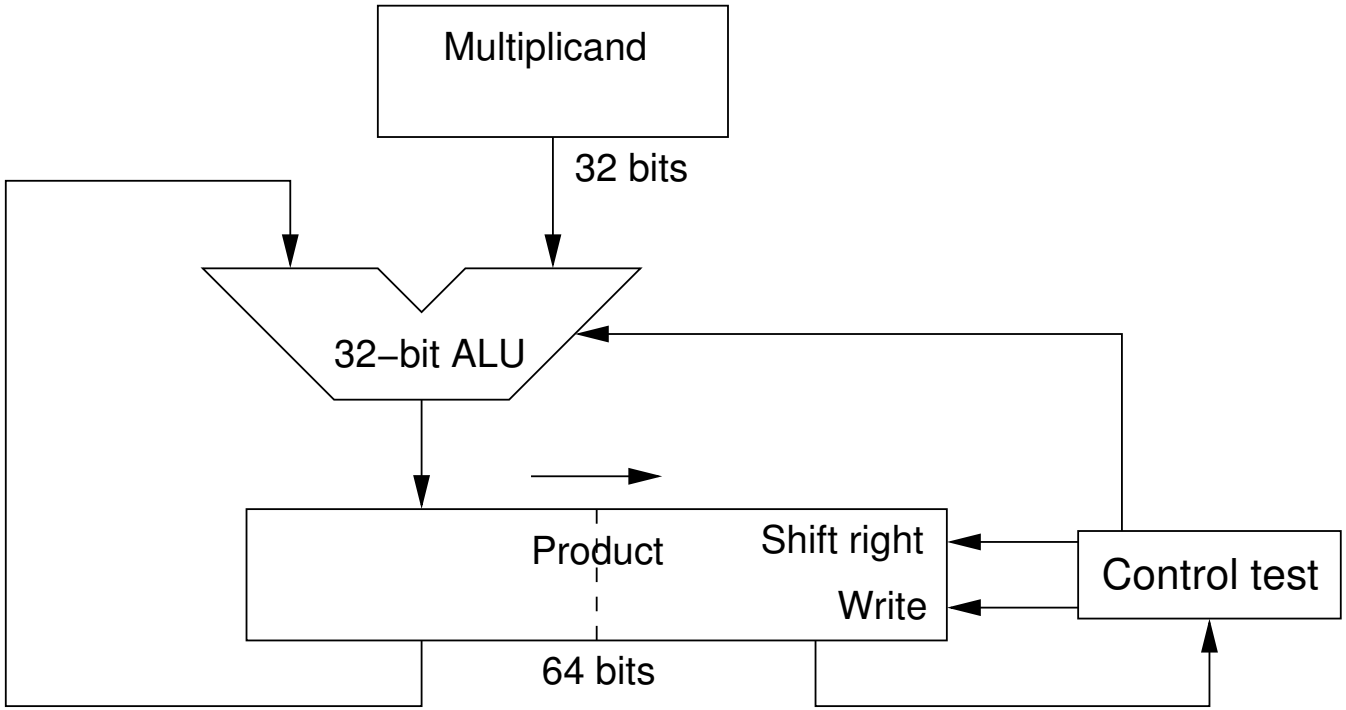
Using 4-bit representation, multiply 5_{10} by 6_{10} ($0101_2 \times 0110_2$)

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0110	0000 0101	0000 0000
1	1a: (0) thus Prod = Prod	0110	0000 0101	0000 0000
	2: Shift left Multiplicand	0110	0000 1010	0000 0000
	3: Shift right Multiplier	0011	0000 1010	0000 0000
2	1a: (1) thus Prod = Prod + Mcand	0011	0000 1010	0000 1010
	2: Shift left Multiplicand	0011	0001 0100	0000 1010
	3: Shift right Multiplier	0001	0001 0100	0000 1010
3	1a: (1) thus Prod = Prod + Mcand	0001	0001 0100	0001 1110
	2: Shift left Multiplicand	0001	0010 1000	0001 1110
	3: Shift right Multiplier	0000	0010 1000	0001 1110
4	1a: (0) thus Prod = Prod	0000	0010 1000	0001 1110
	2: Shift left Multiplicand	0000	0101 0000	0001 1110
	3: Shift right Multiplier	0000	0101 0000	0001 1110

