KCG COLLEGE OF TECHNOLOGY
AFFILIATED TO ANNA UNIVERSITY | AUTONOMOUS

Department of Computer Science and Engineering

23CS404 - COMPUTER ARCHITECTURE

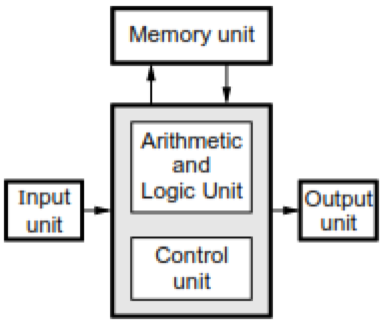
Second Year / Fourth Semester

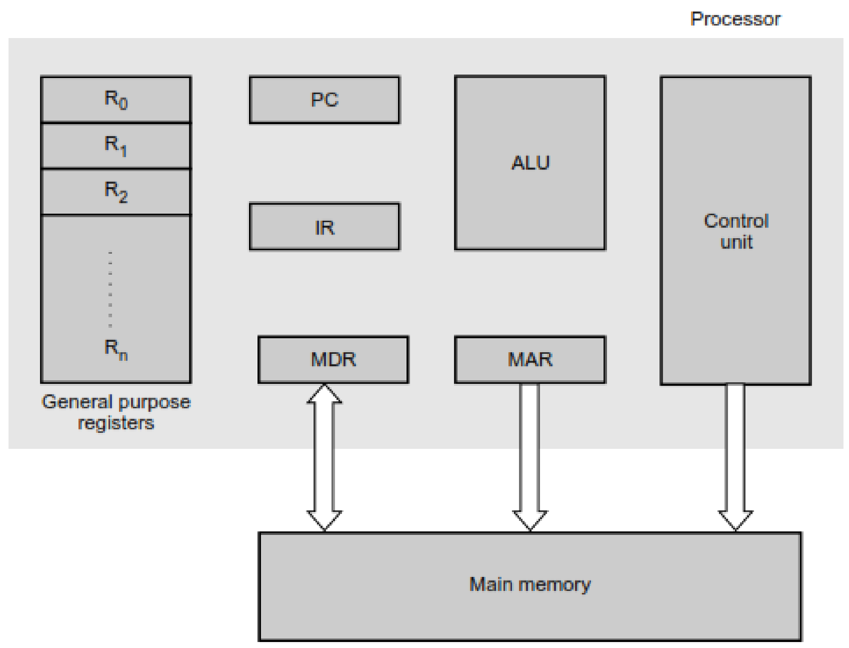
Unit I

QNo	Questions	COs	Bloom's Level
PART - A			
1	Define computer architecture. It is concerned with the structure and behaviour of the computer. It includes the information formats, the instruction set and techniques for addressing memory.	CO1	K1
2	Define computer organization. It describes the function and design of the various units of digital computer that store and process information. It refers to the operational units and their interconnections that realize the architectural specifications.	CO1	K1
3	Define RISC and CISC. RISC, which is an acronym for Reduced Instruction Set Computer, and CISC, short for Complex Instruction Set Computer, refer to the category of the processor, or more accurately, the instruction set architecture (ISA)	CO1	K1
4	Define MIPS. Million instructions per second (MIPS) is an approximate measure of a computer's raw processing power. Since the MIPS measurement doesn't take into account other factors such as the computer's I/O speed or processor architecture.	CO1	K1
5	What are the components of a computer? Input Unit, Output unit, ALU, Memory and control unit	CO1	K1
6	What are the functions of control unit? The memory, arithmetic and logic, and input and output 'units store and process information and perform input and output operations. The operation of these units must be coordinated in some way. This is the task of the control unit.	CO1	K1
7	Define MAR and MDR. The MAR holds the address of the location to be accessed. The MDR contains the data to be written into or read out of the addressed location.	CO1	K1
8	What is program counter and instruction register? The program counter (PC) is another specialized register. It keeps track of the execution of a program. The instruction register (IR) holds the instruction that is currently being executed.	CO1	K1
9	Compare clock cycle and clock period. Clock cycle :The time for one clock period, usually of the processor clock, which runs at a constant rate. Clock period :The length of each clock cycle.	CO1	K2

10	How performance is measured in computing. N denotes number of machine Instructions, Suppose that the average number of basic steps needed to execute one machine instruction is S, where each basic step is completed in one clock cycle. If the clock cycle rate is R cycles per second, the processor time is given by T = (N x S) / R This is often referred to as the basic performance equation.	CO1	K1
11	Write the formula for CPU execution time for a program. CPU execution time = CPU clock cycles for a program / Clock rate	CO1	K1
12	Define CPI (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.	CO1	K2
13	What is a computer instruction?	CO1	K1
14	State the various types of operations required for instruction? • Data transfers between the main memory and the CPU registers • Arithmetic and logic operation on data • Program sequencing and control • I/O transfer	CO1	K1
15	Name various registers available in MIPS. MIPS has two primary types of registers, integer registers and floating point registers. In addition, MIPS has a small number of special purpose control registers	CO1	K1
16	Define instruction format. A standard machine which can be directly decoded and executed by the CPU is called instruction format.	CO1	K1
17	Write MIPS code for the given C code. if (i == j) f = g + h; else f = g – h 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:	CO1	K1
18	Define addressing modes and list its types. The different way a processor can access a data is called addressing mode.	CO1	K1

	Types of addressing modes are : 1. Register addressing mode. 2. Absolute or direct addressing mode. 3. Immediate addressing mode. 4. Indirect addressing mode. 5. Register indirect addressing mode. 6. Displacement addressing mode. 7. Relative addressing mode. 8. Base register addressing. 9. Index addressing mode. 10. Auto-increment addressing mode. 11. Auto-decrement addressing mode. 12. Stack addressing mode.		
19	Compare register and immediate addressing mode. **Register addressing mode** : The operand is the contents of processor register. The name of register is specified in the instruction. **Example** : MOV R2, R1 : This instruction copies the contents of register R2 to register R1. **Immediate addressing mode** : The operand is given explicitly in the instruction. **Example** : MOV #20, A : This instruction copies operand 20 in the register A. The sign # in front of the value of an operand is used to indicate that this value is an immediate operand.	CO1	K2
20	State the use of offset in base or displacement addressing mode. The offset is added to or subtracted from the base register to form the memory address.	CO1	K1
	Part – B		
1	Explain the various components of computer system with neat diagram. • **Input unit** - A computer accepts a digitally coded information through input unit using input devices. This unit contains basic components for accepting data and instruction in some form	CO1	K2

