Arithmetic (Ch: 3)

Dr. Rajib Ranjan Maiti

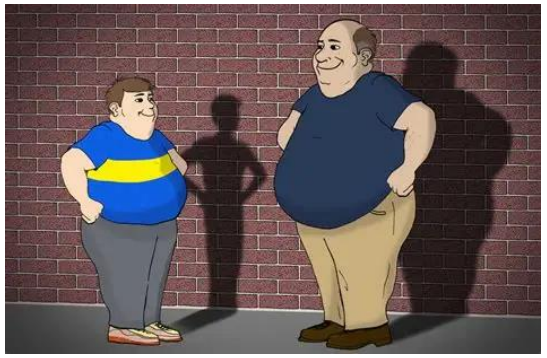
CSIS, BITS-Pilani, Hyderabad

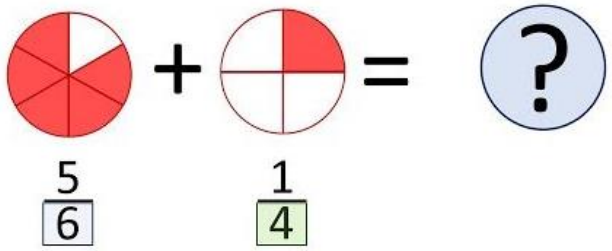
MIPS Instructions for Multiplication and Division

Instruction	Example	Meaning	Comments
multiply	mul $s1, $s2, $s3	Hi, {Lo,$s1} ← $s2 * $s3	Should be used for 32 bits of product
	mult $s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
Divide	div $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
divide unsigned	divu $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo
		Pseudoinstructions	
Multiply	mul $t0, $t2, -100	$t0 ← $t2 * -100	16-bit or 32-bit signed immediate
	mulo $t0, $t1, -100	$t0 ← $t2 * -100	With overflow
	mulo $t0, $t1, $t2	$t0 ← $t1 * $t2	With overflow
	mulou $t0, $t1, $t2	$t0 ← $t1 * $t2	With overflow unsigned
Remainder	rem $t0, $t1, 100	$t0 ← $t1 mod 100	16-bit or 32-bit signed immediate
	rem $t0, $t1, $t2	$t0 ← $t1 mod $t2	

Operations beyond simple integers

- The other numbers that commonly occur includes:
- ■What happens if an operation creates a number bigger than can be represented?
- ■What about fractions and other real numbers?
- ■ How does hardware really multiply or divide numbers?





Addition and Subtraction

- Basic idea
  - $c - a = c + (-a)$

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
- & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}
\end{array}
$$

or via addition using the two's complement representation of -6:

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
+ & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}
\end{array}
$$

Addition and Subtraction

- Overflow
  - occurs when the result from an operation cannot be represented with the available hardware
  - In MIPS, registers are 32-bit long

Overflow conditions for addition and subtraction.

Operation	Operand A	Operand B	Result indicating overflow	Example (4 bit signed)
A + B	≥ 0	≥ 0	< 0	0111 + 0111 = 1110
A + B	< 0	< 0	≥ 0	1010 + 1010 = 0100
A - B	≥ 0	< 0	< 0	
A – B	< 0	≥ 0	≥ 0	

Overflow detection in MIPS

- MIPS detects
  - overflow with an **exception**, also called an **interrupt** on many computers

- MIPS includes
  - A register called the ***exception program counter*** (**EPC**) in ***coprocessor 0*** to contain the address of the instruction that caused the exception
  - Instruction: ***move from coprocessor 0*** (**mfc0**) used to copy **EPC** into a general-purpose register
    - so that MIPS software has the option of returning to the offending instruction via a jump register instruction
  - For example,
    - **mfc0 $t0, $14** #register 14 in c0 is EPC