The answer is $0001\ 1110_2 = 30_{10}$

Revised Hardware

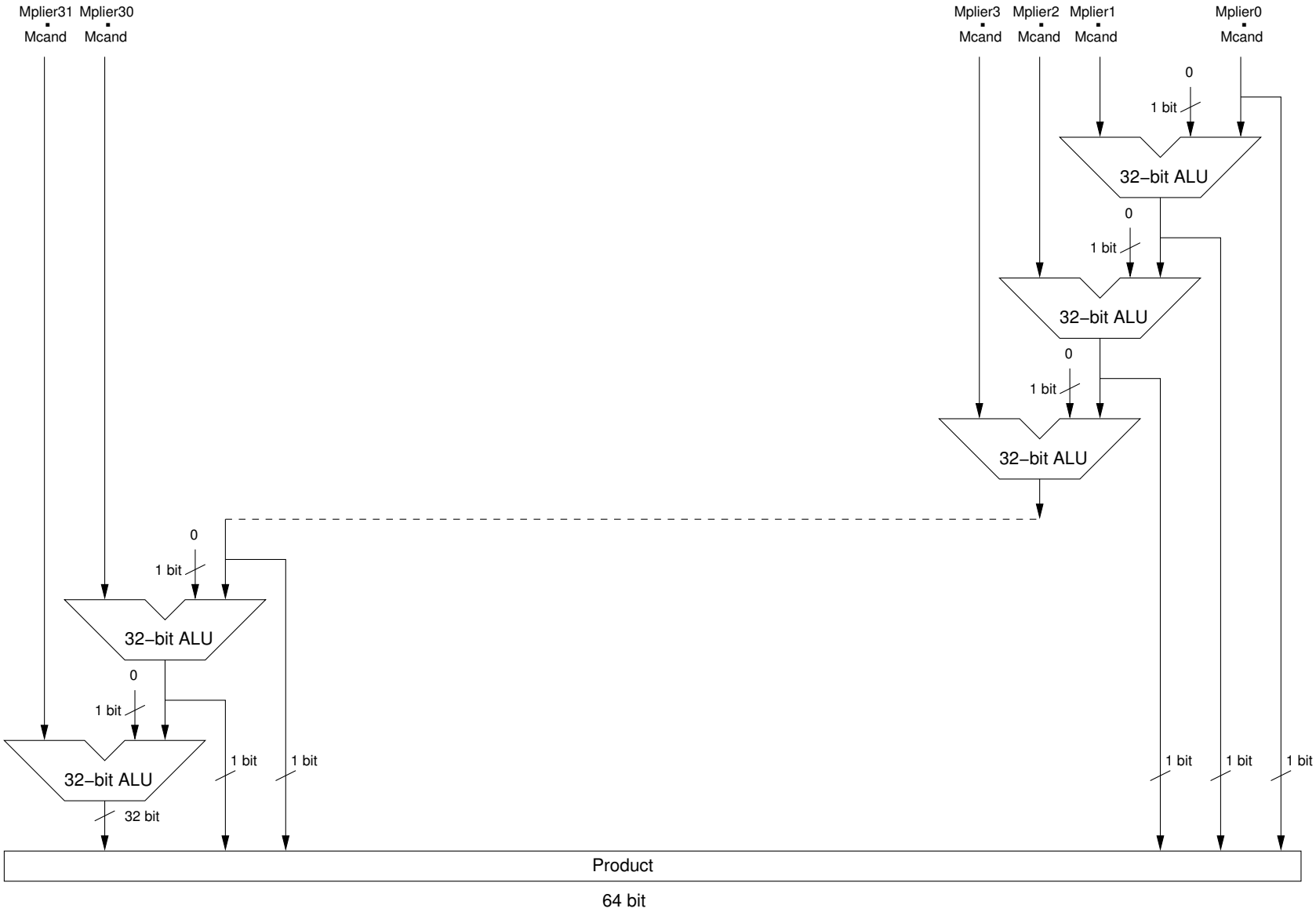
- Unused portion of product register can be used to store multiplier



