Chapter 3
Arithmetic for Computers

- **Operations on integers** –
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- **Floating-point real numbers** –
  Representation and operations

Integer Addition:

Example: 7 + 6

```
.....0 0 0 0 1 1 1
.....0 0 0 0 1 1 0
-------------------
.....0 0 0 1 1 0 1
```

Overflow happens the result goes beyond the range of what can be represented by in a fixed number of bits.

- Adding +ve and -ve operand, no overflow because the sum is always within the range
- Adding two +ve operands or two -ve operands,

Possible overflow:

⇒ two +ve ~~sum~~ operands
Overflow, if result sign is 1 (negative)

⇒ two -ve operands
overflow, if result sign is 0 (positive)

Integer substraction: Add negation of second operand

$$7 - 6 = 7 + (-6)$$

+7	:	0000 0000 ..... 0000 0111
-6	:	1111 1111 ..... 1111 1010
		0000 0000 ..... 0000 0001

Overflow happens if the result is outside the valid range.

- No overflow: ↑ subtracting two +ve number or two -ve operand
- Possible overflow:
  - Subtracting a +ve from -ve operand (overflow if result sign is 0)
  - Subtractive -ve from +ve operand (overflow if result is 1)

Dealing with overflow:

- Some languages ignore overflow. (e.g. C)
  ⇒ Use MIPS addu, addui, subu instructions
- Some languages (e.g. Ada, Fortran) require raising an exception by detecting overflow.
  ⇒ Use MIPS add, addi, sub instructions.
  ⇒ On overflow, occurs, the processor:
  - invoke exception handler in EPC (Exception program counter) register, to track error

P.T.O

• jump to predetermined handler address to ~~dete~~ handle the error.

⇒ mfc0 (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

Arithmetic for Multimedia:

- Graphics and media processing operates on vectors of 8-bit and 16 bit data.
- Use 64-bit adder, to process multiple values at once

⇒ Operate on 8×8 bit, 4×16 bit or 2×32 bit vectors

- SIMD (single instruction, multiple data)
- Saturating operations: On overflow, result is largest representable value, i.e. the largest positive number or the most negative number when overflow occurs
- Saturation operation for media operations.