Coprocessor 1 (c1) implements floating point operations

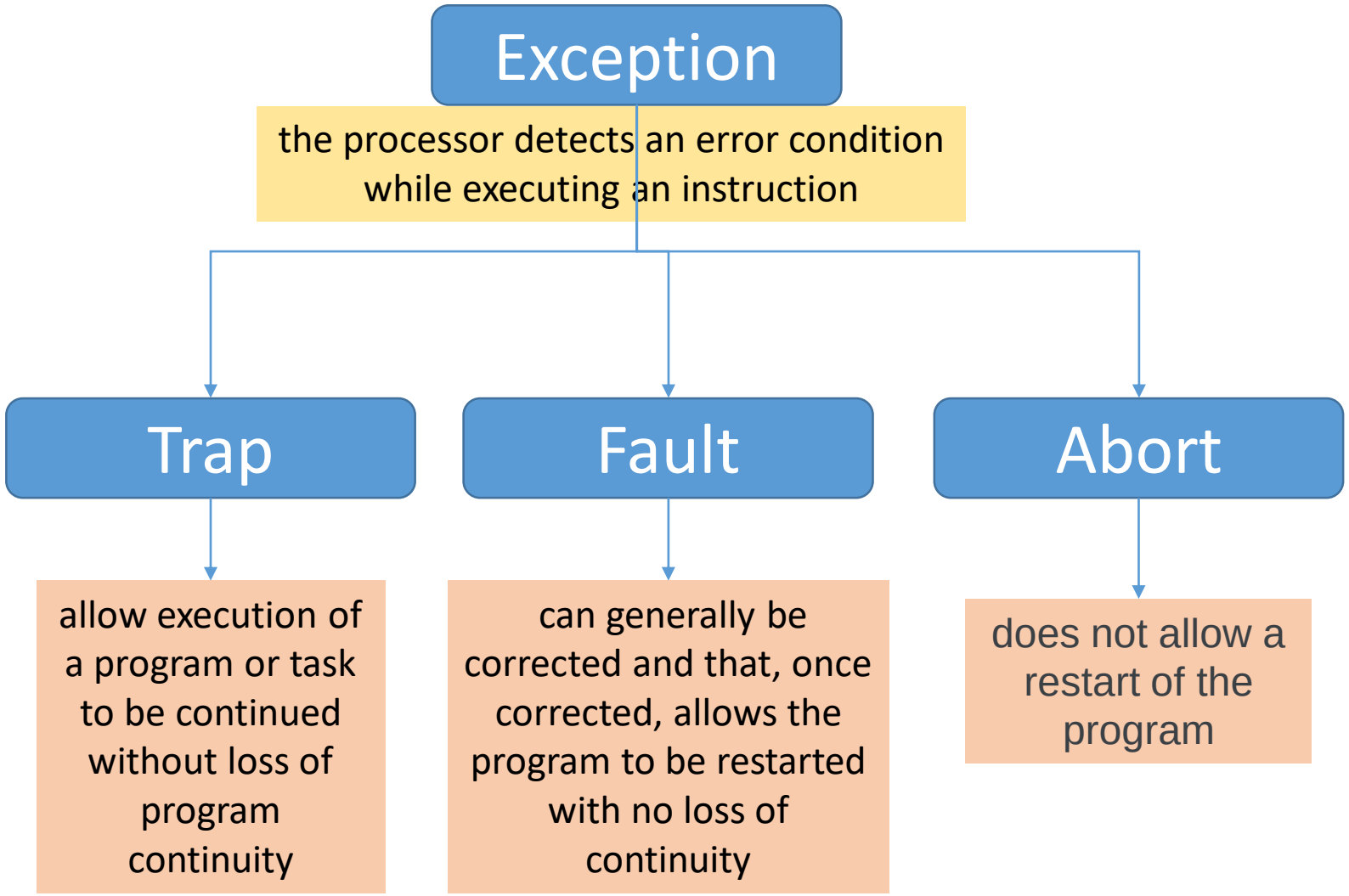
- ***coprocessor 0*** *has four usable registers*

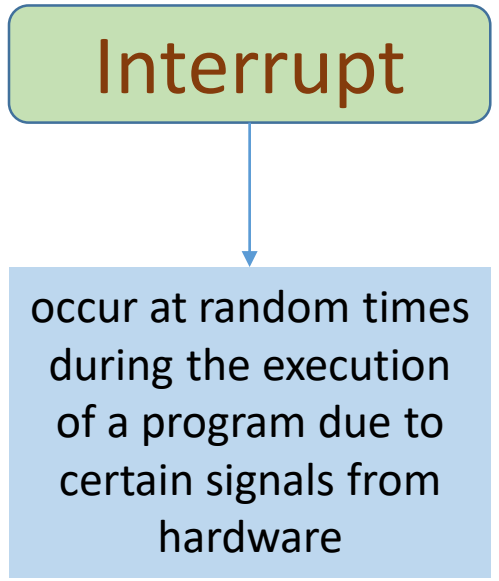
Reg#	Meaning	Comment
8	BadVAddr	Memory address where exception occurred
12	Status	Interrupt mask, enable bits, and status when exception occurred
13	Cause	Type of exception and pending interrupt bits
14	EPC	Address of instruction that caused exception

Instructions causing exception

- Instructions that cause exception on overflow
  - add (add),
  - add immediate (addi), and
  - subtract (sub)
- Instructions that do not cause exception on overflow
  - Add unsigned (addu),
  - add immediate unsigned (addiu), and
  - subtract unsigned (subu)

Trap, Exception and Interrupt





Exception handling using c0

- When an exception happens,
  - the **control is transferred to a different program**, called exception handler
  - In MIPS, exception handler is located at **0x80000080**
    - written explicitly for the purpose of dealing with exceptions.

