Computer Architecture

UNIT I BASIC STRUCTURE OF A COMPUTER SYSTEM

Functional Units - Basic Operational Concepts - Performance - Instructions: Language of the Computer - Operations, Operands - Instruction representation Logical operations - decision making - MIPS Addressing.

EIGHT IDEAS

1. Design for Moore's Law

.

- Moore's Law resulted from a 1965 prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel.
- The resources available per chip can easily double or quadruple between the start and finish of the project. i.e., it states that integrated circuit resources double every 18-24 months.
- Computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts.
- We use an "up and to the right" Moore's Law graph to represent designing for rapid change.

2. Use Abstraction to Simplify Design

- A major productivity technique for hardware and software is to use abstractions to represent the design at different levels of representation; lower-
level details are hidden to offer a simpler model at higher levels.

3. Make the Common Case Fast

- Making the common case fast will tend to enhance performance better than optimizing the rare case.

4. Performance via Parallelism

- Computer architects have offered designs that get more performance by performing operations in parallel.

5. Performance via Pipelining

- Pipeline icon is a sequence of pipes, with each section representing one stage of the pipeline.

6. Performance via Prediction

- It can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate.

7. Hierarchy of Memories

- The fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom.

8. Dependability via Redundancy

- Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems dependable by including redundant components that can take over when a failure occurs and to help detect failures.

FUNCTIONAL UNITS

- .A computer consists of five functionally independent main parts Input, Memory, Arithmetic and Logic unit (ALU), Output and Control unit.

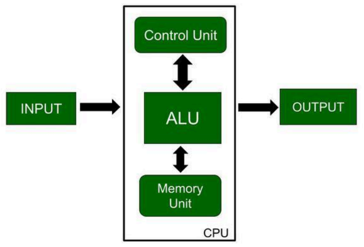
INPUT UNIT

- The source program/high level language program/coded information/simply data is fed to a computer through input devices keyboard is a most common type. Whenever a key is pressed, one corresponding word or number is translated into its equivalent binary code over a cable & fed either to memory or processor.
- Example:- Joysticks, trackballs, mouse, scanners etc are other input devices.

Mouse

- The original mouse was electromechanical and used a large ball that when rolled across a surface would cause an x and y counter to be incremented.
- Nowadays, the electromechanical mouse is replaced by optical mouse. The replacement of optical mouse, reduce cost and increase reliability.
- It includes Light emitting diode (LED) to provide lighting, a tiny black and white camera.The LED is underneath the mouse, the camera takes 1500 sample pictures/second.
- The sample pictures are sent to optical processor that compare the images and determine how far the mouse has moved.

CENTRAL PROCESSOR UNIT (CPU)



- A CPU is also called a processor. The active part of the computer, which contains the datapath and control units and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

ALU or DATAPATH UNIT

- It performs the arithmetic operations.
- It is also called as brawn of the processor.
- The entire operation can be done with the help of registers.
- Registers are faster than memory.
- Registers can hold variables and intermediate results.
- Memory traffic is reduced, so program runs faster.

CONTROL UNIT

- It is also called as a brain of the processor.
- It fetches and analyses the instructions one-by-one and issue control signals to all other units to perform various operations.
- For a given instruction, the exact set of operations required is indicated by the control signals. The results of instructions are stored in memory.
- The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.

MEMORY UNIT

- Memory is nothing but a storage device. It stores the program and data.
- It is divided into ‘n’ number of cells. Each cell is capable of storing one bit information at

a time.

- There are 2 classes of memory.1.Primary 2.Secondary.

Primary Memory

- It is made up of semiconductor material. So it is called Semiconductor memory
- Data storage capacity is less than secondary memory.
- Cost is too expensive than secondary memory.
- CPU can access data directly. Because it is an internal memory.
- Data accessing speed is very fast than secondary memory.
- Ex.RAM & ROM

RAM	ROM
Random Access Memory	Read Only Memory
Volatile memory	Non volatile memory
Data lost when the power turns off and that is used to hold data and program while they are running.	It retains data even in the absence of a power source and that is store programs between runs.
Temporary storage medium	Permanent storage medium
User perform both read and write operation	User can perform only read operation

RAM: There are two types of memory available namely, 1. SRAM 2. DRAM

SRAM	DRAM
Static RAM	Dynamic RAM
Information is stored in 1 bit cell called Flip Flop.	Information is represented as charge across a capacitor
Information will be available as long as power is available	It retains data for few ms based on the charge of capacitor even in the absence of a power
No refreshing is needed	Refreshing is needed
Less packaging density	High packaging density
More complex Hardware	Less complex hardware
More expensive	Less expensive
No random access	Random access is possible
Access time 10 ns	Access time 50 ns

Cache Memory: A small, fast memory that acts as a buffer for a slower, larger memory.

Secondary memory

- Secondary memory (Nonvolatile storage) is a form of storage that retains data
- even in the absence of a power source and that is used to store the programs between runs.
- It is made up of magnetic material. So it is called magnetic memory.
- Data storage capacity is high than primary memory.
- Cost is too low than primary memory.
- CPU cannot access data directly. Because it is an external memory.
- Data accessing speed is very slow than primary memory.
- Ex. Magnetic disk, Hard Disk, CD, DVD, Floppy Disk

Magnetic Disk

- It consists of a collection of platters, which rotate on a spindle at 5400 revolution/min.
- The metal platters are covered with magnetic recording material on both sides.
- Also called hard disk. A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material.
- Because they are rotating mechanical devices, access times are about 5 to 20 milliseconds and cost per gigabyte in 2012 was $0.05 to $0.10.

Optical Disk: Include both Compact Disk (CD) and Digital Video Disk(DVD).

Read-Only CD/DVD

- Data is recorded in a spiral fashion, with individual bits being recorded by burning small pit.
- The disk is read by shining a laser at the CD surface and determining by examining the reflected light whether there is a pit or flat surface.

Rewritable CD/DVD

- Use different recording surface that as a crystal line, reflective materials, pits are formed that are not reflective.

Erase CD/DVD

- The surface is heated and cooled slowly, allowing an annealing process to restore the surface recording layer to its crystalline structure.

Flash Memory

- A nonvolatile semiconductor memory. It is cheaper and slower than DRAM but more expensive per bit and faster than magnetic disks. Access times are about 5 to 50 microseconds and cost per gigabyte in 2012 was $0.75 to $1.00.

OUTPUT UNIT

- A mechanism that conveys the result of a computation to a user, such as a display, or to another computer.

Liquid Crystal Display

- A display technology using a thin layer of liquid polymers that can be used to transmit or block light according to whether a charge is applied.
- All laptops and desktop computers use Liquid Crystal Display (LCD) to get a thin, low-power display. A tiny transistor switch at each pixel to control current and make sharper images.
- The image is composed of a matrix of picture elements or pixels, which can be represented as a matrix of bits called **bitmap.**
- Depending on the size of the screen and the resolution, the display matrix ranges in size from 640*480 to 2560*1600.
- A red, green, blue (RGB) associated with each dot on the display determines the intensity of the three color components in the final image.
- **Pixel:** The smallest individual picture element. Screens are composed of hundreds of thousands to millions of pixels, organized in a matrix.

<u>**BASIC OPERATIONAL CONCEPTS**</u>

To perform a given task an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be stored are also stored in the memory.

Examples: Add LOCA, R0

This instruction adds the operand at memory location LOCA, to operand in register R0 & places the sum into register. This instruction requires the performance of several steps,

1. First the instruction is fetched from the memory into the processor
2. The operand at LOCA is fetched and added to the contents of R0
3. Finally the resulting sum is stored in the register R0

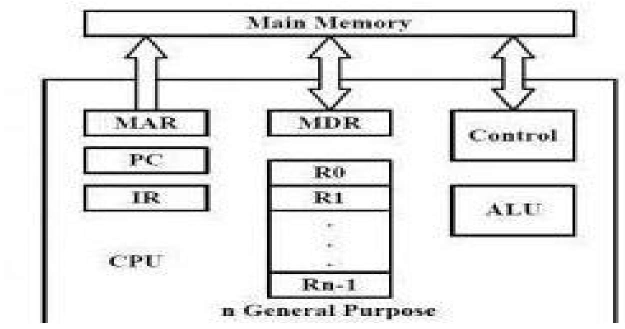


Fig: Connections between the processor and the memory

The fig shows how memory & the processor can be connected. In addition to the ALU & the control circuitry, the processor contains a number of registers used for several different purposes.

Instruction Register (IR)

- Holds the instruction that is currently being executed.
- Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction.

Program Counter PC

- This is another specialized register that keeps track of execution of a program.
- It contains the memory address of the next instruction to be fetched and executed. Besides IR and PC, there are n-general purpose registers R0 through Rn-1. MAR PC IR MEMORY MDR R0 R1 …

The other two registers which facilitate communication with memory are:

1. MAR – (Memory Address Register):- It holds the address of the location to be accessed.

2. MDR – (Memory Data Register):- It contains the data to be written into or read out of the address location.

Operating steps are

1. Programs reside in the memory & usually get these through the I/P unit.
2. Execution of the program starts when the PC is set to point at the first instruction of the program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to the memory.
4. After the time required to access the memory elapses, the address word is read out of the memory and loaded into the MDR.
5. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
6. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
7. An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.
8. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
9. After one or two such repeated cycles, the ALU can perform the desired operation.
10. If the result of this operation is to be stored in the memory, the result is sent to MDR.
11. Address of location where the result is stored is sent to MAR & a write cycle is initiated.
12. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

<u>PERFORMANCE</u>

The machine (or CPU) is said to be faster or has better performance running this program if the total execution time is shorter. When trying to choose among different computers, performance is an important attribute.

1. Throughput and Response

Time Response time

- Response time also called execution time. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

Throughput or bandwidth

- The total amount of work done in a given time. Decreasing response time almost always improves throughput.
- To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\text{Performance}_X > \text{Performance}_Y$$
$$\frac{1}{\text{Execution time}_X} > \frac{1}{\text{Execution time}_Y}$$
$$\text{Execution time}_Y > \text{Execution time}_X$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

We will use the phrase "X is n times faster than Y"—or equivalently "X is n times as fast as Y"—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is n times as fast as Y, then the execution time on Y is n times as long as it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Relative Performance

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

We know that A is n times as fast as B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times as fast as B.

In the above example, we could also say that computer B is 1.5 times slower than computer A

2. Measuring Performance

CPU execution time: Also called **CPU time**. The actual time the CPU spends computing for a specific task.

User CPU time: The CPU time spent in a program itself.

System CPU time: The CPU time spent in the operating system performing tasks on behalf of the program.