	and converting them into usable form by the system. Most commonly used input devices are keyboard and mouse.  Memory unit Arithmetic and Logic Unit Control unit Input unit Output unit • **Memory unit** - Memory unit is used to store programs and data. - Usually primary storage memory and secondary storage memory devices form a total memory unit. - Primary memory (main memory) is fast semiconductor RAM. - Size of this main memory is kept smaller as it is very expensive. - Secondary storage memories such as magnetic tapes, magnetic disk are used for the storage of larger amount of data. • **Arithmetic and Logic unit** - ALU is responsible for performing arithmetic and logical operations. - To perform these operations operands from main memory are brought into internal registers of processor. - After performing operation the result is either stored in the register or memory location. • **Control unit** - A control unit co-ordinates and controls all the activities amongst the functional units. - A basic function of control unit is to fetch the instructions stored in main memory, identify the operations and devices involved in it and accordingly generate control signals to execute the desired operations. - It uses control and timing signals to determine when a given action is to take place. • Output unit - A system for reporting a result is needed and this forms a output unit. - The output unit sends the processed results to the user using output devices such as video monitor, printer, plotter etc.		
2	Explain in detail about the basic operational concepts of a computer. • The basic function of computer is to execute program, sequence of instructions. • Instructions are stored in the computer memory. • Instructions are executed to process data which is loaded into the computer memory through input unit. • After processing the data, the result is either stored back into the computer memory for further reference or it is sent to the outside world through the output port. • All functional units of the computer contribute to execute a program.	CO1	K2



- The special function registers include Program Counter (PC), Instruction Register (IR), Memory Address Register (MAR) and Memory Data Register (MDR).

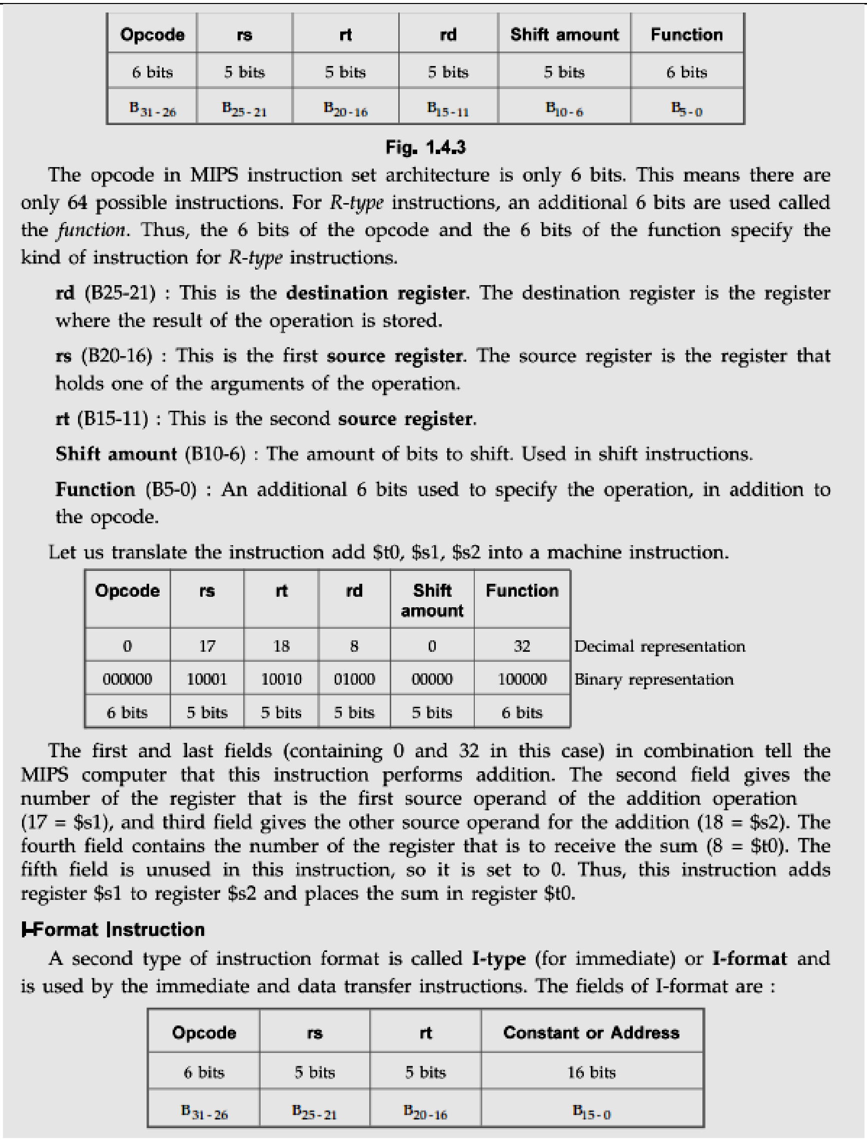
Program Counter (PC) :

- A program is a series of instructions stored in the memory. These instructions tell the CPU exactly how to get the desired result.
- It is important that these instructions must be executed in a proper order to get the correct result.
- The sequence of instruction execution is monitored by the program counter.
- It keeps track of which instruction is being executed and what the next instruction will be.

Instruction Register (IR) :

- It is used to hold the instruction that is currently being executed.
- The contents of IR are available to the control unit, which generate the timing signals that control the various processing elements involved in executing the instruction.

	Memory Address Register (MAR) and Memory Data Register (MDR) : • These registers are used to handle the data transfer between the main memory and the processor. • The MAR holds the address of the main memory to or from which data is to be transferred. • The MDR sometimes also called MBR (Memory Buffer Register) contains the data to be written into or read from the addressed word of the main memory. **General Purpose Registers :** • These are used to **hold the operands** for arithmetic and logic operations and/or used to store the result of the operation. • Since the access time of these registers is lowest, these are used to **store frequently used data**. **Execution of Program :** • PC is set to point to the first instruction of the program. • The contents of PC are transferred to the MAR, whose output is connected to the address lines of memory, and simultaneously Read Control Signal is sent to the memory. • After the access time of memory, the addressed word is received from memory and stored into the MDR. • The received instruction code, i.e. the contents of MDR are transferred to IR and now instruction is ready to be decoded and executed. • If the instruction involved an operation to be performed by the ALU and operand is available in general-purpose registers the operation is performed by the ALU. • If operand is not available in the general-purpose registers, it is reside in the memory. It has to be fetched by sending its address to the MAR and initiating a Read cycle. • When the operand is read from the memory into the MDR, it is transferred from the MDR to the ALU. • ALU then performs the desired operation and stores the result either in the general purpose registers or in the memory. • If the result of operation is to be stored in the memory, then the result is sent to the MDR. The address of the memory location where the result is to be stored is sent to the MAR, and a write cycle is initiated. • During the execution of current instruction, the contents of PC are incremented so that they points to the next instruction to be executed.		
3	Explain various instruction formats and illustrate the same with an example. MIP instruction is divided into segments called **fields**. Fig. 1.4.3 shows instruction format for register type (R- Type) instruction.	CO1	K2