- After the execution of exception handler,
  - the control is returned to the program that was executing when the exception occurred
    - that program then continues as if nothing happened

Exception handling using c0

- An exception handler appears as a procedure
  - with no parameters and no return value
  - inserted in the program by the assembler

- the exception handler must
  - save any registers it may modify, and
  - then restore them before returning control to the interrupted program
  - **The registers $26 and $27 (i.e., $k0 and $k1) are reserved to be used by the interrupt handler**

Exception handling using c0

- User programs run in **user** mode of CPU.
  - The CPU enters the **kernel** mode when an exception happens.
  - Thus, Coprocessor 0 can only be used in **kernel** mode.
- The whole upper half of the memory space is reserved for the kernel mode:
  - it can not be accessed in user mode

- Instructions which access the registers of coprocessor 0
  - mfc0 Rdest, C0src #Move from coprocessor's register (C0src) to Rdest
  - mtc0 Rsrc, C0dest #Move from register (Rsrc) to C0dest
  - lwc0 C0dest, address #Load word from address to C0dest
  - swc0 C0src, address #Store the content of C0src to address in memory

Byte or halfword arithmetic

- Which are MIPS instructions for byte and halfword operations?
  - 1. **Load:** lbu, lhu; **arithmetic:** add, sub, mult, div; **store:** sb, sh
  - 2. **Load:** lb, lh; **arithmetic:** add, sub, mult, div; **store:** sb, sh
  - 3. **Load:** lb, lh; **arithmetic:** add, sub, mult, div, using AND to mask result to 8 or 16 bits after each operation; **store:** sb, sh

- Some examples
  - A sequence of MIPS instructions that can **discover overflow**

```
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 # if $t3 ≠ 0, i.e., sign does not differ,
                            #then 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 ≠; goto overflow
```

Multiplication And Division

Multiplication

- the length of the product of an *n*-bit multiplicand and an *m*-bit multiplier is *n + m* bits long
  - Multiplicand 1000_{ten}
  - Multiplier x 1001_{ten}
  - 1000
  - 0000 .
  - 0000 . .
  - 1000 . . .
  - Product 1001000_{ten}
- With only two choices, each step of the multiplication is simple:
  - 1. Just place a copy of the **multiplicand** (i.e., 1 x multiplicand) in the **proper place** if the **multiplier digit is a 1**, or
  - 2. Place 0 (i.e., 0 x multiplicand) in the **proper place** if the digit is 0.

Multiplication in MIPS

- MIPS provides
  - a separate pair of 32-bit registers, ***Hi* and *Lo,*** to hold 64-bit product

- MIPS has two instructions:
  - multiply (mult) and multiply unsigned (multu)
  - **mult $t2, $t3** **#[hi, lo] <- $t2 * $t3**
  - **multu $t0, $t1** **# [hi, lo] <- $t2 * $t3**