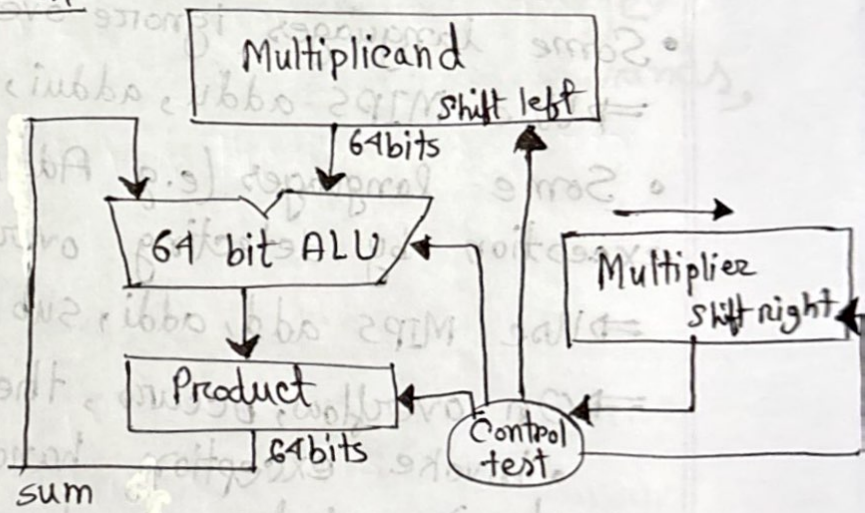
example - volume knob

Long multiplication approach:-

Multiplicand: 1000
Multiplier : 1001

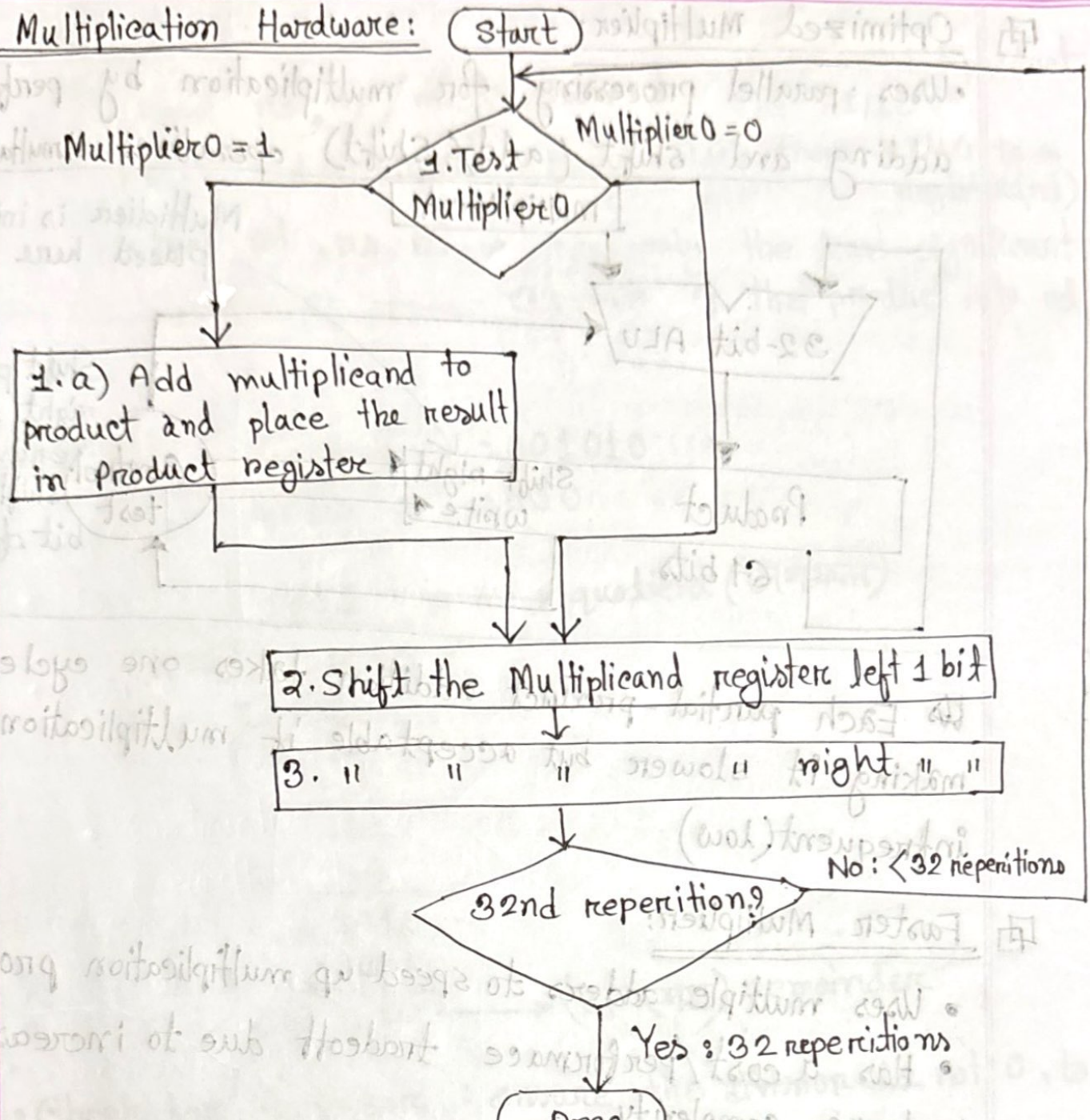
```
    1000
   0000x
  0000xx
 1000xxx
---------
1001000
```

product: 1001000

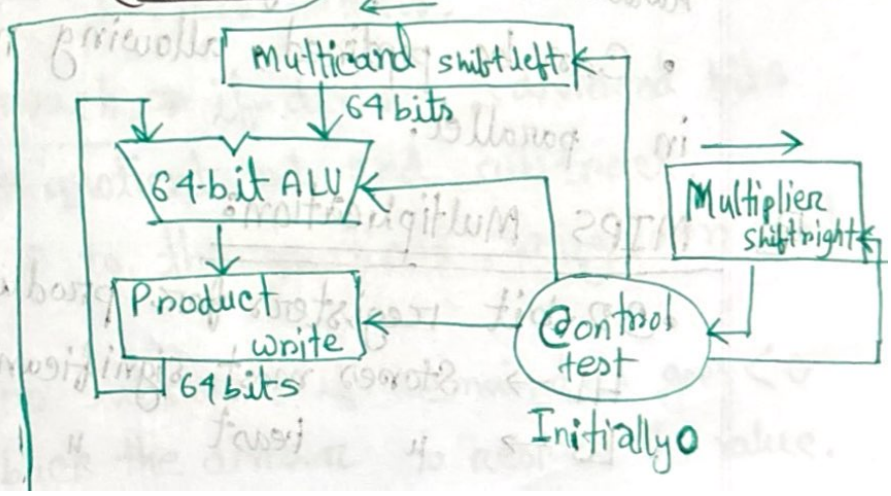


Length of product is the sum of operand lengths

Multiplication Hardware:

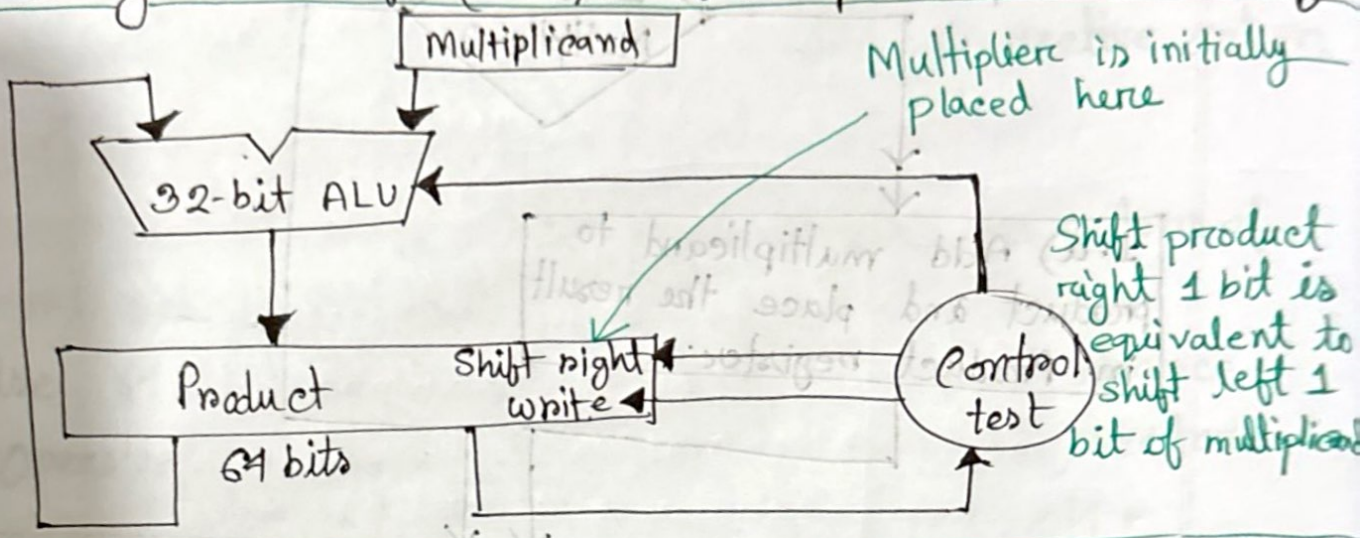


Slide - (pg 10) chart



Optimized Multiplier:

- Uses parallel processing for multiplication by performing adding and shift (add/shift) operations simultaneously.



- Each partial-product addition takes one cycle, making it slower but acceptable if multiplication is infrequent (low)

Faster Multiplier:

- Uses multiple adders to speed up multiplication process,
- Has a cost/performace tradeoff due to increased hardware complexity.
- Can be pipelined, allowing multiplications performed in parallel.

MIPS Multiplication:

- 32-bit registers for product:
  - HI → Stores most significant 32bits
  - LO → " least " "

Instructions:

- mult rs, rt / multu rs, rt → Stores 64-bit product in HI/LO
- mfhi rd / mflo rd → Moves values from HI/LO to a register (rd)
- mul rd, rs, rt → Stores only the least-significant 32-bits of the product into rd

Division:

dividend 1001010
divisor 1000

1001 ⟶ quotient (ভাগফল)

```
1000 ) 1001010 (
      -1000
      -----
        101
         -0
        ---
        1010
       -1000
       -----
          10  ⟶ (ভাগশেষ) remainder
```

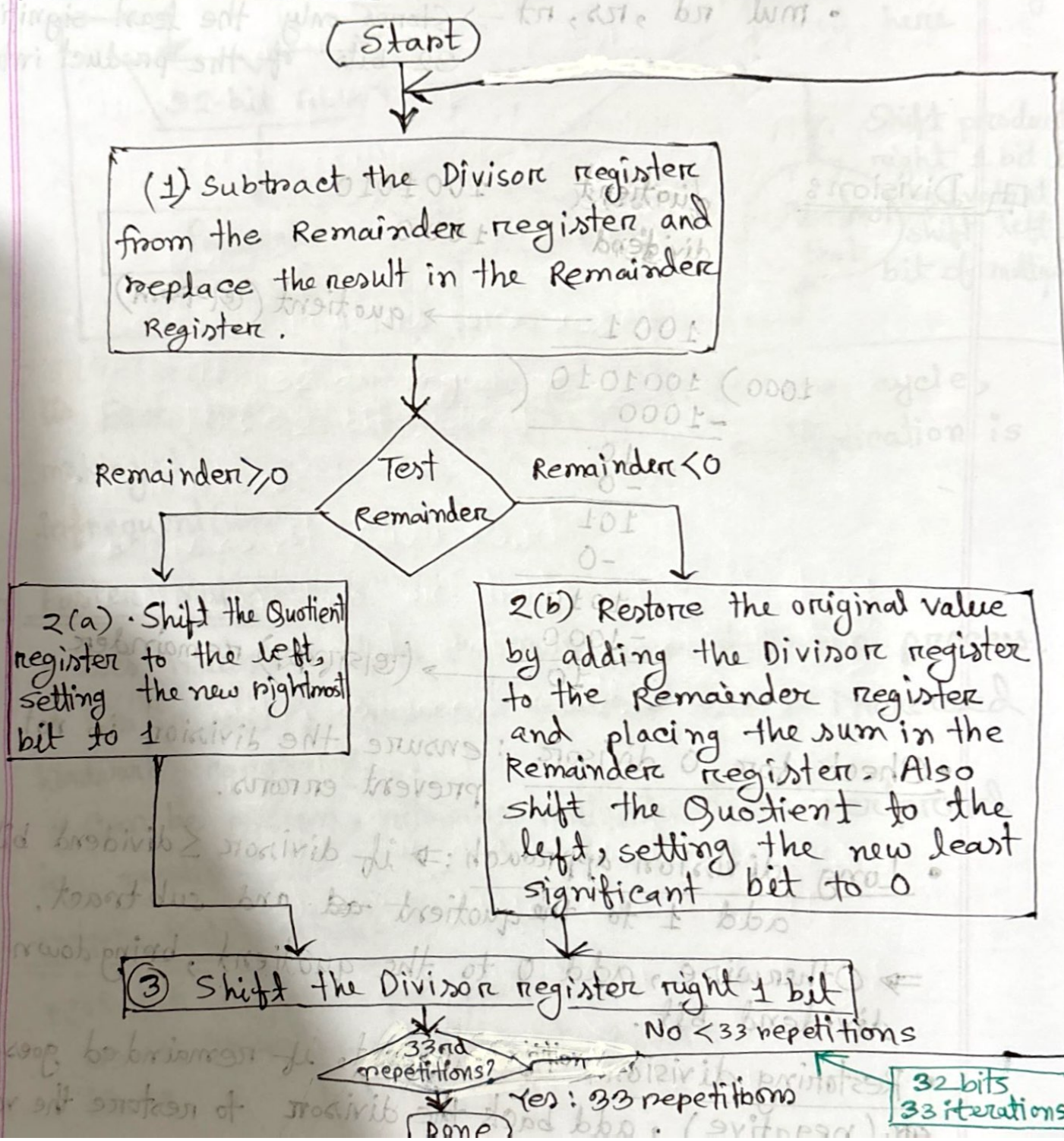
- Check for 0 divisor : ensure the division is not 0, to prevent errors.
- Long division approach :⇒ if divisor ≤ dividend bits add 1 to the quotient and subtract.