Clock cycle: Also called **tick**, **clock tick**, **clock period**, **clock**, or **cycle**. The time for one clock period, usually of the processor clock, which runs at a constant rate.

Clock period: This is the length of each clock cycle.

Clock rate: This is the inverse of the clock period.

3. **CPU Performance and Its Factors**
   - A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time.
   - This formula makes it clear that the hardware designer can improve performance by reducing the number of clock cycles required for a program or the length of the clock cycle.

$$\text{CPU execution time for a program} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

Example

Improving Performance

Our favorite program runs in 10 seconds on computer A, which has a 2 GHz clock. We are trying to help a computer designer build a computer, B, which will run this program in 6 seconds. The designer has determined that a substantial increase in the clock rate is possible, but this increase will affect the rest of the CPU design, causing computer B to require 1.2 times as many clock cycles as computer A for this program. What clock rate should we tell the designer to target?

Let's first find the number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \text{ cycles}$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \text{ cycles}}{\text{second}} = \frac{4 \times 10^9 \text{ cycles}}{\text{second}} = 4 \text{ GHz}$$

To run the program in 6 seconds, B must have twice the clock rate of A.

4. **Instruction Performance**
   - Execution time is that it equals the number of instructions executed multiplied by the average time per instruction.
   - Therefore, the number of clock cycles required for a program can be written as

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \text{Average clock cycles per instruction}$$

Clock cycles per Instruction

- The term clock cycles per instruction, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as CPI.
- Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program.

Example

Suppose we have two implementations of the same instruction set architecture.Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program,and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

We know that each computer executes the same number of instructions for the program; let's call this number I. First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$
$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\text{CPU time}_A = \text{CPU clock cycles}_A \times \text{Clock cycle time}$$
$$= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

5. **The Classic CPU Performance Equation**

We can now write this basic performance equation in terms of instruction count (the number of instructions executed by the program), CPI, and clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

$$\text{Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

Example

Comparing Code Segments

A compiler designer is trying to decide between two code sequences for a particular computer. The hardware designers have supplied the following facts:

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

Code sequence	Instruction counts for each instruction class		
	A	B	C
1	2	1	2
2	4	1	1

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

Sequence 1 executes 2 + 1 + 2 = 5 instructions. Sequence 2 executes 4 + 1 + 1 = 6 instructions. Therefore, sequence 1 executes fewer instructions.

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2.0$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$

Amdahl's Law

A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

Million Instructions per Second (MIPS)

- A measurement of program execution speed based on the number of millions of instructions.
- MIPS are computed as the instruction count divided by the product of the execution time. MIPS is easy to understand, and faster computers.

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

- There are three problems with using MIPS as a measure for comparing computers.
- First, we cannot compare computers with different instruction sets using MIPS, since the instruction counts will certainly differ.
- Second, MIPS varies between programs on the same computer; thus, a computer cannot have a single MIPS rating.
- For example, by substituting for execution time, we see the relationship between MIPS, clock rate, and CPI:

$$\text{MIPS} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

- Finally, and most importantly, if a new program executes more instructions but each instruction is faster, MIPS can vary independently from performance.

<u>INSTRUCTION</u>

The words of a computer's language are called instructions, and its vocabulary is called an instruction set. Instruction performs one of the following operations

- Data transfer between register and memory.
- ALU operation.
- Program control and sequencing.
- I/O transfer.

Stored program concept

The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer.

<u>OPERATIONS OF THE COMPUTER HARDWARE</u>

1. Arithmetic Operations

Add and Subtract instruction use three Operands - Two Sources and one Destination.

Example add $t0, $s1, $s2

2. Data Transfer Operations

These operations help in moving the data between memory and registers.

Example: 1

C code:

```
g = h + A[8];
```

- g in $s1, h in $s2, base address of A in $s3

Compiled MIPS code:

- Index 8 requires offset of 32
  - 4 bytes per word

```
lw  $t0, 32($s3)     # load word
add $s1, $s2, $t0
```

offset

base register

Example: 2

C code:

```
A[12] = h + A[8];
```

- h in $s2, base address of A in $s3

Compiled MIPS code:

- Index 8 requires offset of 32

```
lw  $t0, 32($s3)     # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)     # store word
```

3. Logical Operations

Logical Instructions are used for bitwise manipulation. It is useful for extracting and inserting groups of bits in a word

Operation	C	MIPS
Shift left	<<	sll
Shift right	>>	srl
Bitwise AND	&	and, andi
Bitwise OR	|	or, ori
Bitwise NOT	~	nor

4. Conditional Operations

1. Branch to a labeled instruction if a condition is true. Otherwise, continue sequentially.

    beq rs, rt, L1

    if (rs == rt) branch to instruction labeled L1;

    bne rs, rt, L1

    if (rs != rt) branch to instruction labeled L1;

    j L1

2. Unconditional jump to instruction labeled L1. Set result to 1 if a condition is true. Otherwise, set to 0.

    slt rd, rs, rt

    if (rs < rt) rd = 1; else rd = 0;

    slti rt, rs, constant

    if (rs < constant) rt = 1; else rt = 0;

3. Use in combination with beq,
    bne **slt $t0, $s1, $s2** # if ($s1 <
    $s2) **bne $t0, $zero, L** #
    branch to L Jump Instructions

Jump Address Table: Also called **jump table**. A table of addresses of alternative instruction sequences.

4. Procedure call: jump and link.

Jump-and-Link Instruction: An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register($rain MIPS).

jal ProcedureLabel
Address of following instruction put in $ra.
Jumps to target address.

5. Procedure return: jump register

Return address: A link to the calling site that allows a procedure to return to the proper address in MIPS it is stored in register $ra.

jr $ra
Copies $ra to program counter.
Can also be used for computed jumps.
e.g., for case/switch statements.

Procedure: A stored subroutine that performs a specific task based on the parameters with which it is provided.

Caller: The program that instigates a procedure and provides the necessary parameter values.

Callee: A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

Program counter (PC): The register containing the address of the instruction in the program being executed.

Stack: A data structure for spilling registers organized as a last-in first-out queue.

Stack pointer: A valued noting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In MIPS, it is register $sp.

Push: Add element to stack.

Pop: Remove element from stack.

Global pointer: The register that is reserved to point to the static area.

<u>**OPERANDS OF THE COMPUTER HARDWARE**</u>

1. Register Operands

- Arithmetic instructions use register operands and MIPS has a 32 × 32-bit register file
- Mainly used for frequently accessed data
- Register numbered 0 to 31 and 32-bit data called a ―word
- $t0, $t1, …, $t9 for temporary values
- $s0, $s1, …, $s7 for saved variables

Word: The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

Compiling a C Assignment Using Registers

Example: f = (g + h) – (i + j); The variables f, g, h, i, and j are assigned to the registers $s0, $s1, $s2, $s3, and $s4, respectively. What is the compiled MIPS code?

MIPS code

add $t0,$s1,$s2 # register $t0 contains g + h

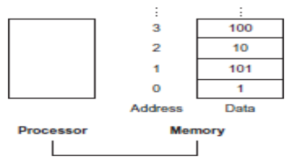
add $t1,$s3,$s4 # register $t1 contains i + j

sub $s0,$t0,$t1 # f gets $t0 – $t1, which is (g + h)–(i + j)

2. Memory Operands

Data transfer instruction: A command that moves data between memory and registers.

Address: A value used to delineate the location of a specific data element within a memory array. For example, in the following figure, the address of the third data element is 2, and the value of Memory [2] is 10.



2.1 Load:

- The data transfer instruction that copies data from memory to a register is traditionally called load.
- The format of the load instruction is the name of the operation followed by the register to be loaded, then a constant and register used to access memory.
- The sum of the constant portion of the instruction and the contents of the second register forms the memory address.
- The actual MIPS name for this instruction is lw, standing for load word.

Compiling an Assignment When an Operand Is in Memory

Example: Let's assume that A is an array of 100 words and that the compiler has associated the variables g and h with the registers $s1 and $s2 as before. Let's also assume that the starting address, or base address, of the array is in$s3. Compile this C assignment statement:

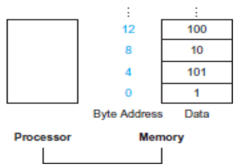
g = h + A[8];

Solution:

```
lw $t0,32($s3)        # Temporary reg $t0 gets A[8]
add $s1,$s2,$t0       # g = h + A[8]
```

The constant in a data transfer instruction (8) is called the off set, and the register added to form the address ($s3) is called the base register.

Alignment restriction: A requirement that data be aligned in memory on natural boundaries.



Little Endian Address: A lower byte address of the word is specified in least significant byte of the word is known as Little endian address.

Big Endian Address: A lower byte address of the word is specified in most significant byte of the word is known as big endian address.

2.2 Store:

- The instruction complementary to load is traditionally called store; it copies data from a register to memory. The format of a store is similar to that of a load.
- The actual MIPS name is sw, standing for store word.

Example:

Assume variable h is associated with register $s2 and the base address of the array A is in $s3. What is the MIPS assembly code for the C assignment statement below?

A[12] = h + A[8];

Solution:

```
lw $t0,32($s3)     # Temporary reg $t0 gets A[8]
add $t0,$s2,$t0    # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into A[12], using 48 (4×12) as the offset and register $s3 as the base register.