- To fetch the integer 32-bit product
  - **mflo $t1**
  - **mfhi $t0**

Multiplication in MIPS

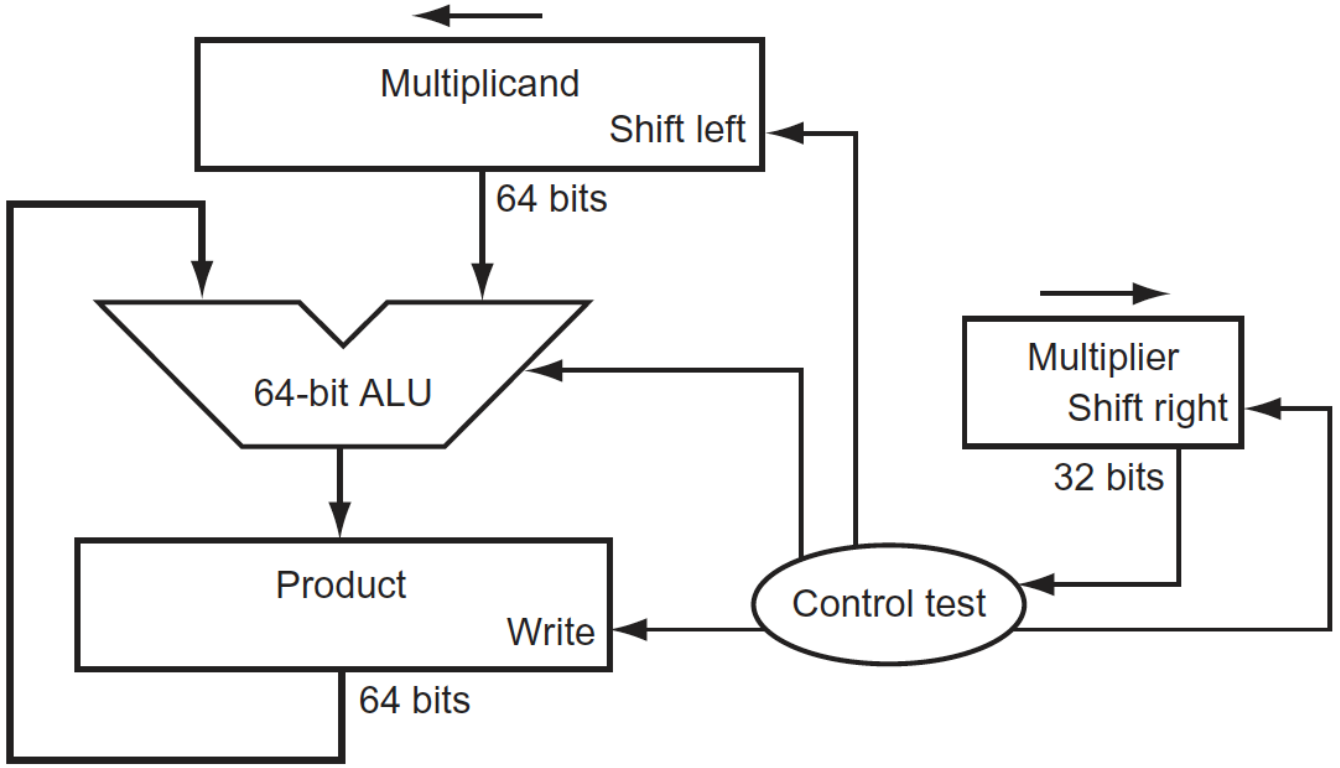
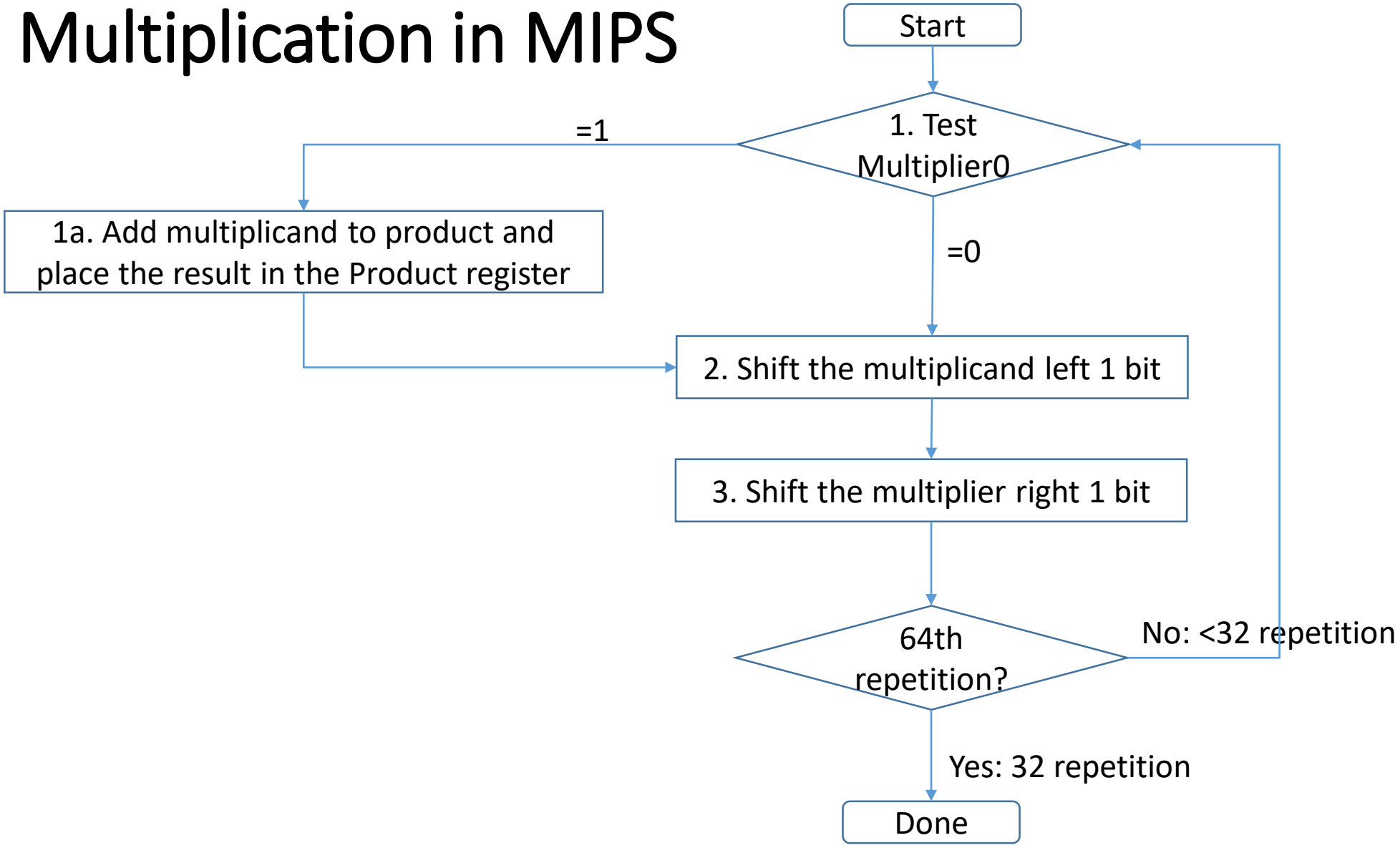


FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix B describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

Multiplication in MIPS



Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Improved Multiplication Circuit

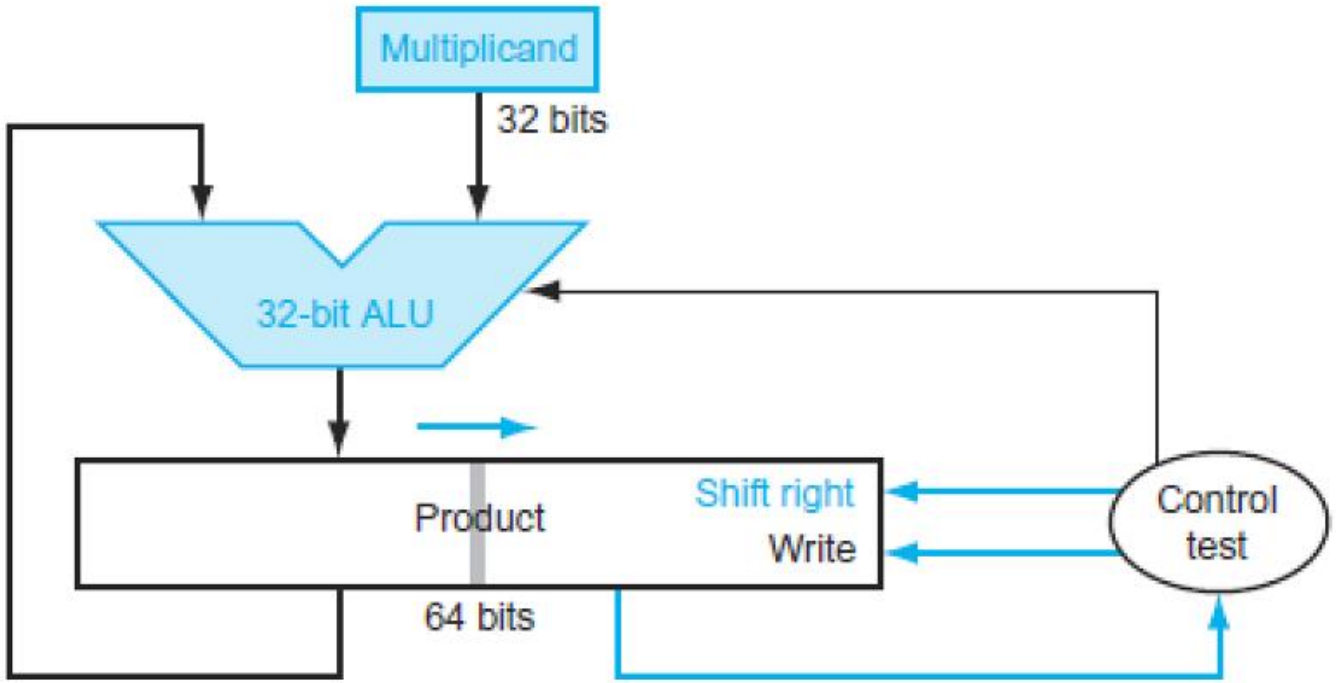


FIGURE 3.5 Refined version of the multiplication hardware. Compare with the first version in Figure 3.3. The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from Figure 3.3.)