Opcode	rs	rt	rd	Shift amount	Function
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
B_{31-26}	B_{25-21}	B_{20-16}	B_{15-11}	B_{10-6}	B_{5-0}

Fig. 1.4.3

The opcode in MIPS instruction set architecture is only 6 bits. This means there are only 64 possible instructions. For *R-type* instructions, an additional 6 bits are used called the *function*. Thus, the 6 bits of the opcode and the 6 bits of the function specify the kind of instruction for *R-type* instructions.

rd (B25-21) : This is the **destination register**. The destination register is the register where the result of the operation is stored.

rs (B20-16) : This is the first **source register**. The source register is the register that holds one of the arguments of the operation.

rt (B15-11) : This is the second **source register**.

Shift amount (B10-6) : The amount of bits to shift. Used in shift instructions.

Function (B5-0) : An additional 6 bits used to specify the operation, in addition to the opcode.

Let us translate the instruction add \$t0, \$s1, \$s2 into a machine instruction.

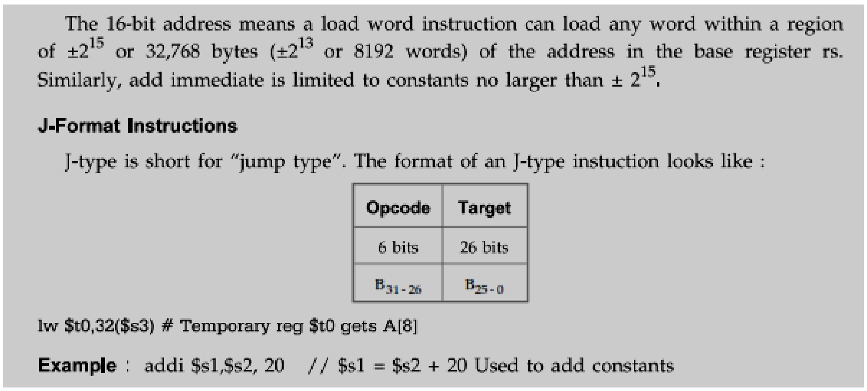
Opcode	rs	rt	rd	Shift amount	Function	
0	17	18	8	0	32	Decimal representation
000000	10001	10010	01000	00000	100000	Binary representation
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

The first and last fields (containing 0 and 32 in this case) in combination tell the MIPS computer that this instruction performs addition. The second field gives the number of the register that is the first source operand of the addition operation (17 = \$s1), and third field gives the other source operand for the addition (18 = \$s2). The fourth field contains the number of the register that is to receive the sum (8 = \$t0). The fifth field is unused in this instruction, so it is set to 0. Thus, this instruction adds register \$s1 to register \$s2 and places the sum in register \$t0.

I-Format Instruction

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 :

Opcode	rs	rt	Constant or Address
6 bits	5 bits	5 bits	16 bits
B_{31-26}	B_{25-21}	B_{20-16}	B_{15-0}



The 16-bit address means a load word instruction can load any word within a region of $\pm 2^{15}$ or 32,768 bytes ($\pm 2^{13}$ or 8192 words) of the address in the base register rs. Similarly, add immediate is limited to constants no larger than $\pm 2^{15}$.

J-Format Instructions

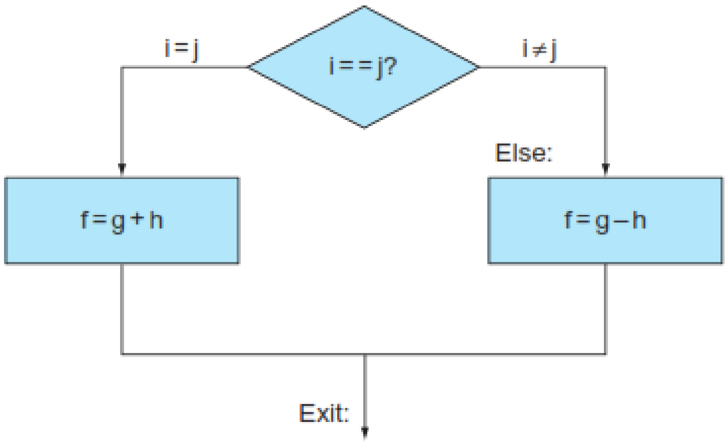
J-type is short for "jump type". The format of an J-type instuction looks like :