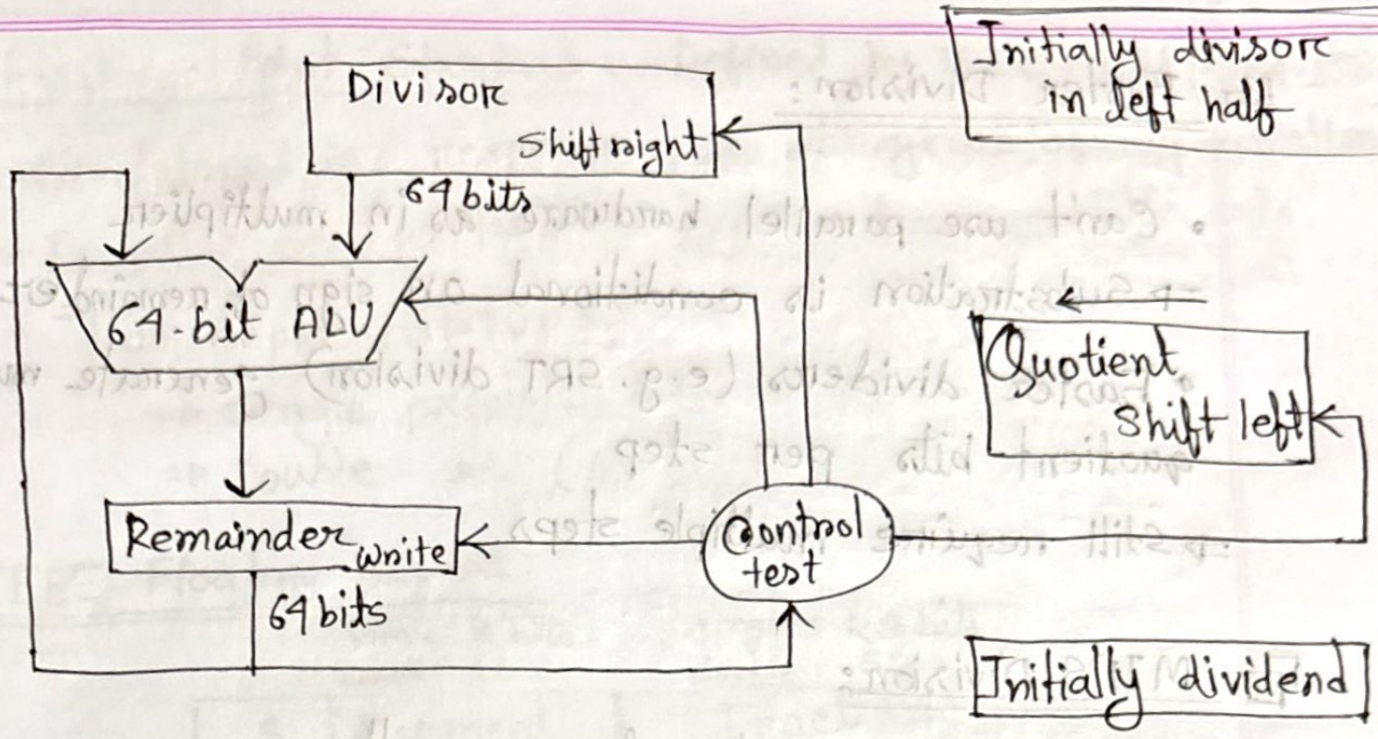
⇒ Otherwise, add 0 to the quotient, bring down next dividend bit.