Division

- 1001_{ten} Quotient
- Divisor 1000_{ten} | 1001010_{ten} Dividend
- -1000
- 10
- 101
- 1010
- -1000
- 10_{ten} Remainder

Division

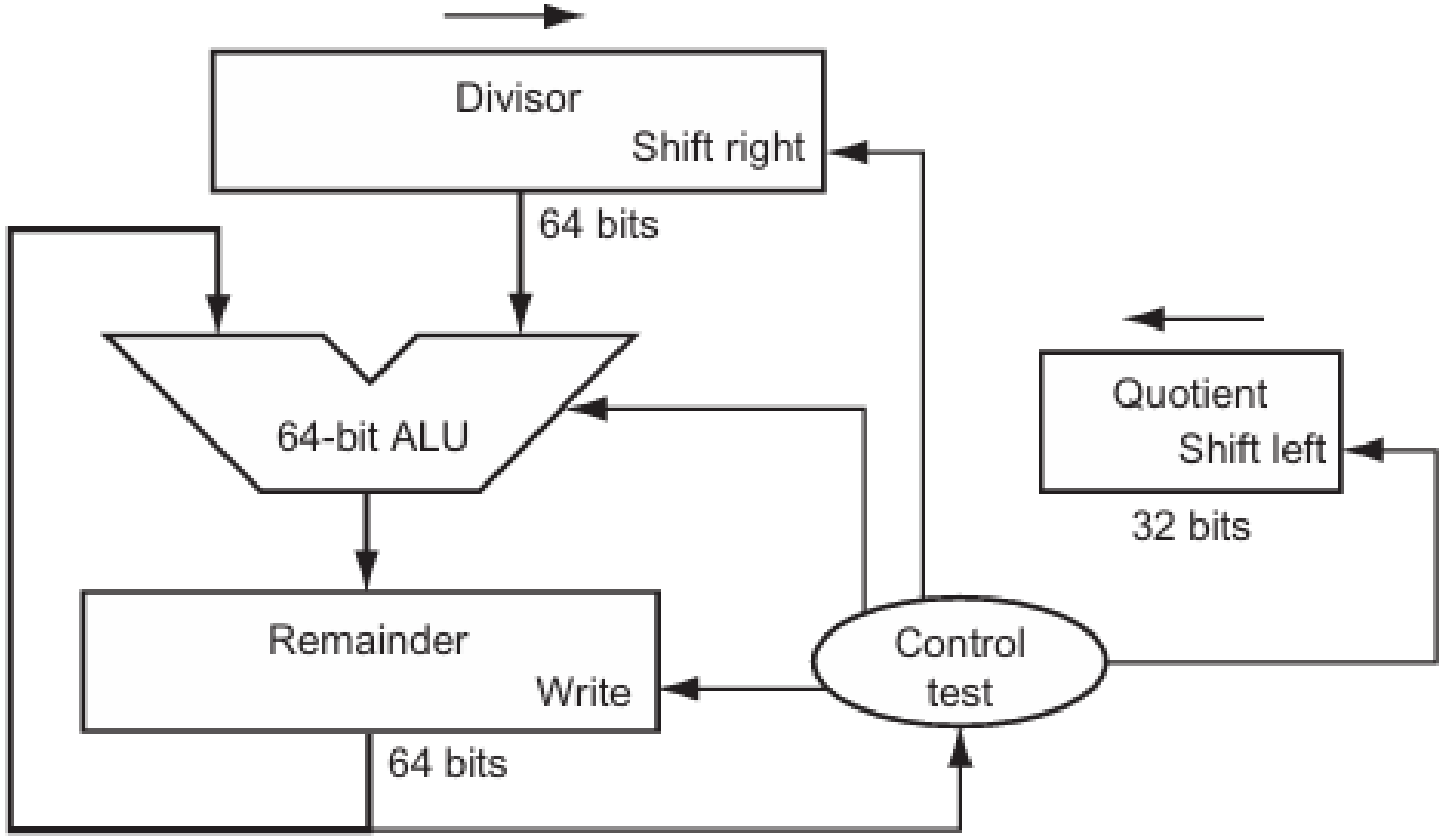
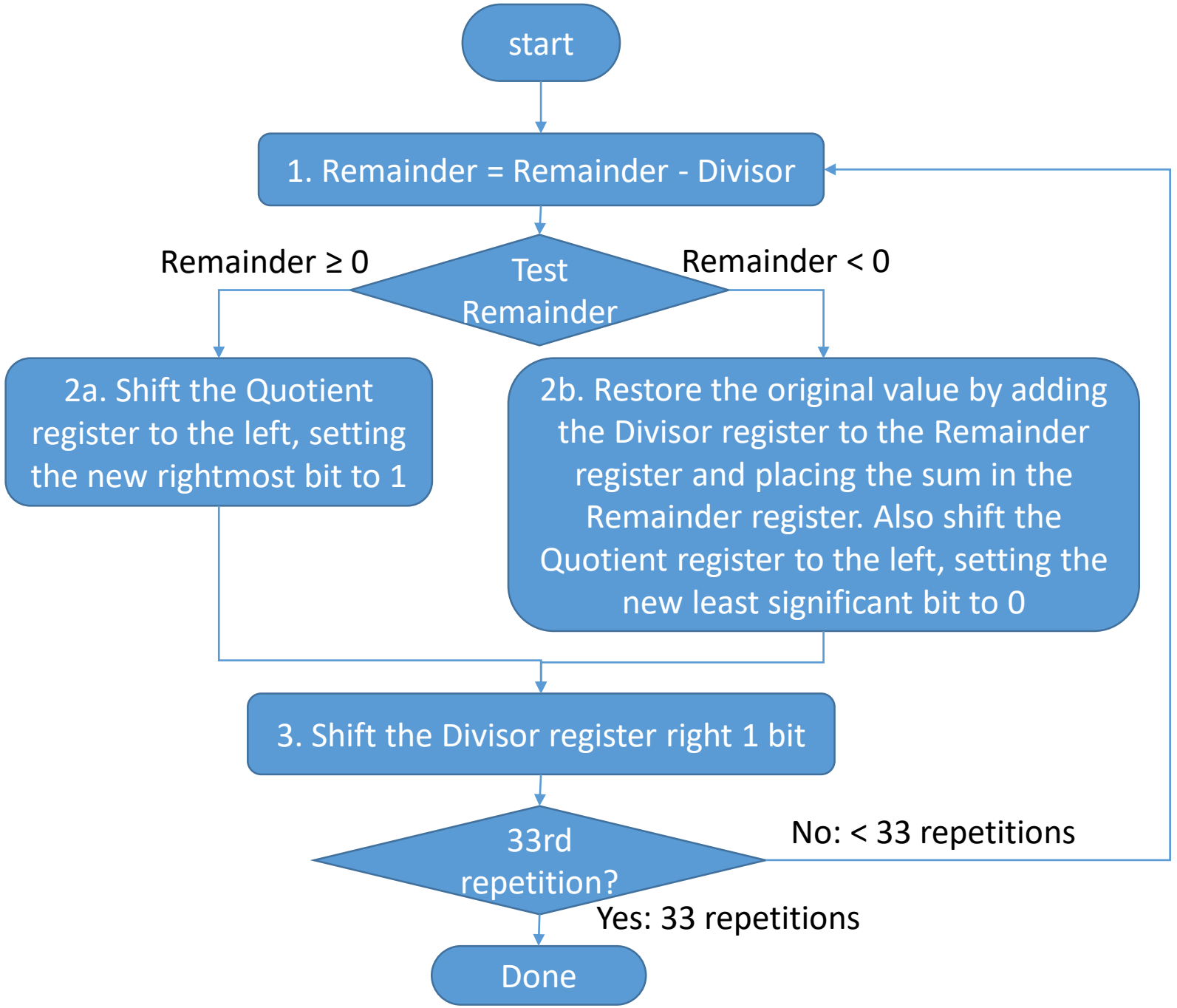