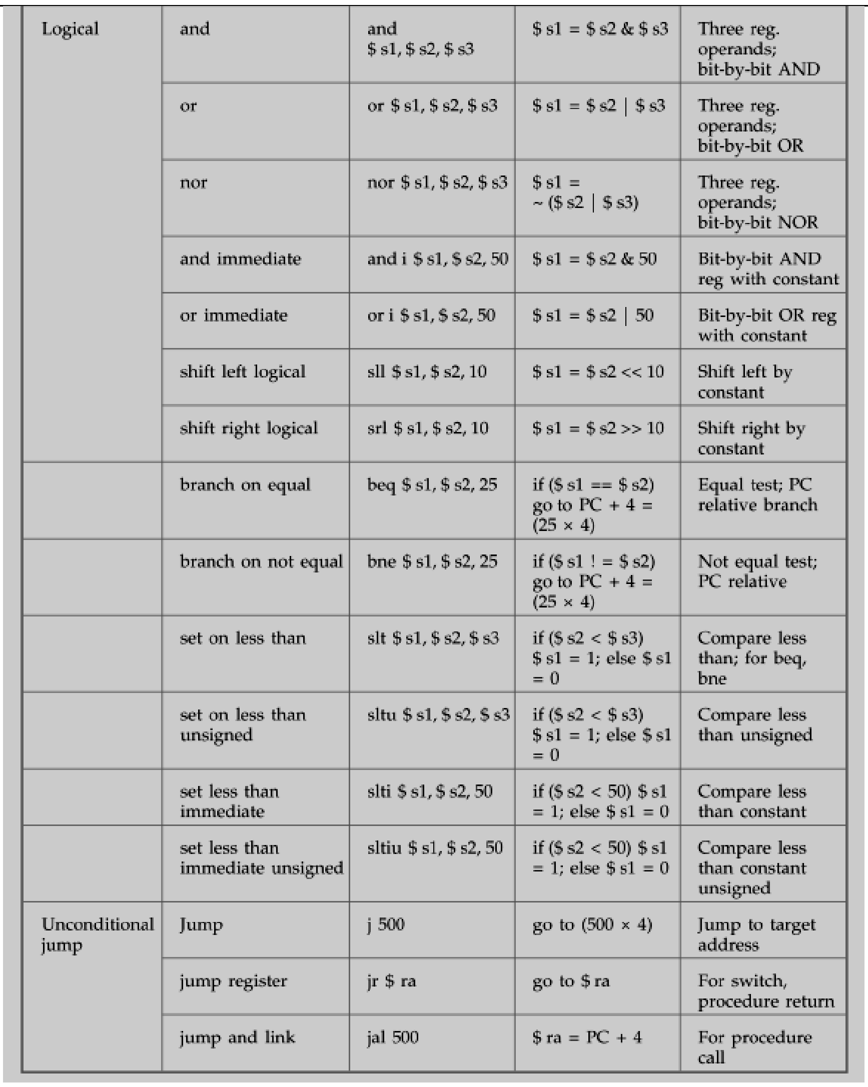
Opcode	Target
6 bits	26 bits
B_{31-26}	B_{25-0}

lw \$t0,32(\$s3) # Temporary reg \$t0 gets A[8]

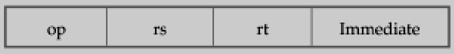
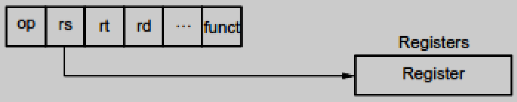
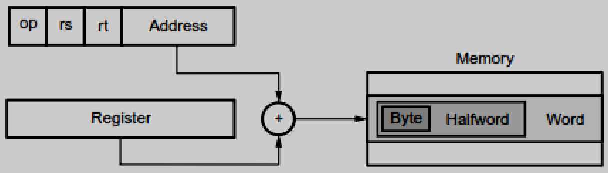
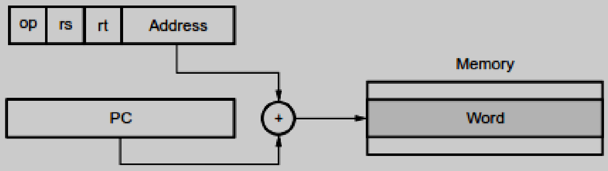
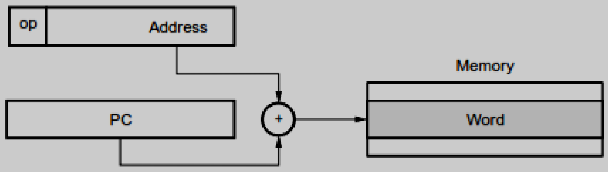
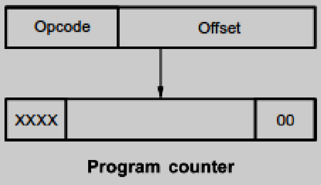
Example : addi \$s1,\$s2, 20 // \$s1 = \$s2 + 20 Used to add constants

4	Explain the various control operations available in MIPS and explain the instructions supporting the control operations? Control Operations Decision making and branching makes the computers more powerful. Decision Making: Decision making in MIPS assembly language includes two decision-making instructions	CO1	K2

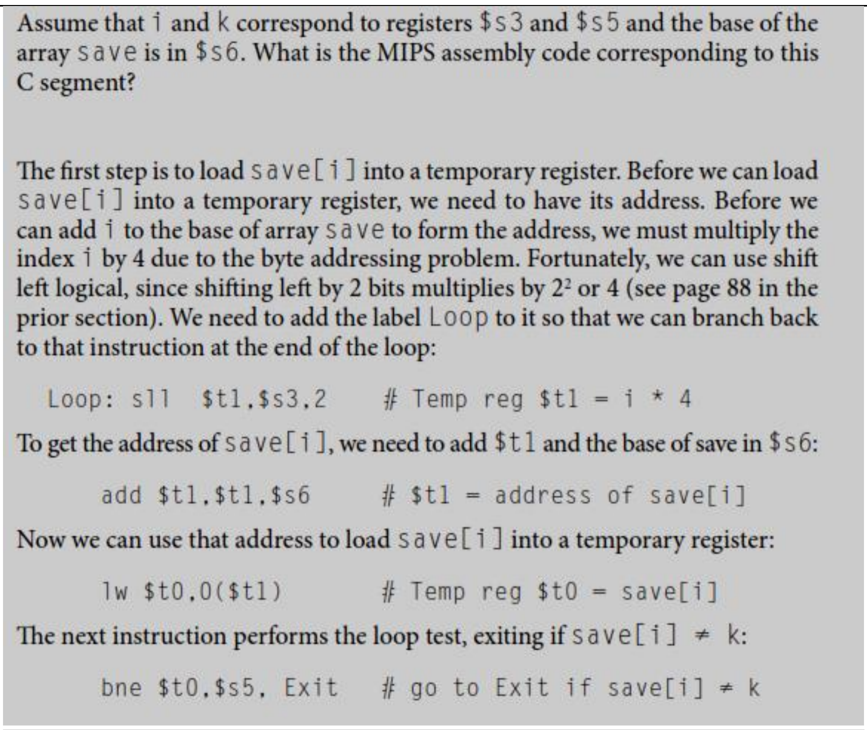
	(conditional branches): i) Branch if Equal (BEQ): beq register1, register2, L1 In this instruction, the go to the statement labeled L1 if the value in register1 is equal to the value in register2. i) Branch if not Equal (BNE): bne register1, register2, L1 In this instruction, the go to the statement labeled L1 if the value in register1 does not equal the value in register2. **Example** if (i == j) f = g + h; else f = g – h;  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:		
5	What are the various logical operations in MIPS and explain the instructions supporting the logical operations?	CO1	K2