- Restoring division: Do subtract, if remainded goes < 0 (negative), add back the divisor to restore the value.

• Signed Division: • Divide using absolute values.
• Adjust the signs of quotient and remainder as required.

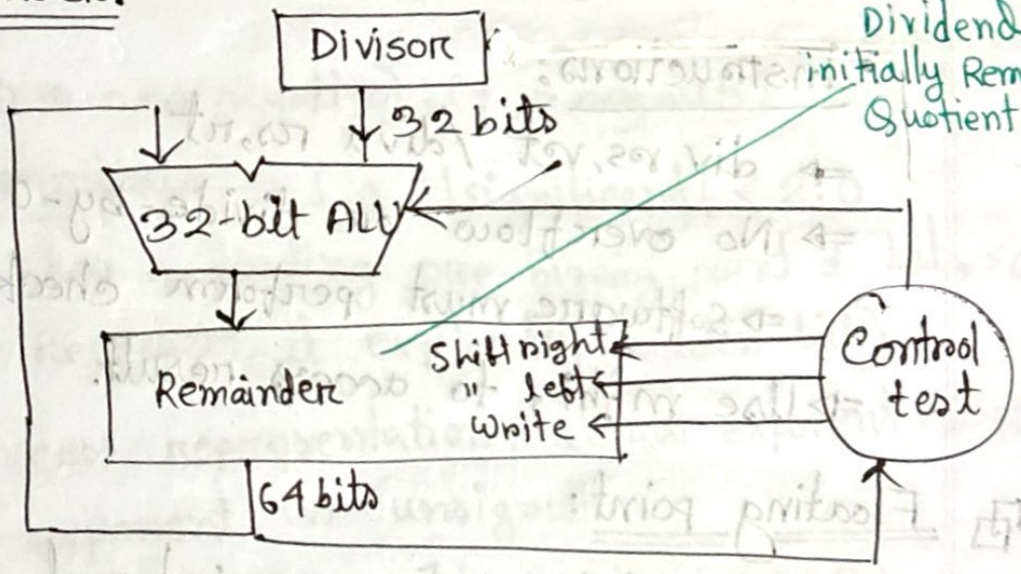
Division Hardware:





Slide : Pg - 17 (Chart)

Optimized Divider:



- One cycle per partial-remainder subtraction
- Looks like a multiplier
  ⇒ Same hardware can be used for both.

Faster Division:

- Can't use parallel hardware as in multiplier
  ⇒ Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  ⇒ Still require multiple steps.

MIPS Division:

- Use HI/LO registers for result:
  HI : 32-bit remainder
  LO : 32-bit quotient
- Instructions:
  ⇒ div rs, rt / divu rs, rt
  ⇒ No overflow or divide-by-0 checking
    ⇒ Software must perform checks if required
  ⇒ Use mfhi, to access result.

Floating point:

- Representation for non-integral numbers, including very small and very large numbers, like scientific notations —

-2.34×10^{56}
$+0.002 \times 10^{-4}$
$+987.02 \times 10^{9}$

(like float and double in C)

In binary: $\pm 1.xxxxx_2 \times 2^{yyyy}$

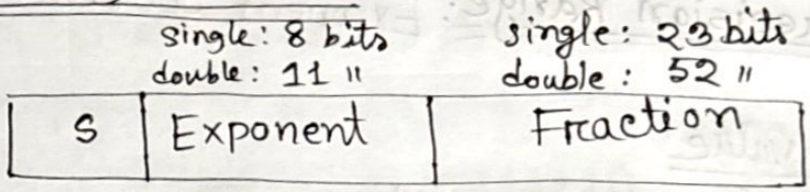
Floating Point Standard: Defined by IEEE Std (754-1985)

- Developed in response to divergence of representations

⇒ Essential for portability issues for scientific code.

- Two representations
  - ⇒ Single precision (32-bit)
  - ⇒ Double " (64-bit)

IEEE Floating Format:



$$X = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$$

→ View + as a binary point

S: sign bit (0 ⇒ non-negative); (1 ⇒ negative)

Normalize significand: $1.0 \leq |\text{significand}| < 2.0$

- Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)

Exponent: excess representation: actual exponent + Bias

- Ensure exponent is unsigned
- Single: Bias = 127; Double: Bias = 1203;

Single-precision Range:

- Exponents 00000000 and 11111111 reserved
- Smallest value: Exponent 00000001
  - ⇒ actual " = 1 − 127 = −126

- Fraction : 0000...00 ⇒ significand = 1.0
- $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

■ Largest value : exponent : 11111110

⇒ actual # = 254 - 127 = +127

- Fraction : 111....11 ⇒ significand ≈ 2.0
- $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double - Precision Range : Exponent 0000...00 and 111....11 reserved