FIGURE 3.8 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit each iteration. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

Division

- Each iteration of the algorithm
  - move the divisor to the right one digit,
- We start with
  - the divisor placed in the left half of the 64-bit Divisor register
  - shift it right 1 bit each step to align it with the dividend
- The Remainder register is initialized with the dividend



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	①111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	⓪000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	⓪000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

A better hardware for divide

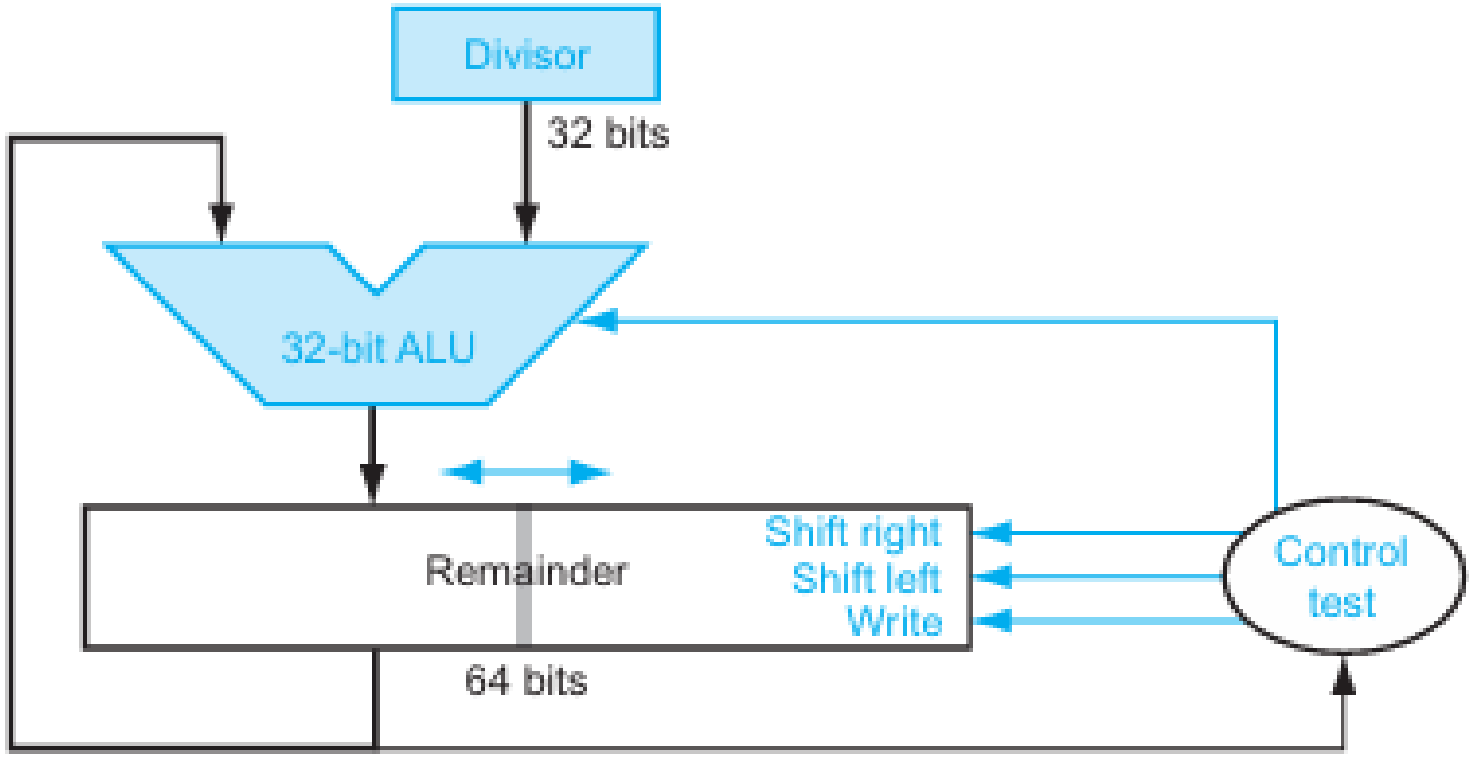


FIGURE 3.11 An improved version of the division hardware. The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to Figure 3.8, the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. (As in Figure 3.5, the Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.)

Division

- The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:
  - Dividend = Quotient X Divisor + Remainder
- Lets consider the cases of dividing all the combinations of $\pm 7_{ten}$ by $\pm 2_{ten}$
- The first case:
  - +7 ÷ +2 ➔ Quotient = +3, Remainder = +1
  - Checking the results:
    - +7 = 3 x 2 + (+1) = 6 + 1
- The second case:
  - -7 ÷ +2 ➔ Quotient = -3
  - Checking the result:
    - Remainder = (Dividend X Quotient - Divisor)
    - = -7 - (-3 *x* +2)
    - = -7 - (-6) = -1
- So, -7 ÷ +2 ➔ Quotient = -3, Remainder = -1
  - Checking the results again:
    - -7 = -3 x 2 + (-1) = -6 -1
- We calculate the other combinations by following the same rule:
  - -7 ÷ -2 ➔ Quotient = 3, Remainder = -1
  - +7 ÷ -2 ➔ Quotient = -3, Remainder = 1

Division

- To handle both signed integers and unsigned integers,
  - MIPS has two instructions:
  - *divide* (div) and *divide unsigned* (divu).

- The MIPS assembler allows divide instructions to specify three registers,
  - generating the mflo or mfhi instructions to place the desired result into a general-purpose register

- Example MIPS instructions
  - **div $s2,$s3 #Lo = $s2 / $s3, Hi = $s2 mod $s3**
  - **divu $s2,$s3 #Lo = $s2 / $s3, Hi = $s2 mod $s3**


Thank You