- Initialize the right half of product register with multiplier
- Initialize the left half of product register to 0s

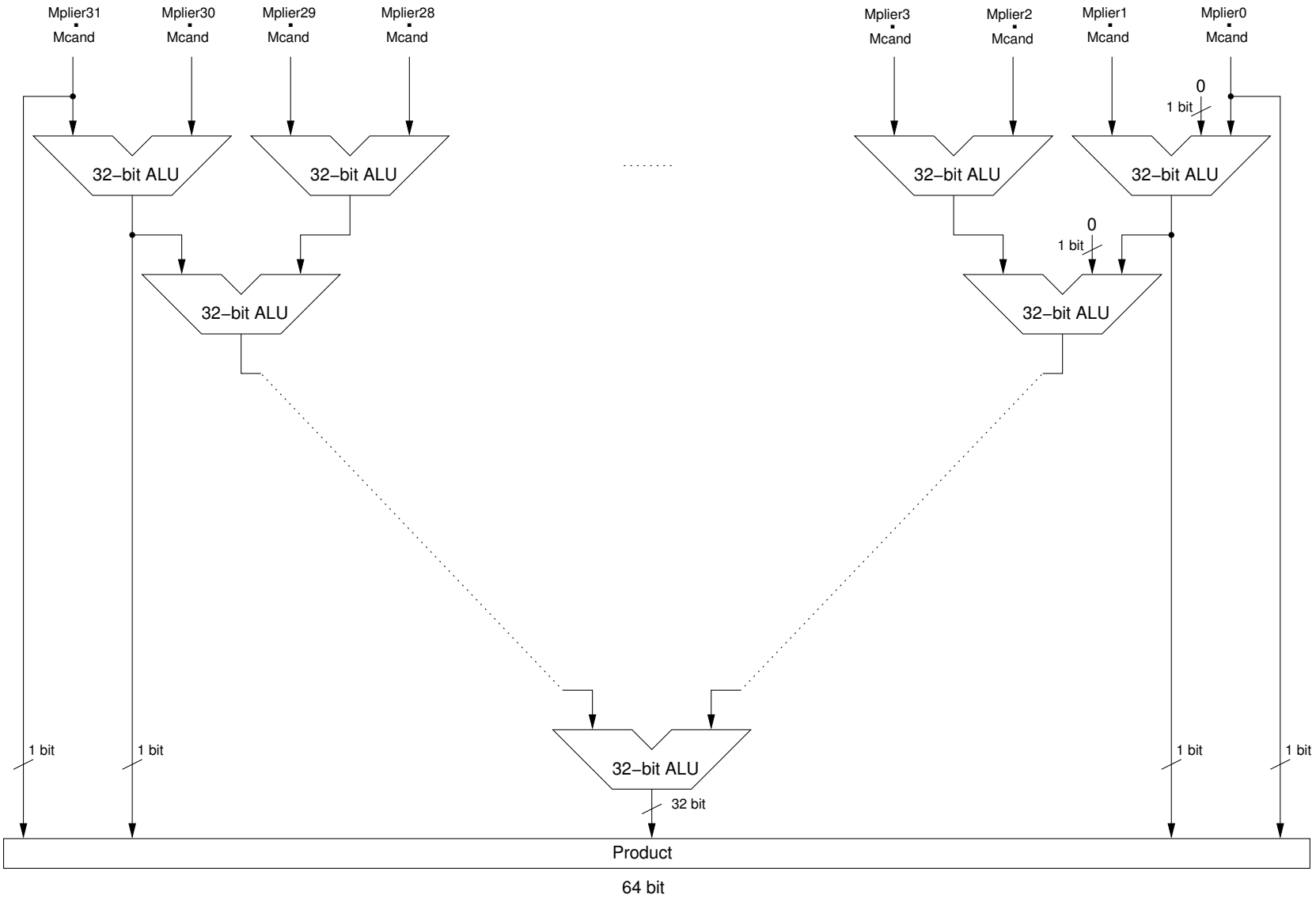
Revised Hardware

Use 31 32-bit wide ALU (31 add time)



Revised Hardware

Use 31 32-bit wide ALU (5 add time)