■ Smallest value

- Exponent : 00000000001

⇒ actual # : 1 - 1023 = -1022

- Fraction : 0000....00

⇒ significand = 1.0

■ Largest value :

- Exponent : 11111111110

⇒ actual # : 2046 - 1023 = +1023

- Fraction : 111....11

⇒ signifacnd ≈ 2.0

- $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Relative Precision :

all fraction bits are significant

- Single : approx 2^{-23}

Equivalent to $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision

- Double : approx 2^{-52}

  Equivalent to $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ decimal # digits of precision

Example of Floating Point:-

Represent -0.75

- $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
- S = 1
- Fraction = $1000\ldots00_2$
- Exponent = -1 + Bias

Single : 1 01111110 1000...00

Double : 1 01111111110 1000...00

What number is represented by the single-precision float.

1 10000001 01000...00

- S = 1
- Fraction = $01000\ldots00_2$
- Exponent = $10000001_2 = 129$

- $X = (-1)^1 \times (1 + 01_2) \times 2^{(129-127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Floating-Point Addition

- Consider a 4-digit decimal example

$9.999 \times 10^1 + 1.610 \times 10^{-1}$

(1) Align decimal points

Shift number with smaller exponent

$9.999 \times 10^1 + 0.016 \times 10^1$

(2) Add significands

$9.999 \times 10^1 + 0.016 \times 10^1$

$= 10.015 \times 10^1$

(3) Normalize result and check for over/underflow

$1.0015_2 \times 2^{-4}$, with no overflow / underflow

(4) Round and renormalize if necessary

$1.000_2 \times 2^{-4}$ (no change)

$= 0.0625$

FP Adder Hardware: More complex than integer adder. Doing it in one cycle would take too long. It typically take multiple cycles to complete. It can be pipelined to improve throughput.

~~Group~~ (Diagram - Pg 32)

FP Multiplication:

Given, $1.000_2 \times 2^{-1}$

$-1.110_2 \times 2^{-2}$

In decimal $(0.5 \times (-0.4375))$

① Add exponents:

Unbiased : $-1 + -2 = -3$

biased : $(-1 + 127) + (-2 + 127) = -3 + 127$

② Multiply significands:

$1.000_2 \times 1.110_2 = 1.1102$

$\Rightarrow 1.110_2 \times 2^{-3}$

③ Normalize result & check for over/underflow.

$1.110_2 \times 2^{-3}$ (no change) with no " "

④ Round and renormalize if necessary.

$1.110_2 \times 2^{-3}$ (no change)

⑤ Determine sign: +ve x -ve $\Rightarrow$ -ve

$-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware:

- Similar complexity to FP adder.

Key difference: Use multiplication for significands instead of an adder.

- FP Arithmetic hardware does addition, multiplication, division, reciprocal, square-root.

FP $\leftrightarrow$ Integer conversion

Accurate Arithmetic:

- IEEE Standard 754 specifies mechanism for improving accuracy.
  - Extra bits of precision (guard, round, sticky)
  - choice of rounding modes
  - Allows programmer to fine-tune numerical behaviour of a computation
- Not all FP units implement every feature or rounding mode.
- There's tradeoff between hardware complexity.

Subword parallelism:

- Used in graphics, audio, other media applications.
- Takes advantage of performing multiple operations simultaneously on short vectors within larger word.

  Example:
  - Sixteen 8-bit adds
  - Eight 16-bit "
  - Four " "

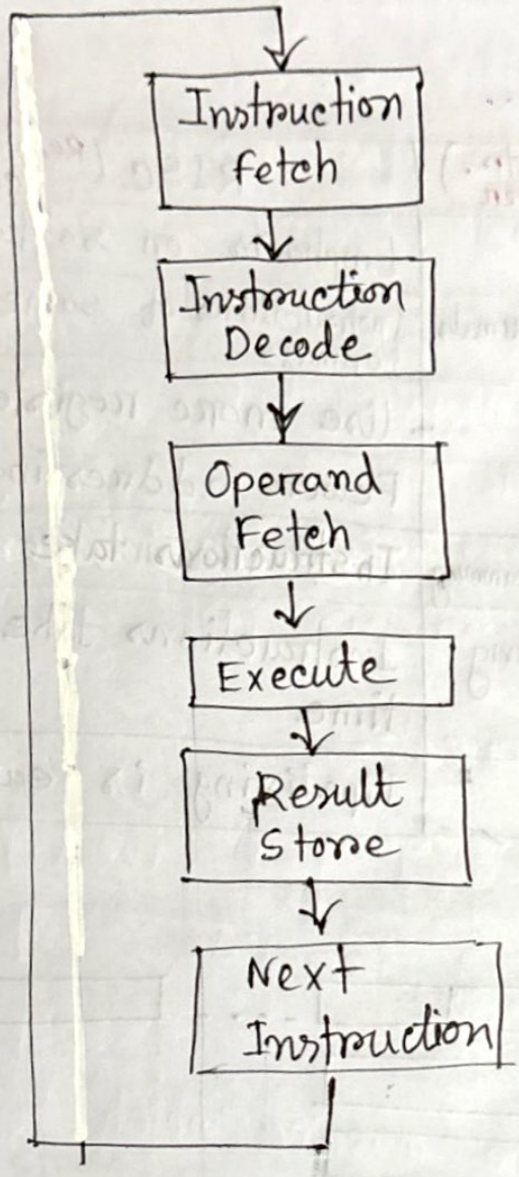
  Also called data level parallelism, vector parallelism or single instruction, Multiple Data (SIMD)

Conclusion:

- Bits have No inherent meaning
- Slide - Pg - 38, 39.