Logical	and	and $ s1, $ s2, $ s3	$ s1 = $ s2 & $ s3	Three reg. operands; bit-by-bit AND
	or	or $ s1, $ s2, $ s3	$ s1 = $ s2 | $ s3	Three reg. operands; bit-by-bit OR
	nor	nor $ s1, $ s2, $ s3	$ s1 = ~ ($ s2 | $ s3)	Three reg. operands; bit-by-bit NOR
	and immediate	and i $ s1, $ s2, 50	$ s1 = $ s2 & 50	Bit-by-bit AND reg with constant
	or immediate	or i $ s1, $ s2, 50	$ s1 = $ s2 | 50	Bit-by-bit OR reg with constant
	shift left logical	sll $ s1, $ s2, 10	$ s1 = $ s2 << 10	Shift left by constant
	shift right logical	srl $ s1, $ s2, 10	$ s1 = $ s2 >> 10	Shift right by constant
	branch on equal	beq $ s1, $ s2, 25	if ($ s1 == $ s2) go to PC + 4 = (25 × 4)	Equal test; PC relative branch
	branch on not equal	bne $ s1, $ s2, 25	if ($ s1 ! = $ s2) go to PC + 4 = (25 × 4)	Not equal test; PC relative
	set on less than	slt $ s1, $ s2, $ s3	if ($ s2 < $ s3) $ s1 = 1; else $ s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $ s1, $ s2, $ s3	if ($ s2 < $ s3) $ s1 = 1; else $ s1 = 0	Compare less than unsigned
	set less than immediate	slti $ s1, $ s2, 50	if ($ s2 < 50) $ s1 = 1; else $ s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $ s1, $ s2, 50	if ($ s2 < 50) $ s1 = 1; else $ s1 = 0	Compare less than constant unsigned
Unconditional jump	Jump	j 500	go to (500 × 4)	Jump to target address
	jump register	jr $ ra	go to $ ra	For switch, procedure return
	jump and link	jal 500	$ ra = PC + 4	For procedure call

6	Explain in detail about addressing modes with an example. **MIPS Addressing Modes** The MIPS addressing modes are as follows : 1. **Immediate addressing :** In this addressing mode, the operand is a constant within the instruction itself.  **Example** : lui \$s0, 61 // Loads decimal 61 in upper 16 bits register \$s0 2. **Register addressing :** In this addressing mode, the operand is a register  op rs rt rd ... funct Registers Register **Example** : add \$t1,\$s0, \$s1 // Adds contents of \$s0 and \$s1 and store result in St1. 3. **Base or displacement addressing :** In this addressing mode, the operand is at the memory location whose address is the sum of a register and a constant in the instruction.  op rs rt Address Memory Register + Byte Halfword Word **Example** : lw \$t1 , 4 (\$t2) // where \$t1 = rs, \$t2 = base (memory address), 4 = offset value Thus ; \$t1 = Memory [\$t2 +4] 4. **PC-relative addressing :** In this addressing mode, the branch address is the sum of the PC and a constant in the instruction.  **Example** : beq \$0,\$3,Label 5. **Pseudodirect addressing :** In this addressing mode, the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC. Address in Pseudo-Direct must be a multiple of four.   **Example** : j label // go to location label.	CO1	K2
7	Consider three different processors P1, P2 and P3 executing the same instruction set. P1 has 3 GHz clock rate and a CPI of 1.5. P2 has a 2.5 GHz clock rate and a CPI of 1.0. P3 has a 4.0 GHz clock rate and has a CPI of 2.2. a) Which processor has the highest performance expresses in	CO1	K3

	instructions per second? (7) b) If the processors each execute a program in 10 seconds. Find the number of cycles and the number of instructions in each processor. (8) Performance of P1 (instructions / sec) = $\frac{3\times10^9}{1.5} = 2\times10^9$ Performance of P2 (instructions / sec) = $\frac{2.5\times10^9}{1.0} = 2.5\times10^9$ Performance of P3 (instructions / sec) = $\frac{4.0\times10^9}{2.2} = 1.82\times10^9$ **b)** Number of cycles = Time × Clock rate Cycles (P1) = $10\times3\times10^9 = 30\times10^9$ Cycles (P2) = $10\times2.5\times10^9 = 25\times10^9$ Cycles (P3) = $10\times4.0\times10^9 = 40\times10^9$ Time = $\frac{(\text{Number of instructions}\times\text{CPI})}{\text{Clock rate}}$ $\therefore$Number of instructions = $\frac{\text{Time}\times\text{Clock rate}}{\text{CPI}} = \frac{\text{Number of cycles}}{\text{CPI}}$ No. of Instructions (P1) = $\frac{30\times10^9}{1.5} = 20\times10^9$ No. of Instructions (P2) = $\frac{25\times10^9}{1.0} = 25\times10^9$ No. of Instructions (P3) = $\frac{40\times10^9}{2.2} = 18.18\times10^9$		
8	Translate the following C code to MIPS assemble code. Use a minimum number of instructions. Assume that i and k correspond to registers \$s3 and \$s5 and the base of the array save in \$s6. While (save[i] == k) i+=1;	CO1	K3



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?

The first step is to load save[i] into a temporary register. Before we can load save[i] into a temporary register, we need to have its address. Before we can add i to the base of array save to form the address, we must multiply the index i by 4 due to the byte addressing problem. Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by 2^2 or 4 (see page 88 in the prior section). We need to add the label Loop to it so that we can branch back to that instruction at the end of the loop:

```
Loop: sll  $t1,$s3,2    # Temp reg $t1 = i * 4
```

To get the address of save[i], we need to add $t1 and the base of save in $s6:

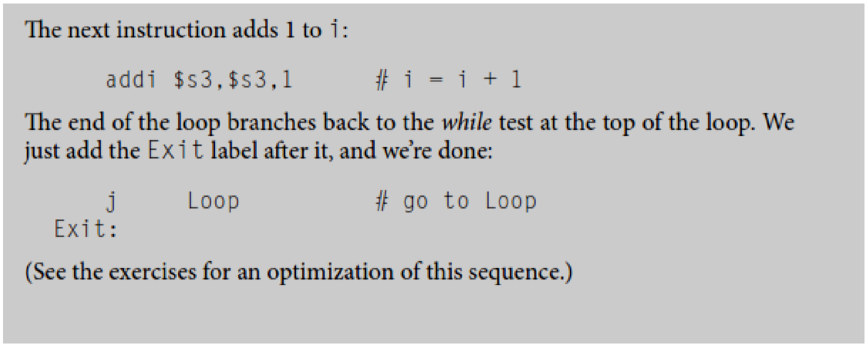
```
     add $t1,$t1,$s6    # $t1 = address of save[i]
```