```
sw $t0,48($s3)      # Stores h + A[8] back into A[12]
```

3. Constant or Immediate Operands

- The constants would have been placed in memory when the program was loaded. For example, to add the constant 4 to register $s3.
- we could use the code

  ```
  lw $t0, AddrConstant4($s1)        # $t0 = constant 4
  add $s3,$s3,$t0                   # $s3 = $s3 + $t0 ($t0 == 4)
  ```

- This quick add instruction with one constant operand is called add immediate or addi. To add 4 to register $s3,We just write

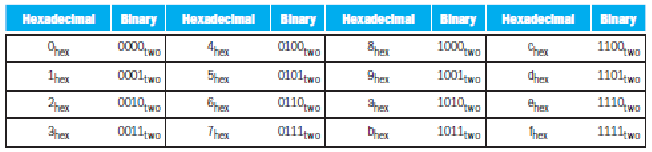
  ```
  addi $s3,$s3,4         # $s3 = $s3 +
  4
  ```

- No subtract immediate instruction, Just use a negative constant

  ```
  addi $s2,$s3,-1
  ```

REPRESENTING INSTRUCTIONS IN THE COMPUTER

- Since registers are referred to in instructions, there must be a convention to map register names into numbers.



Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

Register numbers

$t0 – $t7 are reg's 8 – 15

$t8 – $t9 are reg's 24 – 25

$s0 – $s7 are reg's 16 – 23

Instruction format: A form of representation of an instruction composed of fields of binary numbers.

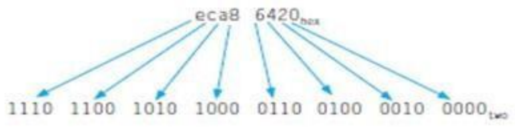
Machine language: Binary representation used for communication within a computer system.

Hexadecimal: Numbers in base 16. The following table shows Hexadecimal to binary conversion.

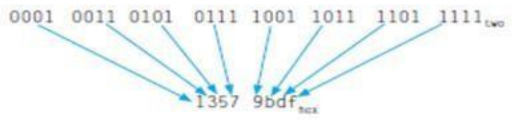
Example:

Convert the following hexadecimal and binary numbers into the other base:

1. eca8 6420hex



2. 0001 0011 0101 0111 1001 1011 1101 1111two



Format Types:

1. R-Format
2. I-Format
3. J-Format

1. R-Type Format:

Opcode: The field that denotes the operation and format of an instruction.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Instruction Fields:

op: operation code (opcode)
rs: first source register number
rt: second source register number
rd: destination register number
shamt: shift amount (00000 for now)
funct: function code (extends opcode)

Example:

add $t0, $s1, $s2

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

2. I-Type Format:

A second type of instruction format is called I-type (for immediate)or I-format and is used by the immediate and data transfer instructions. The fields of I-format are

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

the hardware knows whether to treat the last half of the instruction as three fields(R-type) or as a single field (I-type).

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Example:

Translating MIPS Assembly Language into Machine Language

We can now take an example all the way from what the programmer writes to what the computer executes. If $t1 has the base of the array A and $s2 corresponds to h, the assignment statement

```
A[300] = h + A[300];
```

is compiled into

```
lw   $t0,1200($t1) # Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]
sw   $t0,1200($t1) # Stores h + A[300] back into A[300]
```

What is the MIPS machine language code for these three instructions?

Solution:

Decimal Form:

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Binary Form:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

- The two MIPS instruction formats so far are R and I.
- The first 16 bits are the same: both contain an op field, giving the base operation; an rs field, giving one of the sources; and the rt field, which specifies the other source operand, except for load word, where it specifies the destination register.
- R-format divides the last 16 bits into an rd field, specifying the destination register; the shamt field, and the funct field, which specifies the specific operation of R-format instructions.
- I-format combines the last 16 bits into a single address field.

3. J-Format:

j target- J-type is short for "jump type". The format of an J-type instruction looks like:

Opcode	Address

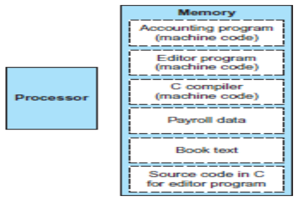
The semantics of the j instruction (j means jump) are:

PC <- PC31-28 IR25-0 00

where PC is the program counter, which stores the current address of the instruction being executed. You update the PC by using the upper 4 bits of the program counter, followed by the 26 bits of the target (which is the lower 26 bits of the instruction register), followed by two 0's, which creates a 32 bit address.

The stored-program concept:

- Stored programs allow a computer that performs by loading memory with programs and data and to begin executing at a given location in memory.



Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

What MIPS instruction does this represent? Choose from one of the four options below.

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. sub $t0, $t1, $t2
2. add $t2, $t0, $t1
3. sub $t2, $t1, $t0
4. sub $t2, $t0, $t1

LOGICAL OPERATIONS

The packing and unpacking of bits into words. These instructions are called logical operations.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

1. SHIFT:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

shamt: how many positions to shift

Shift left logical

Shift left and fill with 0 bits
sll by i bits multiplies by 2i

Shift right logical

Shift right and fill with 0 bits
srl by i bits divides by 2i (unsigned only)

- For example, if register $s0contained

  0000 0000 0000 0000 0000 0000 0000 $1001_{two} = 9_{ten}$
- and the instruction to shift left by 4 was executed, the new value would

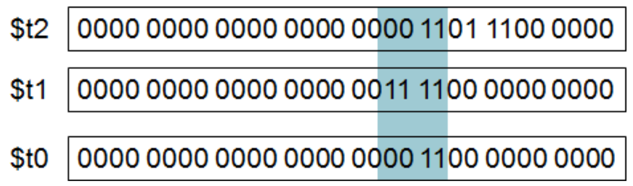
  be: 0000 0000 0000 0000 0000 0000 1001 $0000_{two} = 144_{ten}$
- The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called shift left logical (sll) and shift right logical (srl).
- The following instruction performs the operation above, assuming that the original value was in register $s0 and the result should go in register $t2:

  sll $t2,$s0,4 # reg $t2 = reg $s0 << 4 bits
- The encoding of sll is 0 in both the op and funct fields, rd contains 10 (register $t2), rt contains 16 (register $s0), and shamt contains 4. The rs field is unused and thus is set to 0.Shifting left by i bits gives the same result as multiplying by 2i.

2. AND

- A logical bit by-bit operation with two operands that calculates a 1 only if there is a 1 in both operands.

  Ex: and $t0, $t1, $t2

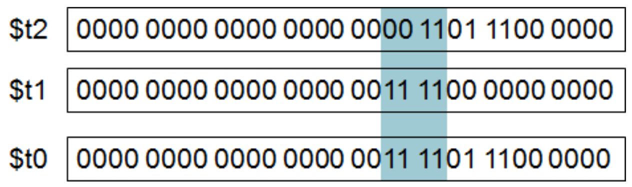


$t2 0000 0000 0000 0000 0000 1101 1100 0000
$t1 0000 0000 0000 0000 0011 1100 0000 0000
$t0 0000 0000 0000 0000 0000 1100 0000 0000

3. OR

- A logical bit-by bit operation with two operands that calculates a 1 if there is a 1 in either operand.

  Ex: or $t0,$t1,$t2



$t2 0000 0000 0000 0000 0000 1101 1100 0000
$t1 0000 0000 0000 0000 0011 1100 0000 0000
$t0 0000 0000 0000 0000 0011 1101 1100 0000

4. NOT

- A logical bit-by bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

```
$t1   0000 0000 0000 0000 0011 1100 0000 0000

$t0   1111 1111 1111 1111 1100 0011 1111 1111
```

5. NOR

- A logical bit-by bit operation with two operands that calculates the NOT of the OR of the two operands. That is, it calculates a 1 only if there is a 0 in both operands. If one operand is zero, then it is equivalent to NOT:

  A NOR 0= NOT (A OR 0) = NOT (A).
- If the register $t1 is unchanged from the preceding example and register $t3has the value 0, the result of the MIPS instruction \

```
nor $t0,$t1,$t3                    # reg $t0 = ~ (reg $t1 | reg $t3)
```

- register $t0

  1111 1111 1111 1111 1100 0011 1111 1111$_{two}$

<u>INSTRUCTIONS FOR MAKING DECISIONS –CONTROL OPERATIONS</u>

- It is ability to make decisions. Based on the input data and the values created during computation.
- The first instruction is **beq register1, register2, L1**
- This instruction means go to the statement labeled L1 if the value in register1 equals the value in register2. The mnemonic beq stands for branch if equal.
- The second instruction is **bne register1, register2, L1**
- It means go to the statement labeled L1 if the value in register1 does not equal the value in register2. The mnemonic bne stands for branch if not equal. These two instructions are traditionally called conditional branches.

Compiling if-then-else into Conditional Branches

- In the following code segment, f, g, h, i, and j are variables. If the five variables f through j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for this C if statement?

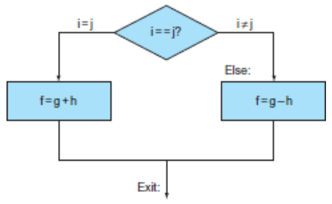
  if (i == j) f = g + h; else f = g – h;

MIPS Code

```
bne $s3,$s4,else              # go to Else if i ≠ j
add $s0,$s1,$s2               # f = g + h (skipped if i ≠ j)
j Exit                        # go to Exit
else: sub $s0,$s1,$s2         # f = g – h (skipped if i =
j) Exit:
```

Conditional Branch

- An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.



Loops

- Decisions are important both for choosing between two alternatives—found in if statements and for iterating a computation found in loops. The same assembly instructions are the building blocks for both cases.

Compiling a while Loop in C

- Here is a traditional loop in C:

```
while (save[i] == k)
i += 1;
```

- Assume that i and k correspond to registers $s3 and $s5 and the base of the array save is in $s6. What is the MIPS assembly code corresponding to this C segment?

MIPS Code:

```
Loop: sll $t1,$s3,2          # Temp reg $t1 = i * 4
 add $t1,$t1,$s6             # $t1 = address of save[i]
 lw $t0,0($t1)               # Temp reg $t0 = save[i]
 bne $t0,$s5, Exit           # go to Exit if save[i] ≠ k
 addi $s3,$s3,1              # i = i + 1
 j Loop                      # go to Loop
 Exit:
```

Basic Block

- A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

Set on Less Than Instruction

- Compares two registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0.
- The MIPS instruction is called set on less than, or slt. For example, slt $t0, $s3, $s4   # $t0 = 1 if $s3 < $s4
- It means that register $t0 is set to 1 if the value in register $s3 is less than the value in register $s4; otherwise, register $t0 is set to 0.

- Constant operands are popular in comparisons, so there is an immediate version of the set on less than instruction. To test if register $s2 is less than the constant10, we can just write

  slti $t0,$s2,10   # $t0 = 1 if $s2 < 10

Signed versus Unsigned Comparison

- Suppose register $s0 has the binary number

  1111 1111 1111 1111 1111 1111 1111 1111$_{two}$ and that register $s1 has the binary number 0000 0000 0000 0000 0000 0000 0000 0001$_{two}$

  What the values are of registers $t0 and $t1 after these two instructions?

Solution

  slt $t0, $s0, $s1   # signed comparison

  sltu $t1, $s0, $s1   # unsigned comparison

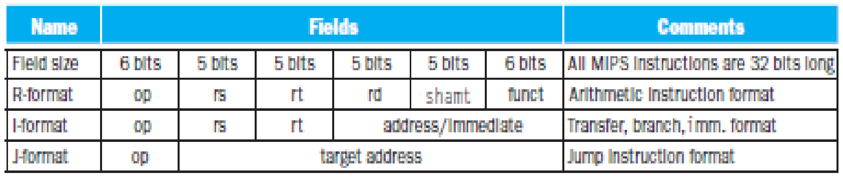
- The value in register $s0 represents -1_{ten} if it is an integer and 4,294,967,295$_{ten}$if it is an unsigned integer.
- The value in register $s1 represents 1_{ten} in either case. Then register $t0 has the value 1, since $-1_{ten}<1_{ten}$, and register $t1 has the value 0, since $4{,}294{,}967{,}295_{ten}>1_{ten}$.

<u>**MIPS ADDRESSING MODE SUMMARY**</u>

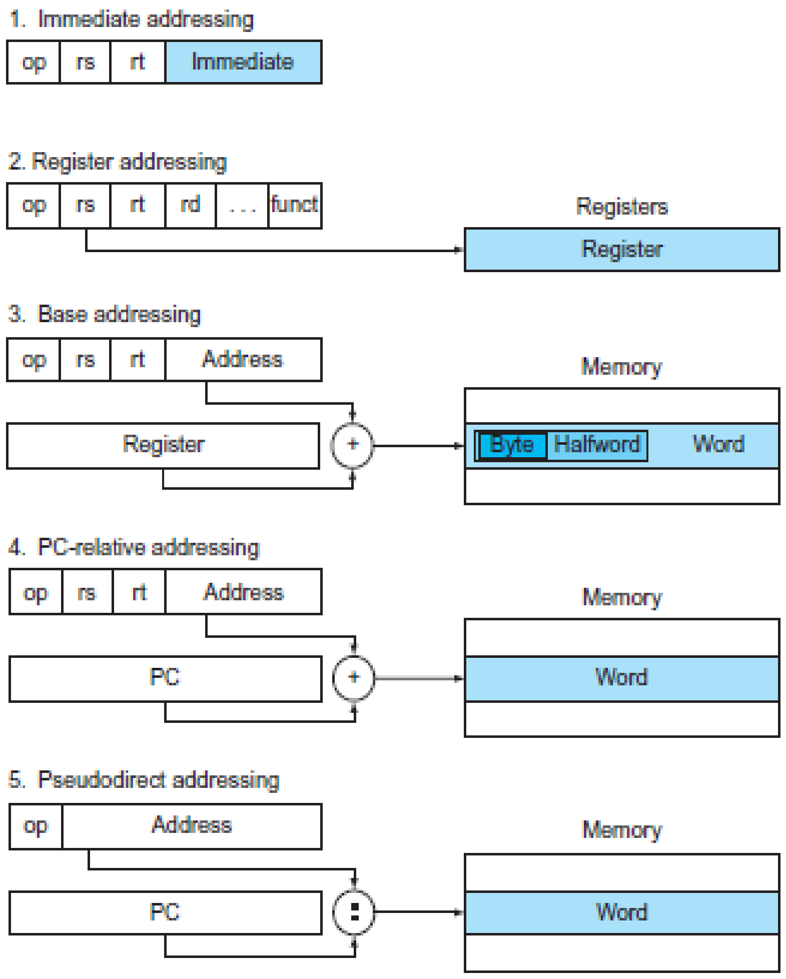
Multiple forms of addressing are generically called **addressing modes.** The following diagram shows how operands are identified for each addressing mode.

1. **Immediate addressing**, where the operand is a constant within the instruction itself
2. **Register addressing**, where the operand is a register
3. **Base or displacement addressing,** where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
4. **PC-relative addressing**, where the branch address is the sum of the PC and a constant in the instruction
5. **Pseudo direct addressing**, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

The following diagram shows all the MIPS instruction formats.



Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

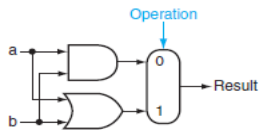


UNIT – II ARITHMETIC FOR COMPUTERS

Addition and Subtraction – Multiplication – Division – Floating Point Representation – Floating Point Operations – Subword Parallelism

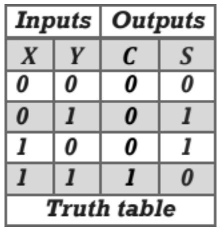
ALU:

- Arithmetic Logic Unit (ALU). Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.
- The arithmetic logic unit (ALU) is the brawn of the computer.



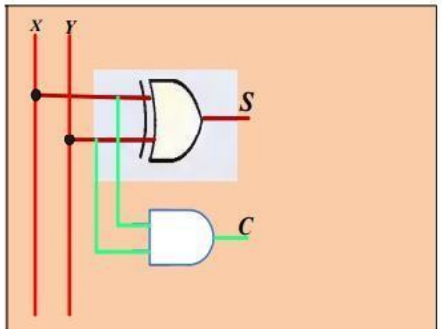
Half Adder

Half Adder: is a combinational circuit that performs the addition of two bits, this circuit needs two binary inputs and two binary outputs.



Inputs		Outputs	
X	*Y*	*C*	*S*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Truth table



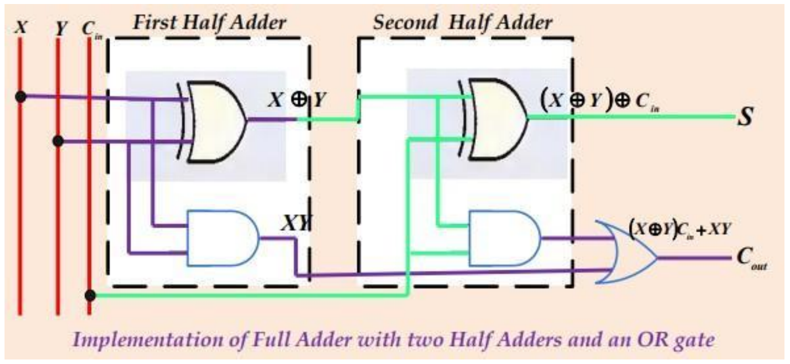
Where S is the sum and C is the carry.

$$\begin{cases} S = X \oplus Y \\ C = XY \end{cases} 2\}$$

(Using XOR and AND Gates)

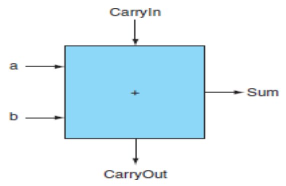
Full Adder

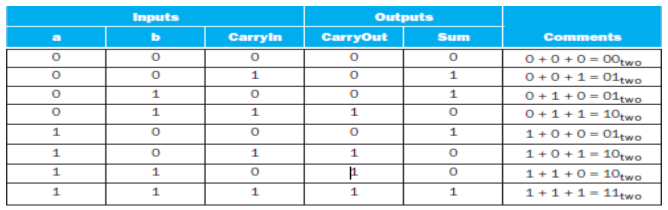
- Full Adder is a combinational circuit that performs the addition of three bits (two significant bits and previous carry).
- It consists of three inputs and two outputs, two inputs are the bits to be added, the third input represents the carry form the previous position.
- The full adder is usually a component in a cascade of adders, which add 8, 16, etc, binary numbers.
- An adder must have two inputs for the operands and a single-bit output for the sum and the second output to pass on the carry, called CarryOut.
- The CarryOut from the neighbor adder must be included as an input, we need a third input. This input is called CarryIn.



Implementation of Full Adder with two Half Adders and an OR gate

$$\begin{Bmatrix} S = C_{in} \oplus (X \oplus Y) \\ C_{out=} C_{in} \cdot (X \oplus Y) + XY \end{Bmatrix}$$

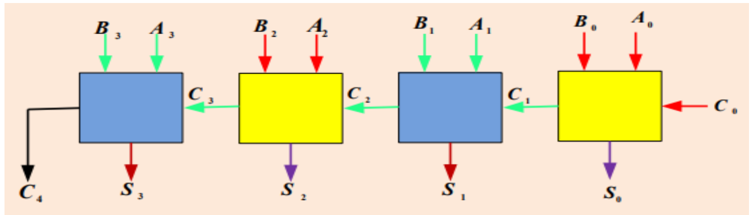




Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

Binary Adder (Asynchronous Ripple-Carry Adder)

- A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers.
- A binary adder can be constructed with full adders connected in¬ cascade with the output carry form each full adder connected to the input carry of the next full adder in the chain.
- The four-bit adder is a typical example of a standard component .It can¬ be used in many application involving arithmetic operations.
- The input carry to the adder is and it ripples through the full adders to the output carry bit binary adder requires full adders



Four-bit Adder (Ripple Carry Adder)

CARRY-LOOK AHEAD ADDER

- Fast adder circuit must speed up the generation of carry signals. Carry look ahead logic uses the concepts of generating and propagating carries. Where Si is the sum and Ci+1 is the carry out.

➢ The ***carry propagation time*** is an important attribute of the adder because it limits the speed with which two numbers are added.

➢ To reduce the carry propagation delay time:

1) Employ faster gates with reduced delays.
2) Employ the principle of Carry Lookahead Logic.

Proof: (using carry lookahead logic)

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

The output sum and carry are:

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

✓ G_i-called a ***carry generate***, and it produces a carry of 1 when both A_i and B_i are 1.

✓ P_i-called a ***carry propagate***, it determines whether a carry into stage i will propagate into stage $i+1$.

✓ The ***Boolean function*** for the carry outputs of each stage and substitute the value of each C_i form the previous equations:

$$\left\{\begin{array}{l} C_0 = \textit{input carry} \\ C_1 = G_0 + P_0 C_0 \\ C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0) \\ \quad = G_1 + P_1 G_0 + P_1 P_0 C_0 \\ C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1 G_0 + P_1 P_0 C_0) \\ \quad = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \end{array}\right\}$$

➢ The three Boolean functions C_1, C_2 and C_3 are implemented in the ***carry lookahead generator.***

The two level-circuit for the output carry C_4 is not shown, it can be easily derived by the equation.

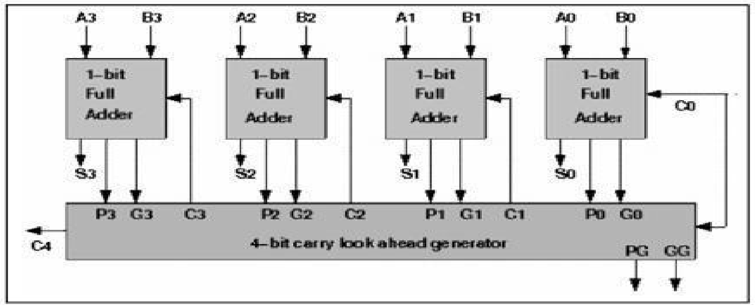
➢ C_3 does not have to wait for C_2 and C_1 to propagate, in fact C_3 is propagated at the same time as C_1 and C_2.

Adv:

1. Circuit is simplicity
2. Structure is slightly faster
3. Easy to understand
4. To eliminate inter stage carry delay.

Dadv:

1. Carry look-ahead is expensive

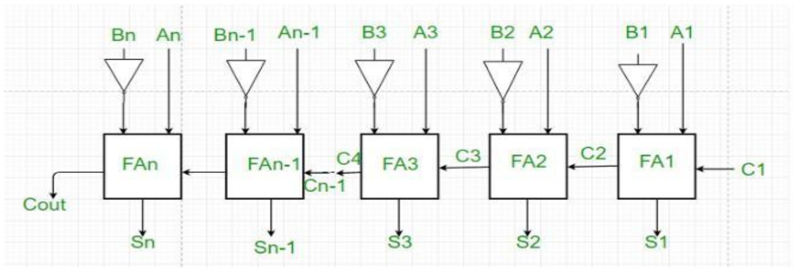


Carry Propagation Delay

The sum and carry output of any stage cannot be produced until the input carry occurs. This leads to a time delay in the addition process.

Parallel Subtractor

- A Parallel Subtractor is a digital circuit capable of finding the arithmetic difference of two binary numbers that is greater than one bit in length by operating on corresponding pairs of bits in parallel.
- The parallel subtractor can be designed in several ways including combination of half and full subtractors, all full subtractors or all full adders with subtrahend complement input.



Advantages of parallel Adder/Subtractor

- The parallel adder/subtractor performs the addition operation faster as compared to serial adder/subtractor.
- Time required for addition does not depend on the number of bits.
- The output is in parallel form i.e all the bits are added/subtracted at the same time.
- It is less costly.

Disadvantages of parallel Adder/Subtractor

- Each adder has to wait for the carry which is to be generated from the previous adder in chain.
- The propagation delay(delay associated with the travelling of carry bit) is found to increase with the increase in the number of bits to be added.

ADDITION AND SUBTRACTION:

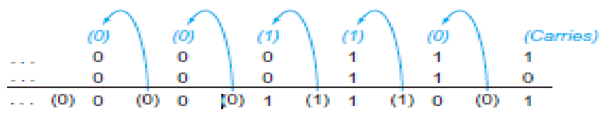
- Digits are added bit by bit from right to left, with carries passed to the next digit to the left.
- Subtraction uses addition. The appropriate operand is simply negated before being added.

Binary addition:

Let's try adding 6_{ten} to 7_{ten} in binary.