General classification of ISA:

- **Instruction set:**



Obtain instruction from program storage

Determine required actions and instruction size

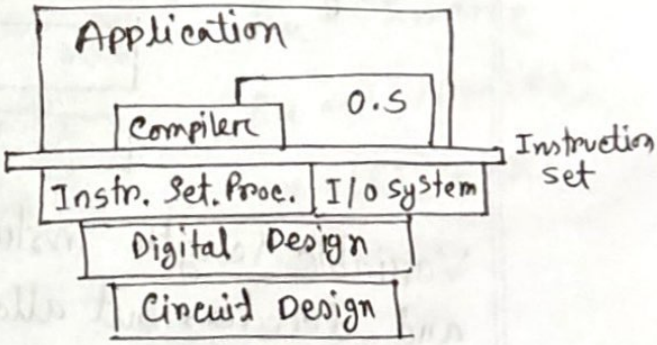
Locate and obtain operand data

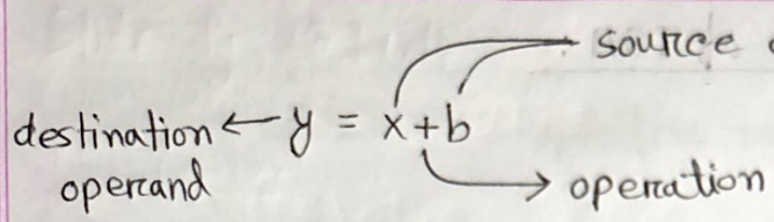
Complete result value or status

Deposit results in storage for later use

Determine successor instruction

- To command a computer's hardware, you must speak its language. The words of a computer

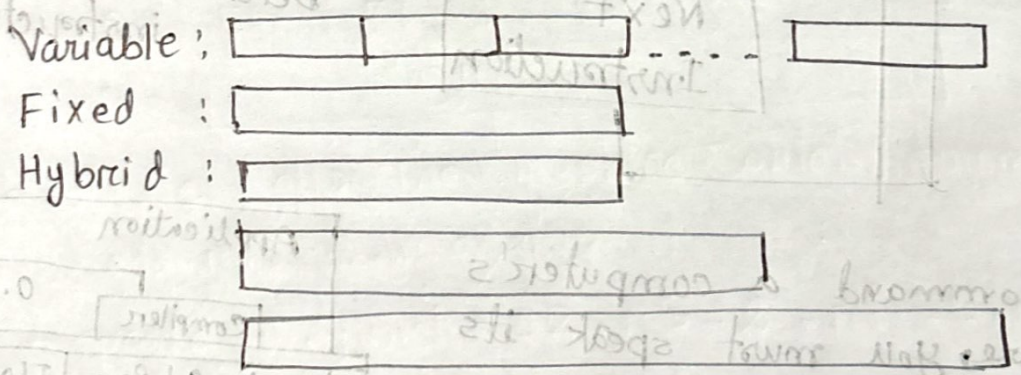




RISC vs CISC :

CISC (Complex Instr. Set Computer)	RISC (Reduced Instr. Set Computer)
Emphasis on hardware	Emphasis on software
Multiple Instruction sizes & formats	Instructions of same set with few formats
Use less registers	Use more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in Compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy

Instruction length :



Variable length instructions require multi-step fetch and decode, but allow for a much more flexible and

compact instruction set.

Fixed-length instructions allow easy fetch and decode, and simplify pipelining and parallelism.

How many operands?

- Most instructions have three operands (e.g $z = x + y$)
- Well-known ISAs specify 0-3 (explicit) operands per instruction.
- Operands can be specified implicitly or explicitly.