Now we can use that address to load save[i] into a temporary register:

```
     lw $t0,0($t1)      # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if save[i] ≠ k:

```
     bne $t0,$s5, Exit  # go to Exit if save[i] ≠ k
```



The next instruction adds 1 to i:

```
     addi $s3,$s3,1     # i = i + 1
```

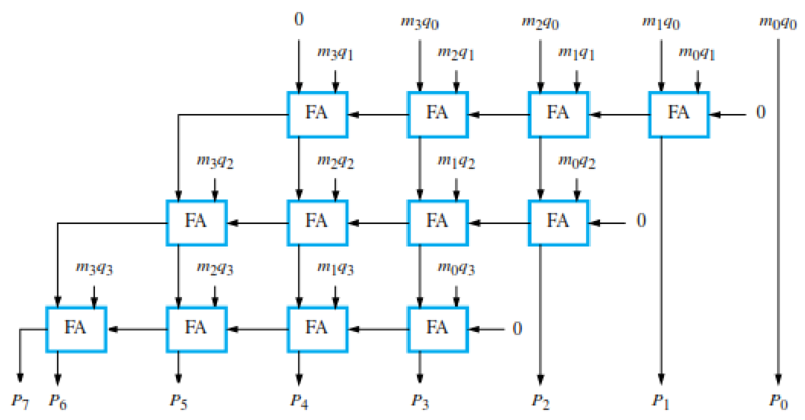
The end of the loop branches back to the *while* test at the top of the loop. We just add the Exit label after it, and we're done:

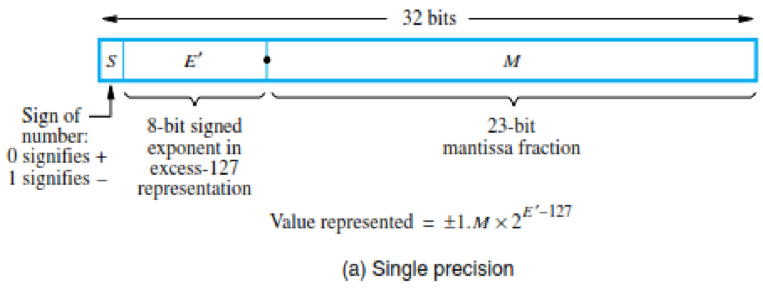
```
     j    Loop          # go to Loop
Exit:
```

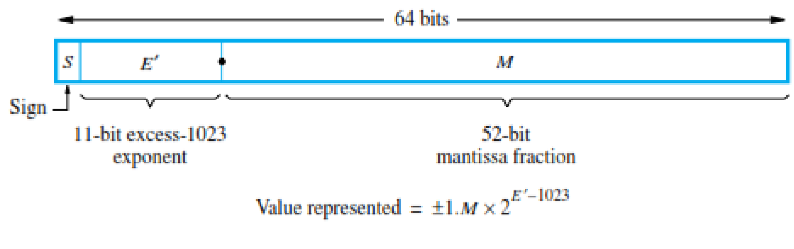
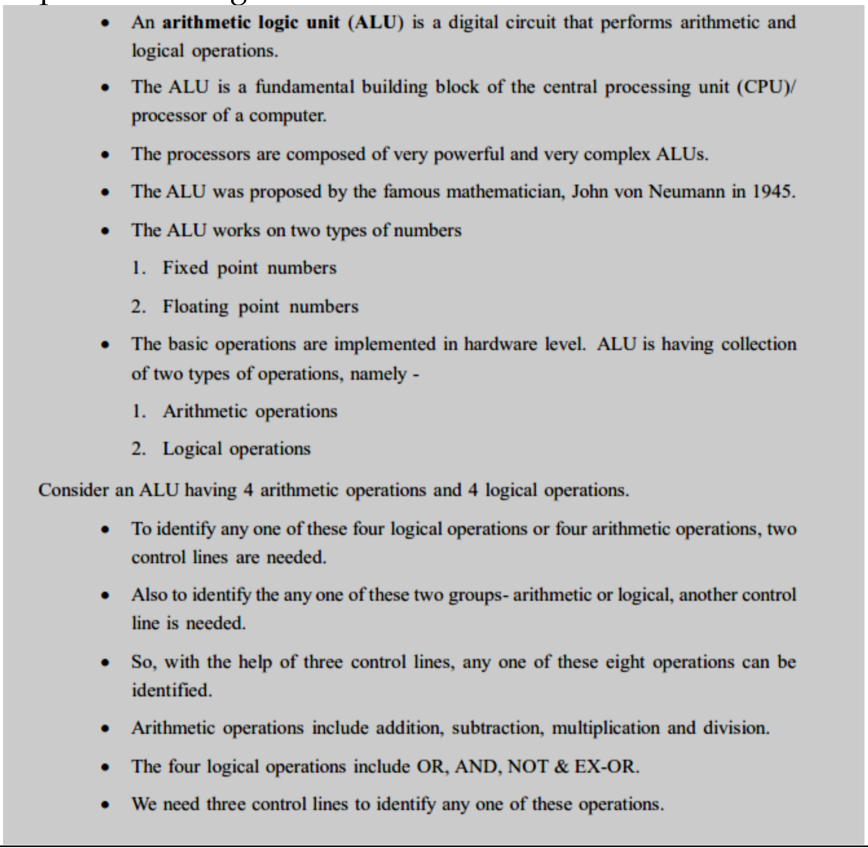
(See the exercises for an optimization of this sequence.)

KCG
COLLEGE OF TECHNOLOGY
AFFILIATED TO ANNA UNIVERSITY | AUTONOMOUS

UNIT II

QNo	Questions	COs	Bloom's Level
	PART - A		
1	Define half adder. A half adder is a circuit that takes two inputs, A and B, and produces two outputs, S (the sum of A and B) and C out (the carry out).	CO2	K1
2	Define full adder. The full adder is used to add three 1-bit binary numbers A, B, and carry C. The full adder has three input states and two output states i.e., sum and carry.	CO2	K1
3	Draw 4-bit full ripple carry adder.  0 m_3q_0 m_2q_0 m_1q_0 m_0q_0 m_3q_1 m_2q_1 m_1q_1 m_0q_1 FA 0 m_3q_2 m_2q_2 m_1q_2 m_0q_2 m_3q_3 m_2q_3 m_1q_3 m_0q_3 P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0 (a) Ripple-carry array	CO2	K2
4	How subtraction happens using adder? A Binary Subtractor is used to perform the subtraction of two binary numbers. Just as in the case of addition, this also can be done using an array of logic gates. The subtraction is done by using two's complement, in which the circuit first negates the binary number to obtain its complement and then adds one to the result before imputing it into a binary adder circuit together with the original number.	CO2	K1
5	Differentiate carry and overflow. The difference is that carry out applies when you have somewhere else to put it, while overflow is when you do not.	CO2	K2
6	What is Booth's algorithm? The booth algorithm is a multiplication algorithm that allows us to multiply the two signed binary integers in 2's complement, respectively.	CO2	K1
7	Define division. Division is repeated subtraction.	CO2	K1
8	What is divide overflow? A program error in which a number divided by zero or by a number that creates a result too large for the computer to handle.	CO2	K1
9	Add 6_{10} to 7_{10} in binary.	CO2	K1

	110 + 111 ------ 1101		
10	Subtract 6_{10} from 7_{10} in binary. 111 - 110 ------ 001	CO2	K1
11	Multiply 100010_2 **x** 100110_2 100010 (34) × 100110 (38) ------------ 000000 (100010 × 0) + 100010 (100010 × 1, shifted 1 bit left) + 0000000 (100010 × 0, shifted 2 bits left) + 10001000 (100010 × 1, shifted 3 bits left) +000000000 (100010 × 0, shifted 4 bits left) +1000100000 (100010 × 1, shifted 5 bits left) ------------ 110110100 (Decimal: 1292)	CO2	K1
12	Divide $(1001010)_2 \div (1000)_2$.	CO2	K1
13	Explain about the floating point number. A floating point number is a type of number used in computer programming that is used to represent real numbers with a high degree of precision.	CO2	K2
14	Give the representation of single precision floating point number.  32 bits S E' M Sign of number: 0 signifies + 1 signifies − 8-bit signed exponent in excess-127 representation 23-bit mantissa fraction Value represented = ±1.M × 2^(E'−127) (a) Single precision	CO2	K1
15	Give the representation of double precision floating point number.	CO2	K1

	 64 bits S E′ M Sign 11-bit excess-1023 exponent 52-bit mantissa fraction Value represented = ±1.M × 2^(E′−1023) (c) Double precision		
16	Write 82.125$_{10}$ in single and double precision format.	CO2	K1
17	What is floating point normalization? Normalisation, in floating-point number systems, is a process of standardising these numbers into a consistent format.	CO2	K1
18	State the purpose of guard bits used in floating point arithmetic. The extra bits that are used in intermediate calculations to improve the precision of the result are called guard bits.	CO2	K1