Multiplication in MIPS

- Recall that we need 64-bit product register
- MIPS uses two special purpose 32-bit product registers called `Hi` and `Lo` to contain 64-bit product
- Two instructions for multiplication
  - `mult` (Multiply)
  - `multu` (Multiply unsigned)
- Use the following instructions to move value from product registers to general purpose registers
  - `mflo` (Move From LOw)
  - `mfhi` (Move From HIgh)

Division

$$\frac{a}{b} = c \text{ with remainder d}$$

- a is called Dividend
- b is called Divisor
- c is called Quotient (result)
- d is called Remainder
- The following equation must hold:

$$\text{Dividend} = (\text{Quotient} \times \text{Divisor}) + \text{Remainder}$$

Division

- Use paper and pencil method to calculate $234 \div 5$

$$
\begin{array}{r}
 & 0 & 4 & 6 \\
\cline{2-4}
5 & 2 & 3 & 4 \\
 & 2 & 0 & \\
\cline{2-3}
 & & 3 & 4 \\
 & & 3 & 0 \\
\cline{3-4}
 & & & 4 \\
\end{array}
$$

- The quotient is 46 and the remainder is 4.
- Let's examine each step closely on the board.

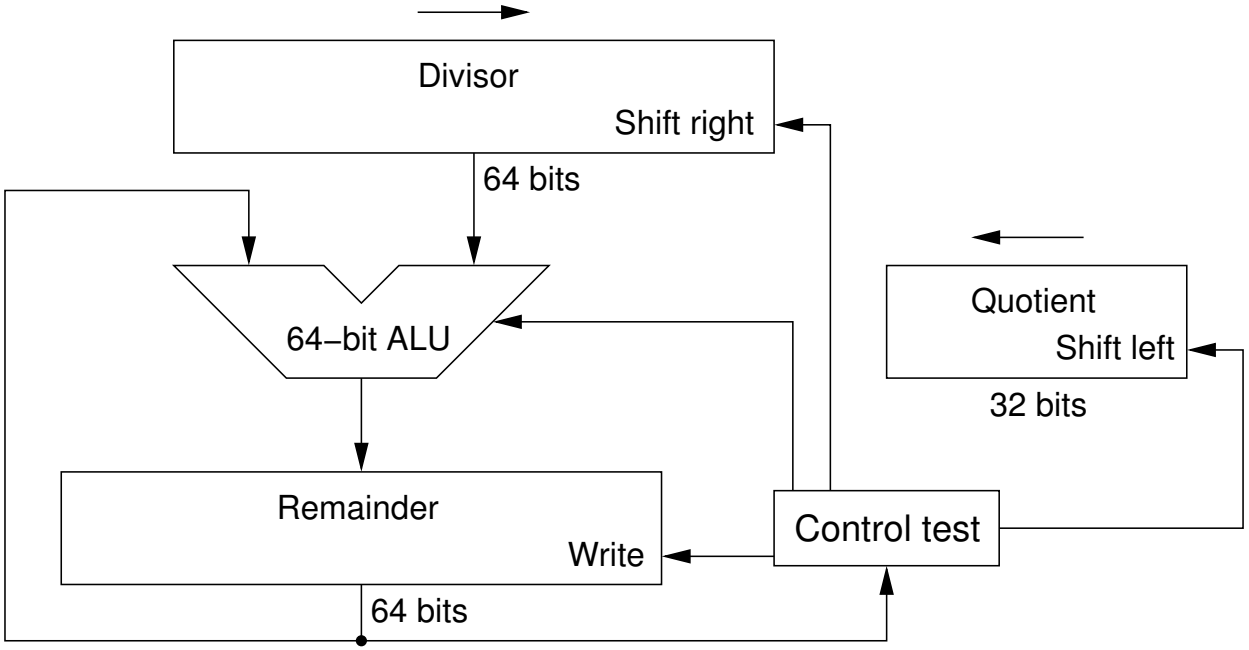
Division

- Example: Use paper and pencil method for $7_{10}/2_{10}$ (4-bit representation)
- Basic Algorithm
  1. Extend Dividend to 8-bit (padded upper four bits with 0s)
  2. Extend Divisor to 8-bit (padded lower four bits with 0s)
  3. Try to subtract Dividend by Divisor
     - If Dividend is larger, make