Basic ISA Classes:

- **Accumulator:**

  1 address — add A — acc ← acc + mem[A]

- **Stack:**

  0 address — add — tos ← tos + next

- **General purpose Register:**

  2 address — add A B — EA

  3 " — add A B C

- **Load/store:**

  3 address — add Ra Rb Rc — Ra ← Rb + Rc

  load Ra Rb — Ra ← mem[Rb]

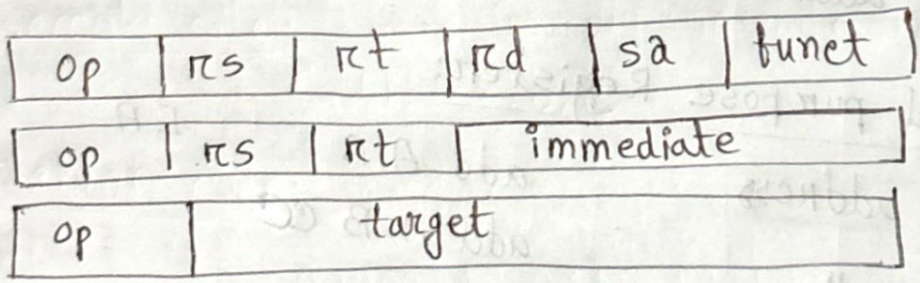
  store Ra Rb — mem[Rb] ← Ra

Slide Pg- 13,14,15

Key points:

- MIPS is a general purpose register, load store, fixed instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count.
- Four Principles of IS architecture
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast

MIPS Instruction Formats:

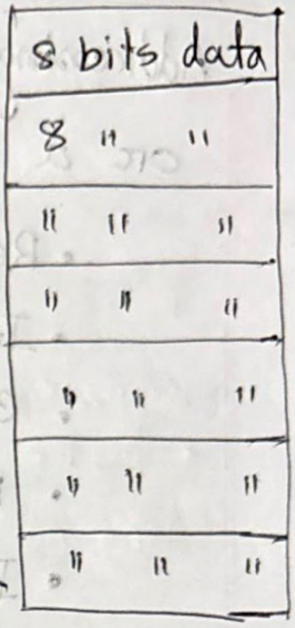


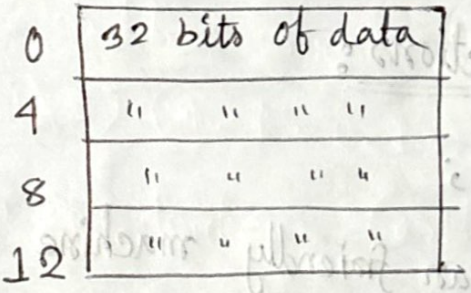
- the opcode tells the machine which format
- so, add r1, r2, r3 has
  - opcode = 0, funct = 32, rs = 2, rt = 3, rd = 1, sa = 0
  - 000000 00010 00011 00001 000000 100000

(Chart – pg 19)

Memory organization:

- Viewed as a large single-dimension array, with an address
- A memory address is an index into the array.
- "Byte Addressing" means that the index points to a byte of memory.
- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes





Registers hold 32 bits of data

- 2^{32} bytes with bytes addresses from 0 to 232-1
- 2^{30} " " " " 0, 4, 8, ..., 232-4
- Words are aligned.

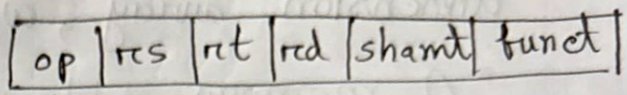
~~i.e. what are th~~

Which Instructions:

- arithmetic
- logical
- data transfers
- conditional branch
- unconditional jump

MIPS addressing modes

Addressing modes are the ways of specifying an operand or a memory address

- Register addressing => 

- Immediate "
- Base "
- PC-relative "
- Indirect "
- Direct " (almost)

Pg – (25 - 30) Slide.

Basic MIPS Instructions:

- C code: $a = b + c;$
- Assembly code: (human-friendly machine instructions)

```
add  a, b, c
```

- Machine code: (hardware " " ")

```
00000010001100100000000100000
```

Translate C code → assembly code:

$a = b + c + d + e;$

```
add  a, b, c
add  a, a, d
add  a, a, e
```

or,

```
add  a, b, c
add  f, d, e
add  a, a, f
```

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code.
- Some sequences are better than others... the second sequence needs one more (temporary) variable f

Subtract Example:

C code: `f = (g+h) - (i+j);`

Assembly code translation with only add and sub instructions:-

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

or,

```
add f, g, h
sub f, f, i
sub f, f, j
```

- Each version may produce a different result because floating-point operations are not necessarily associative and commutative... more on this later.

Operands:

- In C, each variable is a location in memory.
- In hardware, each memory access is expensive - if variable a is accessed repeatedly, it helps to bring the " into an on-chip scratchpad and operate on the scratchpad (registers)
- To simplify the instructions, we require that each instruction (add, sub) only operate on registers.

Note: The number of operands (variables) in a C program is very large; the number of operands in assembly is fixed. There can be only so many scratchpad registers. (temporary location in CPU).

□ Registers:

- MIPS ISA has 32 registers (x86 has 8 registers) - Why not more or less?

⇒ Because each register is 32 bits wide and MIPS is a ~~MIPS~~ 32-bit architecture.

⇒ A 32-bit entity is referred as a word.

⇒ To make the code more readable, registers are partitioned, such as ($s0 - $s7) ← (C/Java variables), $t0 - $t9 (temp variables); which means efficient compiled code support (like C/Java)

⇒ Matches RISC design goals.

⇒ keep hardware simple and fast

So, MIPS is a design trade-off that balances performance, complexity and cost.