```
  0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
+ 0000 0000 0000 0000 0000 0000 0000 0110two = 6ten
---------------------------------------------------
= 0000 0000 0000 0000 0000 0000 0000 1101two = 13ten
```

The following figure shows the sums and carries. The carries are shown in parentheses.



- Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0.
- Hence, the operation for the second digit to the right is 0+1+1.
- This generates a 0 for this sum bit and a carry out of 1.
- The third digit is the sum of 1+1+1, resulting in a carry out of 1 and a sum bit of 1.
- The fourth bit is 1+0+0, yielding a 1 sum and no carry.

Binary subtraction:

- Subtracting 6_{ten} from 7_{ten} can be done directly

```
  0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
– 0000 0000 0000 0000 0000 0000 0000 0110two = 6ten
---------------------------------------------------
= 0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
```

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

```
  0000 0000 0000 0000 0000 0000 0000 0111two = 7ten
+ 1111 1111 1111 1111 1111 1111 1111 1010two = –6ten
---------------------------------------------------
= 0000 0000 0000 0000 0000 0000 0000 0001two = 1ten
```

When overflow cannot occur in addition and subtraction? **Case: 1**

- When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, -10+4=-6.
- Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

Case: 2

- When the signs of the operands are the same, overflow cannot occur. To see this, remember that c – a= c + (-a) because we subtract by negating the second operand and then add.
- Therefore, when we subtract operands of the same sign we end up by adding operands of different signs.

When overflow can occur in addition and subtraction?

Case: 1

- Overflow occurs when adding two positive numbers and the sum is negative

Case: 2

- Overflow occurs when adding two negative numbers and the sum is positive. This spurious sum means a carry out occurred into the sign bit.

Case: 3

- Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result.

Case: 4

- When we subtract a positive number from a negative number and get a positive result. Such a ridiculous result means a borrow occurred from the sign bit.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

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

Exception:

- Exception also called interrupt on many computers. An unscheduled event that disrupts program execution; used to detect overflow.

Interrupt:

- An exception that comes from outside of the processor.

EPC:

- MIPS include a register called the Exception Program Counter (EPC) to contain the address of the instruction that caused the exception.
- The instruction move from system control (mfc0) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the off ending instruction via a jump register instruction.

MULTIPLICATION

- The first operand is called the multiplicand and the second the multiplier. The final result is called the product.
- If we ignore the sign bits, the length of the multiplication of an n-bit multiplicand and an m-bit multiplier is a product that is n+ m bits long.
- That is, n+ m bits are required to represent all possible products.
- For example, **Multiplying** 1000_{ten} by 1001_{ten}: **Multiplicand** 1000_{ten} **Multiplier** 1001_{ten}