       $$\text{Dividend} = \text{Dividend} - \text{Divisor}$$

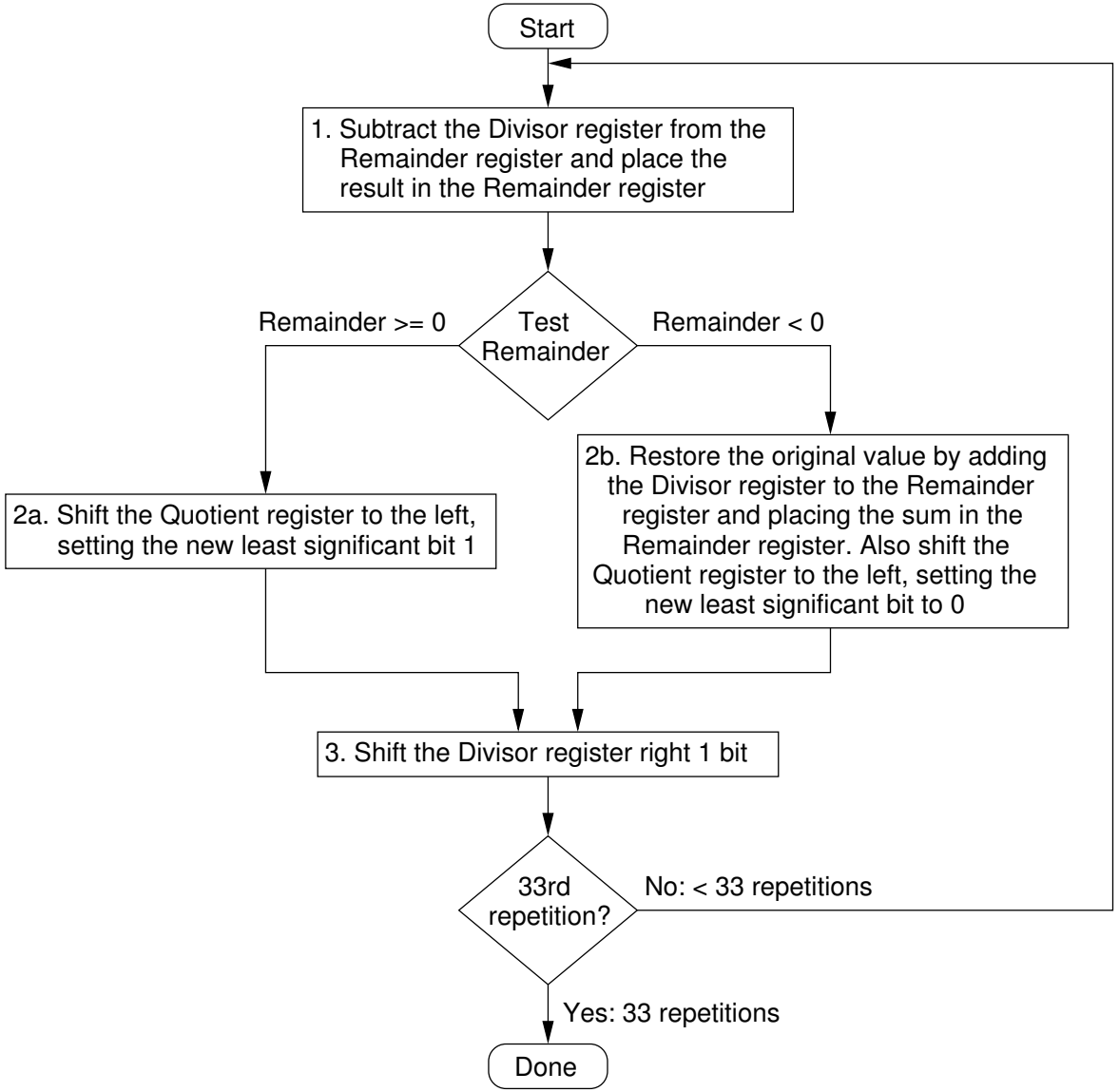
       and put 1 into Quotient
     - If Dividend is smaller, do not subtract, and put 0 into Quotient
  4. Shift Divisor to the right one bit and go back to step 3 until Divisor is 0

A Division Algorithm and Hardware



- Initialize 32-bit Quotient register to 0
- Initialize top 32-bit Divisor register to Divisor and bottom 32-bit to 0s
- Initialize top 32-bit Remainder register to 0s and bottom 32-bit to Dividend.