19	What are the ways to truncate the guard bits? Chopping: simply drop all guard bits.	CO2	K1
20	Define overflow and underflow with examples Overflow happens when the result is too large, while underflow occurs when the result is too small.	CO2	K1
Part – B			
1	Explain the design of ALU in detail.  • An **arithmetic logic unit** (ALU) is a digital circuit that performs arithmetic and logical operations. • The ALU is a fundamental building block of the central processing unit (CPU)/ processor of a computer. • The processors are composed of very powerful and very complex ALUs. • The ALU was proposed by the famous mathematician, John von Neumann in 1945. • The ALU works on two types of numbers 1. Fixed point numbers 2. Floating point numbers • The basic operations are implemented in hardware level. ALU is having collection of two types of operations, namely - 1. Arithmetic operations 2. Logical operations Consider an ALU having 4 arithmetic operations and 4 logical operations. • To identify any one of these four logical operations or four arithmetic operations, two control lines are needed. • Also to identify the any one of these two groups- arithmetic or logical, another control line is needed. • So, with the help of three control lines, any one of these eight operations can be identified. • Arithmetic operations include addition, subtraction, multiplication and division. • The four logical operations include OR, AND, NOT & EX-OR. • We need three control lines to identify any one of these operations.	CO2	K2

- The input combination of these control lines are shown below.
- Control line C_2 is used to identify the group: logical or arithmetic,
  - C_2=0: arithmetic operation
  - C_2=0: logical operation.
- Control lines C_0 and C_1 are used to identify any one of the four operations in a group. One possible combination is given here.

C_1	C_0	Arithmetic $C_2 = 0$	Logical $C_2 = 1$
0	0	Addition	OR
0	1	Subtraction	AND
1	0	Multiplication	NOT
1	1	Division	**EX-OR**

- A 3 x 8 decoder is used to decode the instruction.
- The block diagram of the ALU is shown.

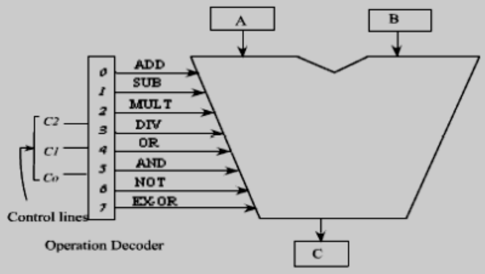


Figure 2.1: Block Diagram of the ALU

- The ALU has got two input registers named as A and B and one output storage register, named as C.
- It performs the operation as: C = A operator B
- The input data are stored in A and B, and according to the operation specified in the control lines, the ALU perform the operation and put the result in register C.

2

Explain with an example how to multiply two unsigned binary numbers.

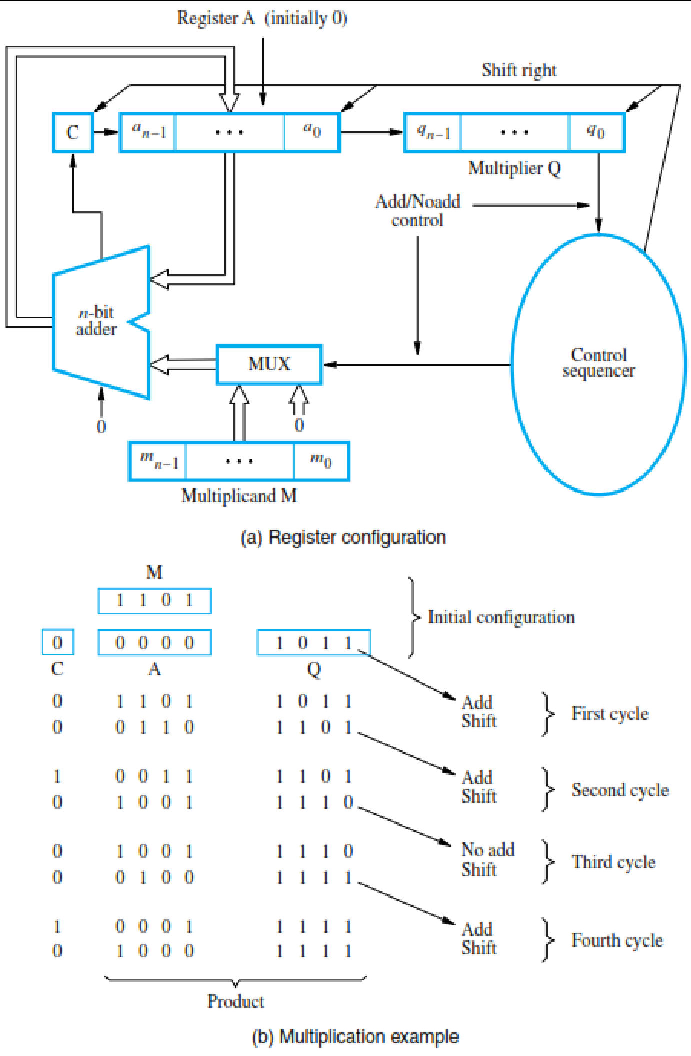
CO2

K2

9.3.2 Sequential Circuit Multiplier

The combinational array multiplier just described uses a large number of logic gates for multiplying numbers of practical size, such as 32- or 64-bit numbers. Multiplication of two n-bit numbers can also be performed in a sequential circuit that uses a single n-bit adder.

The block diagram in Figure 9.7*a* shows the hardware arrangement for sequential multiplication. This circuit performs multiplication by using a single n-bit adder n times to implement the spatial addition performed by the n rows of ripple-carry adders in Figure 9.6*b*. Registers A and Q are shift registers, concatenated as shown. Together, they hold partial product PP*i* while multiplier bit q_i generates the signal Add/Noadd. This signal causes the multiplexer MUX to select 0 when $q_i = 0$, or to select the multiplicand M when $q_i = 1$, to be added to PP*i* to generate PP$(i + 1)$. The product is computed in n cycles. The partial product grows in length by one bit per cycle from the initial vector, PP0, of n 0s in register A. The carry-out from the adder is stored in flip-flop C, shown at the left end of register A. At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0. At the end of each cycle, C, A, and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit q_i appears at the LSB position of Q to generate the Add/Noadd signal at the correct time, starting with q_0 during the first cycle, q_1 during the second cycle, and so on. After they are used, the multiplier bits are discarded by the right-shift operation. Note that the carry-out from the adder is the leftmost bit of PP$(i + 1)$, and it must be held in the C flip-flop to be shifted right with the contents of A and Q. After n cycles, the high-order half of the product is held in register A and the low-order half is in register Q. The multiplication example of Figure 9.6*a* is shown in Figure 9.7*b* as it would be performed by this hardware arrangement.