```
            1000ten x 1001ten
            1000
           0000
          0000
         1000
Product  1001000ten
```

Case: 1

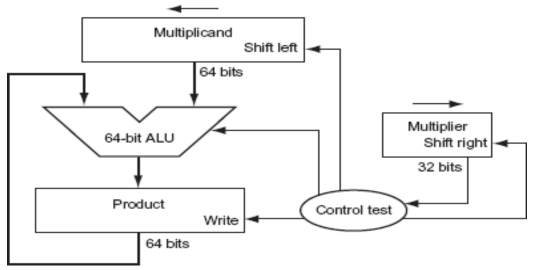
- Just place a copy of the multiplicand (1 x multiplicand) in the proper place if the multiplier digit is a 1.

Case: 2

- Place 0 (0 x multiplicand) in the proper place if the digit is 0.

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.
- 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.



Step: 1

- The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register.

Step: 2

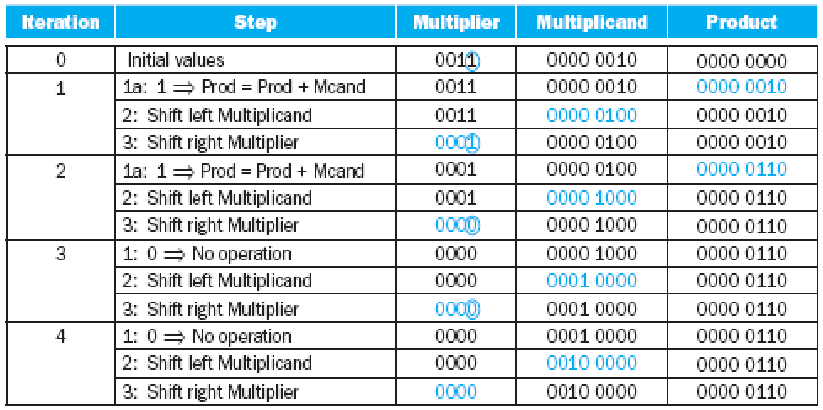
- If the least significant bit of the multiplier is 1, add the multiplicand to the product.
- If not, go to the next step. Shift left the multiplicand register by 1 bit.

Step: 3

- Then shift right the multiplier register by 1 bit. These three steps are repeated 32 times to obtain the product.
- If each step took a clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers.

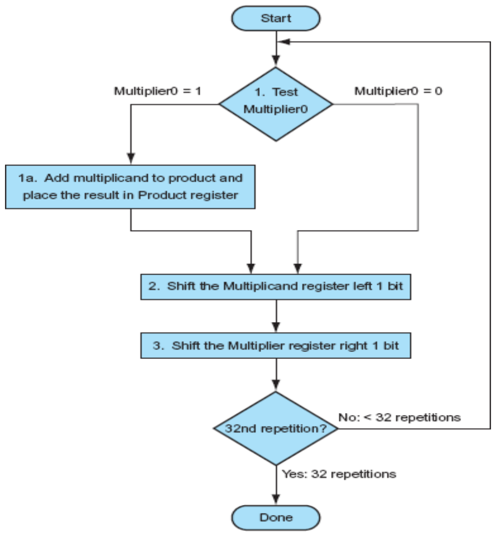
Example:

Using 4-bit numbers to save space, multiply 2ten x 3ten, or 0010two x 0011two.



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

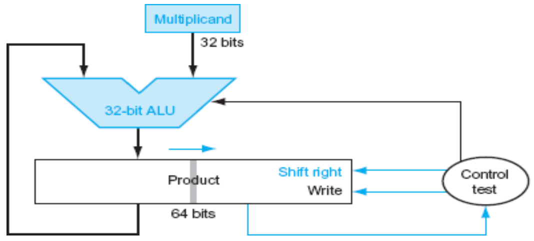
Flowchart:


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Refined version of the multiplication hardware:

- Comparing with the first algorithm 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.

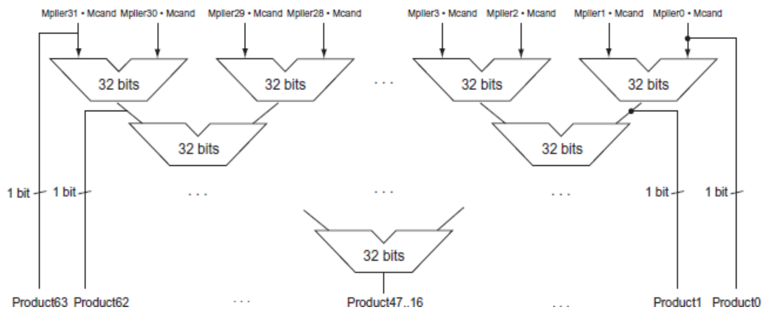


Signed Multiplication

- First convert the multiplier and multiplicand to positive numbers and then remember the original signs.
- The algorithms should then be run for 31 iterations, leaving the signs out of the calculation.

Faster Multiplication

- Hardware designers can now build much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by the 32 multiplier bits.
- Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier:
- One input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.
- To connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 32 high.
- Rather than use a single 32-bit adder 31 times, this hardware "unrolls the loop" to use 31adders and then organizes them to minimize delay.

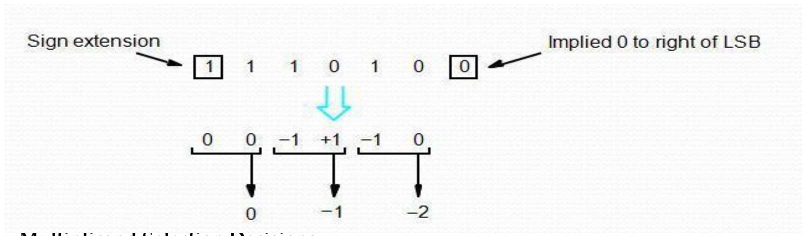


BOOTH'S BIT-PAIR RECODING OF THE MULTIPLIER.

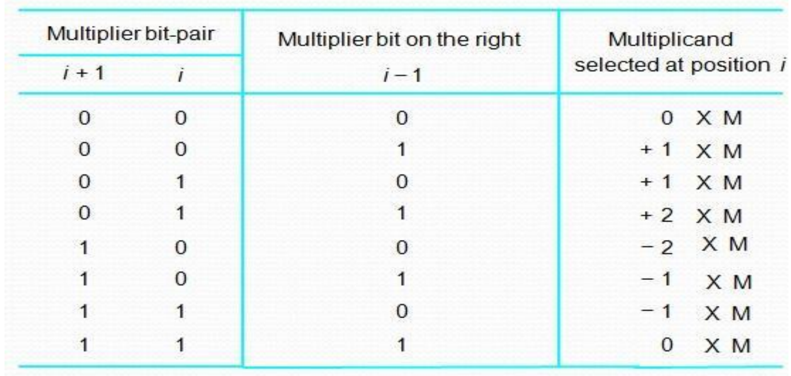
A=+13 (Multiplicand) AND B= -6 (Multiplier)

Bit-pair recoding halves the maximum number of summands (versions of the multiplicand).

Example

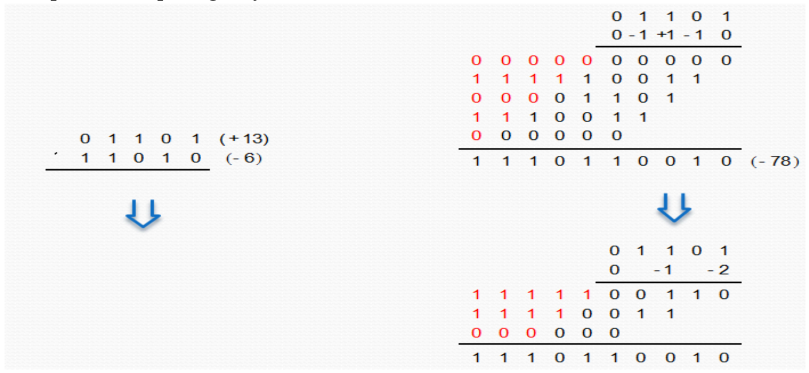


Multiplicand Selection Decisions



Multiplier bit-pair $i+1$	Multiplier bit-pair i	Multiplier bit on the right $i-1$	Multiplicand selected at position i
0	0	0	0 X M
0	0	1	+ 1 X M
0	1	0	+ 1 X M
0	1	1	+ 2 X M
1	0	0	− 2 X M
1	0	1	− 1 X M
1	1	0	− 1 X M
1	1	1	0 X M

Multiplication requiring only n/2 summands



BOOTH'S MULTIPLICATION ALGORITHM WITH SUITABLE EXAMPLE

Booth's Algorithm Principle:

- Performs additions and subtractions of the Multiplicand, based on the value of the multiplier bits.
- The algorithm looks at two adjacent bits in the Multiplier in order to decide the operation to be performed.

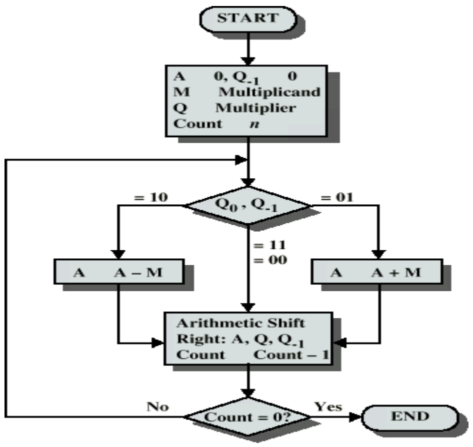
- The Multiplier bits are considered from the least significant bit (right-most) to the most significant bit; by default a 0 will be considered at the right of the least significant bit of the multiplier.
- If Multiplicand has Md bits and Multiplier has Mp bits, the result will be stored in a Md+Mp bit register and will be initialised with 0s
- As repeated operations and shifts are performed on partial results, the result register is the accumulator (A).
- Booth‘s algorithm gives a procedure for multiplying signed binary integer. It is based on the fact that strings of 0‘s in the multiplier require no addition but only shifting and a string of 1‘s in the multiplier require both operations.

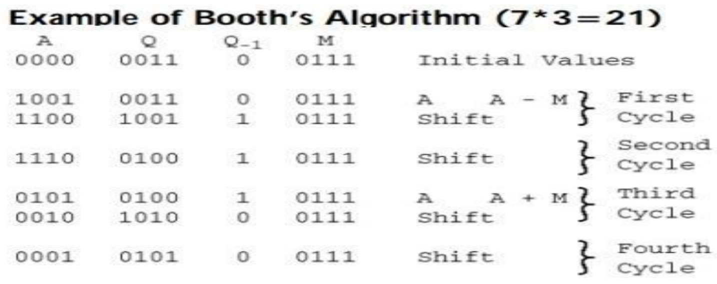
Algorithm

The Q0 bit of the register Q and Q-1 is examined:

- If two bits are the same (11 or 00), then all of the bits of the A, Q and Q1 registers are shifted to the right 1 bit. This shift is called arithmetic shift right.
- If two bits differ i.e., whether 01, then the multiplicand is adder or 10, then the multiplicand is subtracted from the register A. after that, right shift occurs in the register A, Q and Q1.

Flowchart of Booth's Algorithm for 2's complement multiplication





Example of Booth's Algorithm (7*3=21)

A	Q	Q-1	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A – M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

A DIVISION ALGORITHM AND HARDWARE:

Dividend:

- A number being divided is called dividend.

Divisor:

- A number that the dividend is divided by is called divisor.

Quotient:

- It is called the primary result of a division.
- A number that when multiplied by the divisor and added to the remainder produces the dividend is known as quotient.

Remainder:

- It is the secondary result of a division.
- A number that when added to the product of the quotient and the divisor produces the dividend is known as remainder.

The example is dividing $1{,}001{,}010_{ten}$ by 1000_{ten}:

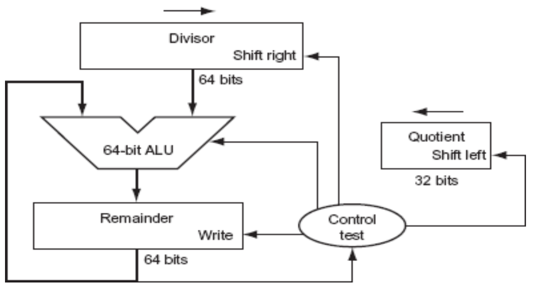
```
                    1001ten      Quotient
Divisor 1000ten ⟌1001010ten      Dividend
                 -1000
                    10
                    101
                    1010
                   -1000
                      10ten      Remainder
```

- Divide's two operands, called the dividend and divisor, and the result, called the quotient, are accompanied by a second result, called the remainder.
- Here is another way to express the relationship between the components:

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

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.



Divide algorithm:

Step: 1

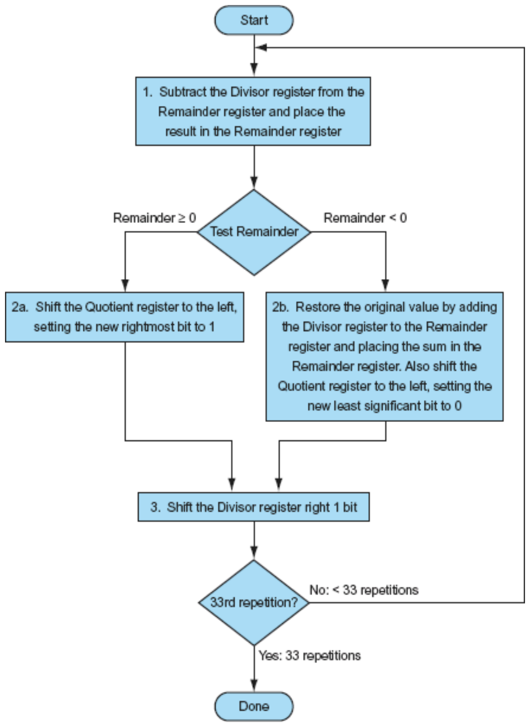
- It must first subtract the divisor register from the Remainder register and place the result in the Remainder register.

Step: 2

- Next we performed the comparison in the set on less than instruction.
- If the result is positive, the divisor was smaller or equal to the dividend, so shift the Quotient register to the left, setting the new rightmost bit to 1.
- If the result is negative, the next step Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register.
- Also shift the Quotient register to the left, setting the new least significant bit to 0

Step: 3

- The divisor is shifted right by 1 bit and then we iterate again.
- The remainder and quotient will be found in their registers after the iterations are complete.

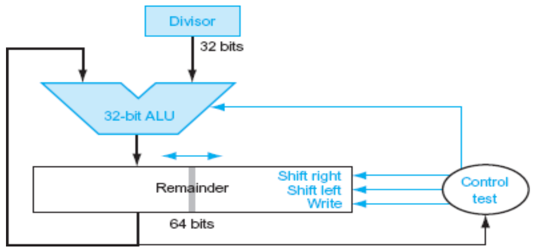

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

Using a 4-bit version of the algorithm to save pages, let's try dividing 7ten by 2ten, or 0000 0111two by 0010two.

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

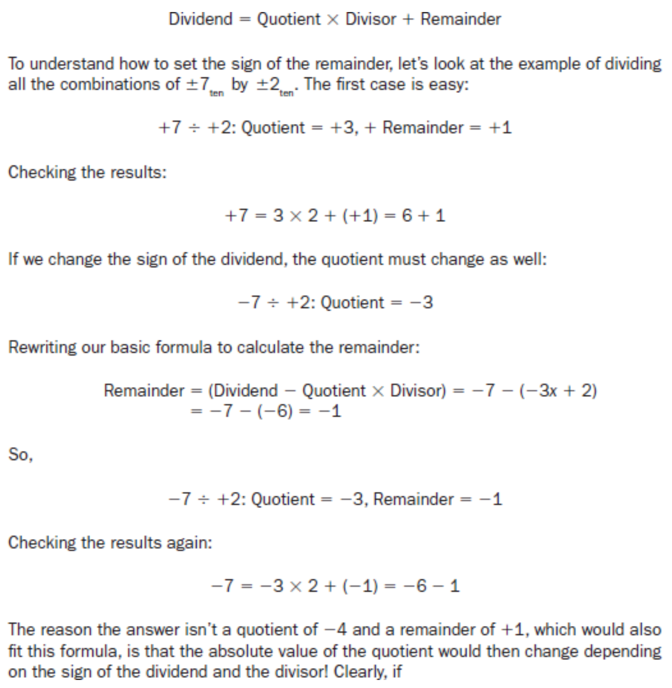
An improved version of the division hardware:

- The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits.
- Compared to above division hardware, 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.



Signed Division

- The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.
- 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 × Divisor + Remainder

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of $\pm 7_{\text{ten}}$ by $\pm 2_{\text{ten}}$. The first case is easy:

$$+7 \div +2\text{: Quotient} = +3,\ +\text{ Remainder} = +1$$

Checking the results:

$$+7 = 3 \times 2 + (+1) = 6 + 1$$

If we change the sign of the dividend, the quotient must change as well:

$$-7 \div +2\text{: Quotient} = -3$$

Rewriting our basic formula to calculate the remainder:

$$\begin{aligned}\text{Remainder} &= (\text{Dividend} - \text{Quotient} \times \text{Divisor}) = -7 - (-3x + 2)\\ &= -7 - (-6) = -1\end{aligned}$$

So,

$$-7 \div +2\text{: Quotient} = -3,\ \text{Remainder} = -1$$

Checking the results again:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

The reason the answer isn't a quotient of −4 and a remainder of +1, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the dividend and the divisor! Clearly, if

Rule:

- The dividend and remainder must have the same signs, no matter what the signs of the divisor and quotient.

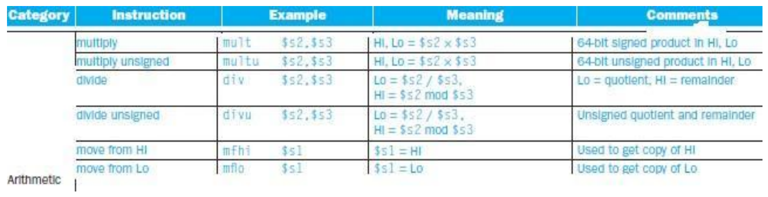
$$-(x \div y) \neq (-x) \div y$$

We calculate the other combinations by following the same rule:

$$+7 \div -2\text{: Quotient} = -3,\ \text{Remainder} = +1$$
$$-7 \div -2\text{: Quotient} = +3,\ \text{Remainder} = -1$$

Faster Division

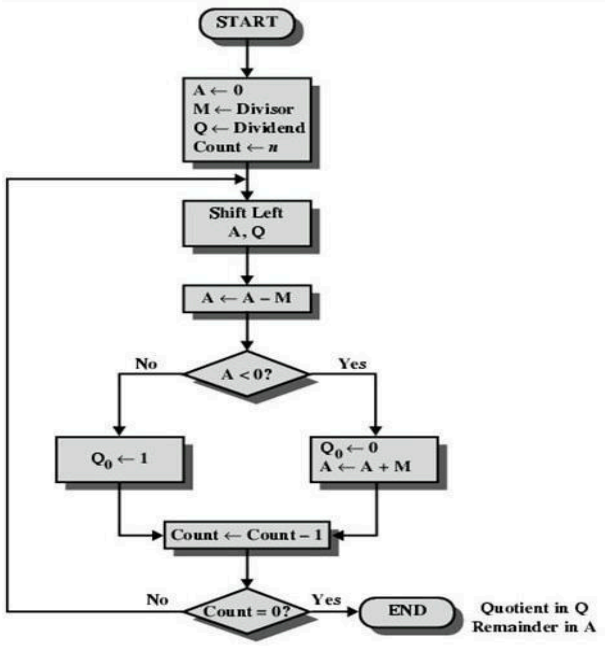
- There are techniques to produce more than one bit of the quotient per step.
- The SRT division technique tries to predict several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder.
- These algorithms use 6 bits from the remainder and 4 bits from the divisor to index a table that determines the guess for each step.



Category	Instruction	Example		Meaning	Comments
	multiply	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
Arithmetic	move from Lo	mflo	\$s1	\$s1 = Lo	Used to get copy of Lo

Restoring Division Algorithm

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Then the content of register A and Q is shifted right as if they are a single unit
- **Step-3:** Then content of register M is subtracted from A and result is stored in A
- **Step-4:** Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M
- **Step-5:** The value of counter n is decremented
- **Step-6:** If the value of n becomes zero we get of the loop otherwise we repeat fro step 2
- **Step-7:** Finally, the register Q contain the quotient and A contain remainder



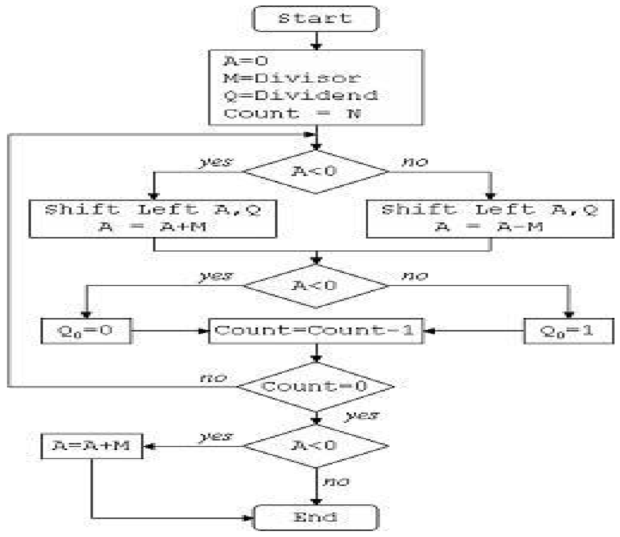
Example: 8 divide by 3=2 (2/3)

The quotient (0010)2 = 2 is in register Q, and the reminder (0010)2 = 2 is in register A.

	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		000.
$A = [A] - [M]$	+	1101		
$A < 0$		1110		0000
$A = [A] + [M]$	+	0011		