A Division Algorithm and Hardware



Example

Dividing 7_{10} by 3_{10} ($0111_2/0011_2$)

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0011 0000	0000 0111
1	1: Rem = Rem - Div	0000	0011 0000	1101 0111
	2b: Rem $<$ 0, +Div, sll Q, Q0 = 0	0000	0011 0000	0000 0111
	3: Shift Div right	0000	0001 1000	0000 0111
2	1: Rem = Rem - Div	0000	0001 1000	1111 0111
	2b: Rem $<$ 0, +Div, sll Q, Q0 = 0	0000	0001 1000	0000 0111
	3: Shift Div right	0000	0000 1100	0000 0111
3	1: Rem = Rem - Div	0000	0000 1100	1111 1011
	2b: Rem $<$ 0, +Div, sll Q, Q0 = 0	0000	0000 1100	0000 0111
	3: Shift Div right	0000	0000 0110	0000 0111
4	1: Rem = Rem - Div	0000	0000 0110	0000 0001
	2a: Rem $\geq$ 0, sll Q, Q0 = 1	0001	0000 0110	0000 0001
	3: Shift Div right	0001	0000 0011	0000 0001
5	1: Rem = Rem - Div	0001	0000 0011	1111 1101
	2a: Rem $<$ 0, +Div, sll Q, Q0 = 0	0010	0000 0011	0000 0001
	3: Shift Div right	0010	0000 0001	0000 0001

The answer is $0010_2 = 2_{10}$ and the remainder is $0001_2 = 1_{10}$

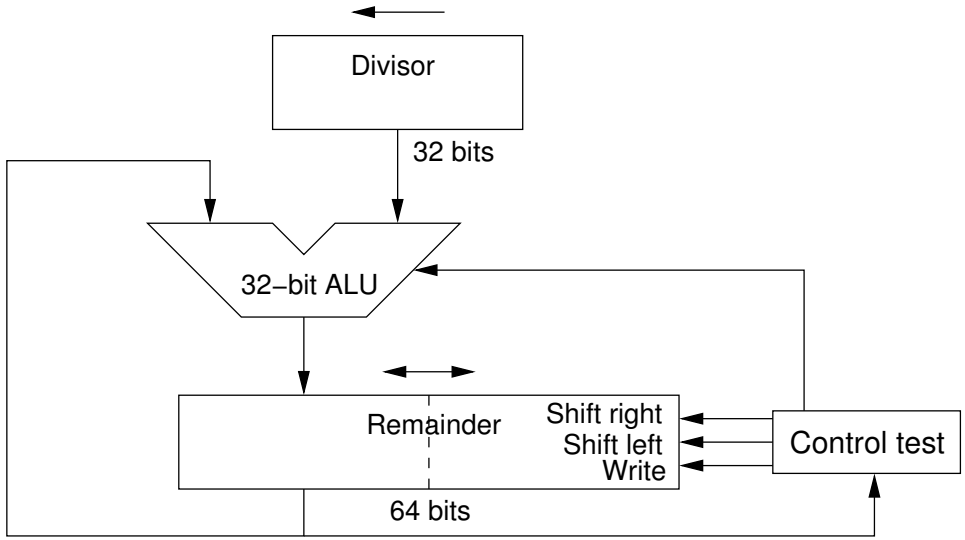
Example

How about $4_{10}/2_{10}$ ($0100_2/0010_2$)?