		0001		0000
left shift A/Q		0010		000.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0000
$A = [A] + [M]$	+	0011		
		0010		0000
left shift A/Q		0100		000.
$A = [A] - [M]$	+	1101		
$A > 0$		0001		0001
left shift A/Q		0010		001.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0010
$A = [A] + [M]$	+	0011		
		0010		0010

Non-Restoring Division Algorithm

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Check the sign bit of register A
- **Step-3:** If it is 1 shift left content of AQ and perform A = A+M, otherwise shift left AQ and perform A = A-M (means add 2's complement of M to A and store it to A)
- **Step-4:** Again the sign bit of register A
- **Step-5:** If sign bit is 1 Q[0] become 0 otherwise Q[0] become 1 (Q[0] means least significant bit of register Q)
- **Step-6:** Decrements value of N by 1
- **Step-7:** If N is not equal to zero go to **Step 2** otherwise go to next step
- **Step-8:** If sign bit of A is 1 then perform A = A+M
- **Step-9:** Register Q contain quotient and A contain remainder



	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		000.
A=[A]-[M]	+	1101		
$A < 0$		1110		0000
left shift A/Q		1100		000.
$A = [A] + [M]$	+	0011		
$A < 0$		1111		0000
left shift A/Q		1110		000.
$A = [A] + [M]$	+	0011		
$A > 0$		0001		0001
left shift A/Q		0010		001.
$A = [A] - [M]$	+	1101		
$A < 0$		1111		0010
$A = [A] + [M]$	+	0011		
		0010		0010

FLOATING POINT

Normalized number:

A number in floating-point notation that has no leading 0^s is known as normalized number. i.e., a number start with a single nonzero digit.

For example, $1.0_{ten} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{ten} \times 10^{-8}$ and $10.0_{ten} \times 10^{-10}$ are not.

Binary numbers in scientific notation:

To keep a binary number in normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. $1.0_{two} \times 2^{-1}$

Floating-Point Representation

Floating point:

Computer arithmetic that represents numbers in which the binary point is not fixed.

Fraction:

The value, generally between 0 and 1, placed in the fraction field. The fraction is also called the mantissa.

Exponent:

In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

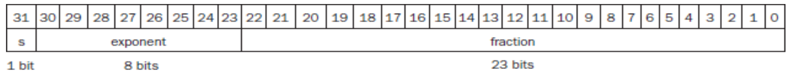
Single precision:

A floating-point value represented in a single 32-bit word. Floating-point numbers are usually a multiple of the size of a word.

Where s is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number.

F involves the value in the fraction field and E involves the value in the exponent field.

Format:



In general, floating-point numbers are of the form

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

Overflow:

A situation in which a positive exponent becomes too large to fit in the exponent field is known as overflow.

Underflow:

A situation in which a negative exponent becomes too large to fit in the exponent field is known as underflow.

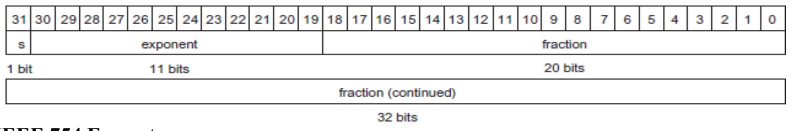
Double precision:

One way to reduce chances of underflow or overflow is called double, and operations on doubles are called double precision floating-point arithmetic.

It has a larger exponent. A floating-point value represented in two 32-bit words.

Where s is still the sign of the number, exponent is the value of the 11-bit exponent field, and fraction is the 52-bit number in the fraction field.

Format:



IEEE 754 Format:

MIPS double precision allows numbers almost as small as $2.0_{ten} \times 10^{+308}$ and almost as large as $2.0_{ten} \times 10^{-308}$.

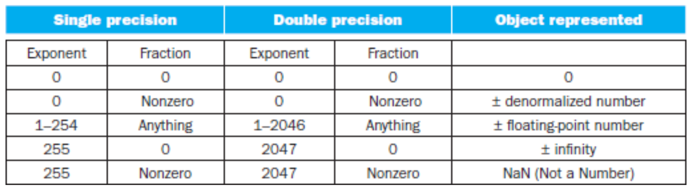
Although double precision does increase the exponent range.

Its primary advantage is its greater precision because of the much larger fraction.

IEEE 754 makes the leading 1-bit of normalized binary numbers implicit.

Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1+52).

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$



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

For example, $1.0_{two} \times 2^{-1}$ would be represented as

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The value $1.0_{twn} \times 2^{+1}$ would look like the smaller binary number

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The desirable notation must therefore represent the most negative exponent as $00 \ldots 00_{two}$ and the most positive as $11 \ldots 11_{two}$.

This convention is called biased notation, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

IEEE 754 uses a bias of 127 for single precision, so an exponent of -1 is represented by the bit pattern of the value -1+127$_{ten}$, or 126$_{ten}$=0111 1110$_{two}$, and +1 is represented by 1+127, or 128$_{ten}$ = 1000 0000$_{two}$.

The exponent bias for double precision is 1023. Biased exponent means that the value represented by a floating-point number is really

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

The range of single precision numbers is then from as small as

$$\pm 1.00000000000000000000000_{two} \times 2^{-126}$$

to as large as

$$\pm 1.11111111111111111111111_{two} \times 2^{+127}.$$

Example: 1

Floating-Point Representation

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2_{ten}$$

It is also represented by the binary fraction

$$-11_{two}/2^2_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$ yields

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126-127)}$$

The single precision binary representation of -0.75_{ten} is then

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022-1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

32 bits

Converting Binary to Decimal Floating Point

What decimal number is represented by this single precision float?

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

The sign bit is 1, the exponent field contains 129, and the fraction field contains $1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$\begin{aligned}(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0\end{aligned}$$

FLOATING-POINT ADDITION

Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Example: 1

Perform floating point addition for the following numbers.

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

Step 1:

Compare the exponent of both the operands.

If it equal add the two operand (significand) .If it is not equal then increase the smaller exponent.

i.e., shift the smaller number to the right until its exponent would match the larger exponent. As per our example

$\mathbf{1.610_{ten} \times 10^{-1}}$= $0.1610_{ten} \times 10^0 = 0.01610_{ten} \times 10^1$

But we can represent only four decimal digits so, after shifting, the number is really

$0.016_{ten} \times 10^1$

Step 2:

Now add the Significand

$$\begin{array}{r} 9.999 \times 10^1 \\ 0.016 \times 10^1 \\ \hline 10.015 \times 10^1 \end{array}$$

Step 3:

Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent.

This sum is not in normalized scientific notation, so we need to adjust it: $10.015 \times 10^1 = 1.0015 \times 10^2 = 1.0015 \times 10^2$

Whenever the exponent is increased or decreased, we must check for overflow or underflow. i.e., we must make sure that the exponent still fits in its field.

Step 4:

Round the significand to the appropriate number of bits.

If the sum may no longer be normalized and we would need to perform step 3 again.

$\mathbf{1.002_{ten} \times 10^2}$

Perform floating point addition for the following numbers.

0.5_{ten} and -0.4375_{ten}

Solution:

Assuming that we keep 4 bits of precision. 0.5_{ten}

Operand 1: Convert the operands to binary

0.5 x 2 =1.0

Scientific Notation

0.1×2^0

Normalizing the above value

1.0×2^{-1}

Operand 2: Convert the operands to binary -0.4375_{ten}

0.4375 x 2 = 0.8750

0.8750 x 2 = 1.7500

0.7500 x 2 = 1.5000

1.5000 x 2 = 1.0000

Scientific Notation

0.0111×2^0

Normalizing the above value

1.110×2^{-2}

Step 1:

The significand of the number with the lesser exponent ($-1.110_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

Step 2:

Add the significands

$$\begin{array}{rl} & 1.000 \times 2^{-1} \\ - & 0.111 \times 2^{-1} \text{ [Subtraction]} \\ \hline & 0.001 \times 2^{-1} \end{array}$$

Step 3: Normalize the sum and checking for overflow or underflow $0.001 \times 2^{-1} = 1.0 \times 2^{-4}$

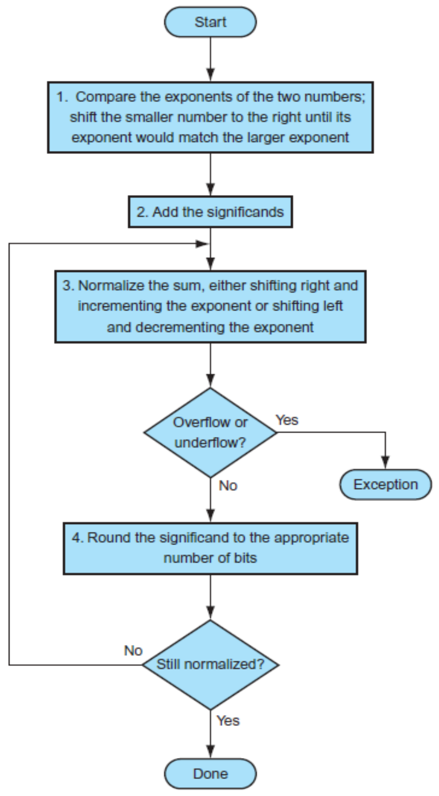
Step 4: Round the sum **1.000×2^{-4}**

Then convert the sum to decimal

$1.000 \times 2^{-4} = 0.0001_{two}$

$1 / 2^4 = 1/16_{ten} = 0.0625_{ten}$

Flowchart:


Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much.

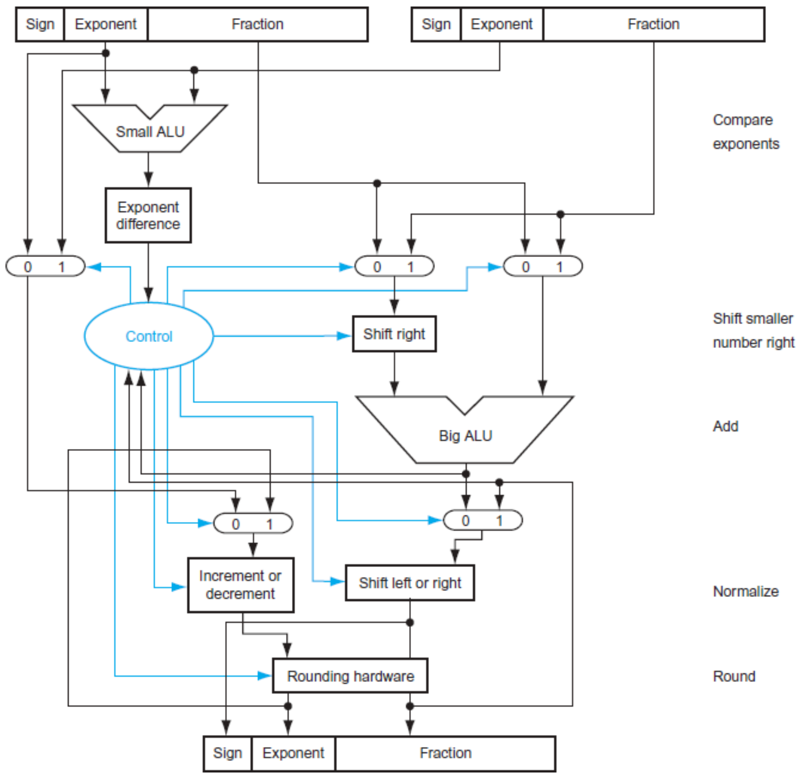
This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number.

The smaller significand is shifted right, and then the significands are added together using the big ALU.

The normalization step then shifts the sum left or right and increments or decrements the exponent.

Rounding then creates the final result, which may require normalizing again to produce the actual final result.

Block Diagram:



FLOATING-POINT MULTIPLICATION

Example: 1

Multiplying decimal numbers in scientific notation:

$1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$

Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1:

We calculate the exponent of the product by simply adding the exponents of the operands together:

New exponent = 10 + (-5) = 5

Let's do this with the biased exponents as well to make sure we obtain the same result:

10 + 127 = 137, and -5 + 127 = 122, so New exponent = 137 +122=259

This result is too large for the 8-bit exponent field.

The problem is with the bias because we are adding the biases as well as the exponents.

New exponent= (10+127)+(-5+127)=(5+2X127)=259

To get the correct biased sum when we add biased numbers, we must subtract the bias from the sum.

New exponent=137+122-127=259-127=132= (5+127) and 5 is indeed the exponent we calculated initially.

Step 2:

Next comes the multiplication of the significands:

```
         1.110ten
    X    9.200ten

         0000
        0000
       2220
      9990
```

The product is 10212000_{ten}

Assuming that we can keep only three digits to the right of the decimal point, the product is 10.212ten x 105

Step 3:

This product is unnormalized, so we need to normalize it:

$10.212_{ten} \times 10^{5} = 1.0212_{ten} \times 10^{6}$

After the multiplication, the product can be shifted right one digit and adding 1 to the exponent.

At this point, we can check for overflow and underflow. Underflow may occur if both operands are small, that is, if both have large negative exponents.

Step 4:

Round of the Product

$1.021_{ten} \times 10^{6}$

Step 5:

The sign of the product depends on the signs of the original operands.

If they are both the same, the sign is positive; otherwise, it's negative. Hence, the product is

$+1.021_{ten} \times 10^{6}$

Example: 2

Multiple the numbers 0.5_{ten} and -0.4375_{ten}, using the steps in the above algorithm

Binary equivalent of $0.5_{ten} = 1.000 \times 2^{-1}$ and $-0.4375_{ten} = -1.110 \times 2^{-2}$

Step 1 :

Adding the Exponents without bias

-1 + (-2) = -3

Or using the biased representation

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2 : Multiplying the significands

```
      1.000two
  ×   1.110two
      --------
      0000
     1000
    1000
   1000
  ----------
  1110000two
```

The Product is $1.110000_{two} \times 2^{-3}$,but we use 4 bits, so it is $1.110_{two} \times 2^{-3}$

Step 3:

Normalize the product, as per our example it is already normalized one

1.110×2^{-3}

Step 4:

Rounding the product no change

1.110 x 2-3

Step 5:

Since the signs of the original operands differ, make the sign of the product negative.

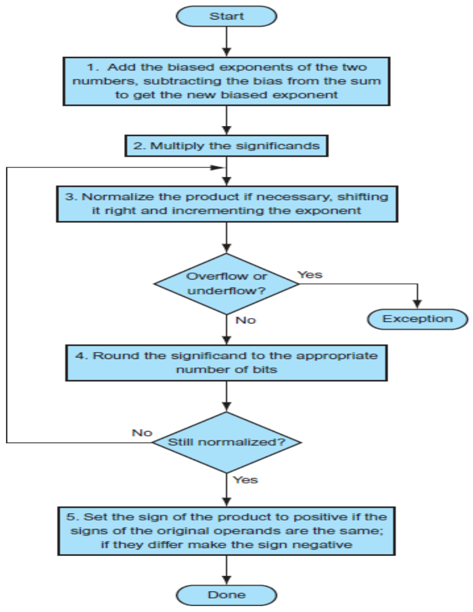
Hence, the product is

-1.110 x 2-3

Converting to decimal to check our results:

-1.110 x 2-3 = - 0.00111 = 1 /8 + 1 / 16 + 1 / 32 = - 0.21875

Flow Chart:


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Guard Bit:

Extra bits kept on the right during intermediate calculations of floating point numbers is called guard bit and it used to improve rounding accuracy.

Round:

Method to make the intermediate floating-point result fit the floating-point format.

The goal is typically to find the nearest number that can be represented in the format.

Sticky Bit:

A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

Subword Parallelism:

By partitioning the 128-bit adder, a processor could use parallelism to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands.

The cost of such partitioned adders was small.

Given that the parallelism occurs within a wide word, the extensions are classified as subword parallelism.

It is also classified under the more general name of data level parallelism.

They have been also called vector or SIMD, for single instruction, multiple data.