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

The answer is $0010_2 = 2_{10}$ and the remainder is $0000_2 = 0_{10}$

Revised Hardware



- Initialize the right half of the Remainder register with Dividend
- Initialize the left half of the Remainder to 0s
- Quotient is insert to the least significant bit when remainder is be shift left
- At the end:
  - The Remainder is in the left half of the Remainder register
  - The Quotient is in the right half of the Remainder register

Signed Division

- What is answer of -7/2?
  - Quotient = -3 and Remainder = -1 or
  - Quotient = -4 and Remainder = +1?
- Both answers satisfy

$$\text{Dividend} = (\text{Quotient} \times \text{Divisor}) + \text{Remainder}$$

- Rules:
  1. The dividend and remainder must have the same signs
  2. Must satisfy the following equation:

$$\text{Dividend} = (\text{Quotient} \times \text{Divisor}) + \text{Remainder}$$

Example

For signed division:

1. Perform division as positive operands
2. If signs of operands are opposite, negates the quotient
3. Make the sign of remainder the same as dividend.

Example: $-7/2$, calculated as $7/2$ and from our example the answer are

- Quotient $= 3$ and
- Remainder $= 1$
- Since signs of operand are opposite, Quotient should be -3
- Since the sign of dividend is minus, Remainder should be -1

Division in MIPS

- Use two special purpose registers `Hi` and `Lo` for 64-bit remainder register
- Supply the following instructions:
  - `div`: Division (signed)
  - `divu`: Division (unsigned)
- Results:
  - Register `Hi` contains the remainder and
  - Register `Lo` contain the quotient
- Use `mflo` and `mfhi` to transfer answers to general purpose registers

Floating Point

- Need support for numbers with fractions (real numbers)
- Example:
  - π: 3.14159265...
  - π: $\frac{22}{7} \approx 3.1428571\ldots$
  - Speed of light: 299,792.458 km/s
  - $e = 1 + \frac{1}{1} + \frac{1}{1\times2} + \frac{1}{1\times2\times3} + \cdots \approx 2.71828\ldots$
  - $\sqrt{2} \approx 1.41421356\ldots$

Scientific Notation

- Examples:
  - 2.4 GHz: 2.4×10^9 Hz or 24×10^8 Hz
  - 5.5 ns: 5.5×10^-9 sec or 0.55×10^{-10} sec
- **Normalized** if single nonzero digit to the left of the decimal point
- Examples:
  - π: 3.14159265×10^0 but not 0.314159267×10^1
  - Speed of light: 2.99792458×10^5 km/s but not 29.9792458×10^4
- In the case of speed of light
  - 2.99792458 is called fraction
  - 5 in 10^5 is called exponent

Binary Point

- We can have binary point similar to decimal point
- Example:
  - 1101.0011_2
  - 0.1111_2
  - 1.1001_2
- What is the decimal value of 1101.0011_2 (unsigned)

$$(1\times2^3)+(1\times2^2)+(0\times2^1)+(1\times2^0)+(0\times2^{-1})+(0\times2^{-2})+(1\times2^{-3})+(1\times2^{-4})$$

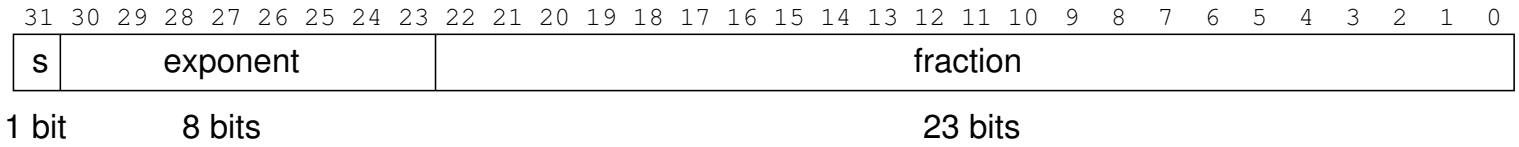
which is

$$8+4+0+1+0+0+\frac{1}{8}+\frac{1}{16}=13.1875$$

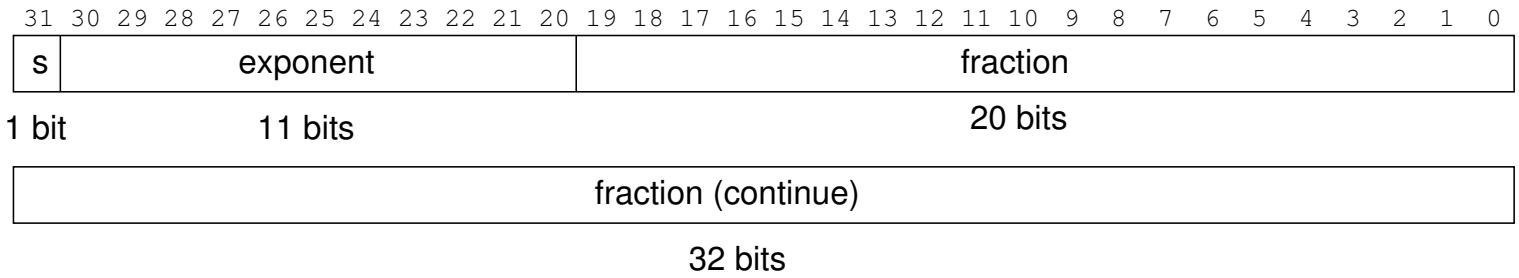
- We can also have scientific notation for binary
- Example:
  - 1101.0011_2 can be written as $1.1010011_2\times2^3$
  - 0.1111_2 can be written as $1.111_2\times2^{-1}$

MIPS Floating-point Number

Single precision (e.g., `float` in C)



Double precision (e.g., `double` in C)



The value is

$$(-1)^s \times F \times 2^E$$

where

- F is the value in the fraction field and
- E is the value is the exponent field.

Overflow and Underflow in Floating-Point

- **Overflow**: the exponent is too large to be represented in the exponent field
- **Underflow**: the negative exponent is too large to fit in the exponent field
- For single precision:
  - The smallest exponent is 2^{-126} (not -127)
  - The largest exponent is 2^{127} (not 128)
- We cannot represent the following numbers:
  - $1.0_2 \times 2^{-127}$
  - $1.0_2 \times 2^{128}$

- Make leading 1 bit implicit since it is always 1 (normalized)
  - 1.0101×2^{-2}
  - 1.0000×2^{5}
  - 1.1110×2^{-1}
  - 0.1011×2^{2} (not normalize). Should be 1.0110×2^{1}
- Thus the value is $(-1)^s \times (1 + F) \times 2^E$
- **fraction** means 23- or 52-bit number
- **significand** means 24- or 53-bit number (1 + fraction)
- Let $s1$, $s2$, ..., $s23$ correspond to the fraction from left to right. The value is

$$(-1)^s \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + \cdots + (s23 \times 2^{-23})) \times 2^E$$

Special Values in IEEE 754

IEEE 754 reserves some representations for special values

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	$\pm$ denormalized number
1-254	Anything	1-2046	Anything	$\pm$ floating-point number
255	0	2047	0	$\pm$ infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Examples:

- Nonzero positive number divided by zero is $+\infty$
- Nonzero negative number divided by zero is $-\infty$
- 0/0 is NaN

- Advantages
  - Sign bit can be used for quick test using integer comparison (less than, greater than, or equal to 0)
  - Exponent can also be used for quick test using integer comparison
- Instead of using two's complement in exponent, IEEE 754 use **bias**
  - $00\ldots00_2$: the most negative
  - $11\ldots11_2$: the most positive
  - The bias is 127 for single precision and 1023 for double precision
  - The the actual value is

$$(-1)^s \times (1 + F) \times 2^{(\text{E - Bias})}$$

Examples

1. Show the IEEE 754 binary representation of the number -0.75_{10}

   1 01111110 10000000000000000000000

2. Show the IEEE 754 binary representation of the number 22.5_{10}

   0 10000011 01101000000000000000000

3. Show the IEEE 754 binary representation of the value of π where $\pi = 3.14159265$.

   0 10000000 100100100...

Example

What is the decimal number represented by

$$1\ 10001000\ 01010100000000000000000$$

The exponent is

$$1000\ 1000_2 - 127_{10} = 136_{10} - 127_{10} = 9_{10}$$

Since the fraction is 0.010101 and the sign bit is 1, the value is

$$\begin{aligned}(-1)^1 \times (1 + 0.010101_2) \times 2^9 &= -1 \times (1.010101_2 \times 2^9) \\ &= -1 \times (10\ 1010\ 1000_2 \times 2^0) \\ &= -1 \times (512 + 128 + 32 + 8) \\ &= -680\end{aligned}$$