	 Register A (initially 0) Shift right C a_{n-1} ... a_0 q_{n-1} ... q_0 Multiplier Q Add/Noadd control n-bit adder MUX 0 0 Control sequencer m_{n-1} ... m_0 Multiplicand M M 1 1 0 1 0 0 0 0 0 1 0 1 1 C A Q Initial configuration 0 1 1 0 1 1 0 1 1 Add 0 0 1 1 0 1 1 0 1 Shift First cycle 1 0 0 1 1 1 1 0 1 Add 0 1 0 0 1 1 1 1 0 Shift Second cycle 0 1 0 0 1 1 1 1 0 No add 0 0 1 0 0 1 1 1 1 Shift Third cycle 1 0 0 0 1 1 1 1 1 Add 0 1 0 0 0 1 1 1 1 Shift Fourth cycle Product (a) Register configuration (b) Multiplication example **Figure 9.7** Sequential circuit binary multiplier.		
3	Explain with an example how to multiply two signed binary numbers. The Booth algorithm [1] generates a $2n$-bit product and treats both positive and negative 2's-complement n-bit operands uniformly. To understand the basis of this algorithm, consider a multiplication operation in which the multiplier is positive and has a single block of 1s, for example, 0011110. To derive the product, we could add four appropriately shifted versions of the multiplicand, as in the standard procedure. However, we can reduce the number of required operations by regarding this multiplier as the difference between two numbers: $$\begin{array}{rll} & 0100000 & (32) \\ - & 0000010 & (2) \\ \hline & 0011110 & (30) \end{array}$$ This suggests that the product can be generated by adding 2^5 times the multiplicand to the 2's-complement of 2^1 times the multiplicand. For convenience, we can describe the sequence of required operations by recoding the preceding multiplier as $0\ {+1}\ 0\ 0\ 0\ {-1}\ 0$. In general, in the Booth algorithm, -1 times the shifted multiplicand is selected when moving from 0 to 1, and $+1$ times the shifted multiplicand is selected when moving from	CO2	K2

1 to 0, as the multiplier is scanned from right to left. Figure 9.9 illustrates the normal and the Booth algorithms for the example just discussed. The Booth algorithm clearly extends to any number of blocks of 1s in a multiplier, including the situation in which a single 1 is considered a block. Figure 9.10 shows another example of recoding a multiplier. The case when the least significant bit of the multiplier is 1 is handled by assuming that an implied 0 lies to its right. The Booth algorithm can also be used directly for negative multipliers, as shown in Figure 9.11.

To demonstrate the correctness of the Booth algorithm for negative multipliers, we use the following property of negative-number representations in the 2's-complement system.

```
                     0  1  0  1  1  0  1
                     0  0 +1 +1 +1 +1  0
                    ---------------------
                     0  0  0  0  0  0  0
                  0  1  0  1  1  0  1
               0  1  0  1  1  0  1
            0  1  0  1  1  0  1
         0  1  0  1  1  0  1
      0  0  0  0  0  0  0
   0  0  0  0  0  0  0
------------------------------------------
0  0  0  1  0  1  0  1  0  0  0  1  1  0

                     0  1  0  1  1  0  1
                     0 +1  0  0  0 -1  0
                    ---------------------
0  0  0  0  0  0  0  0  0  0  0  0  0  0
1  1  1  1  1  1  1  0  1  0  0  1  1     <---  2's complement of
0  0  0  0  0  0  0  0  0  0  0  0              the multiplicand
0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0
0  0  0  1  0  1  1  0  1
0  0  0  0  0  0  0  0
------------------------------------------
0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

Figure 9.9 Normal and Booth multiplication schemes.

4	Explain the algorithm for restoring integer division with suitable example.	CO2	K2

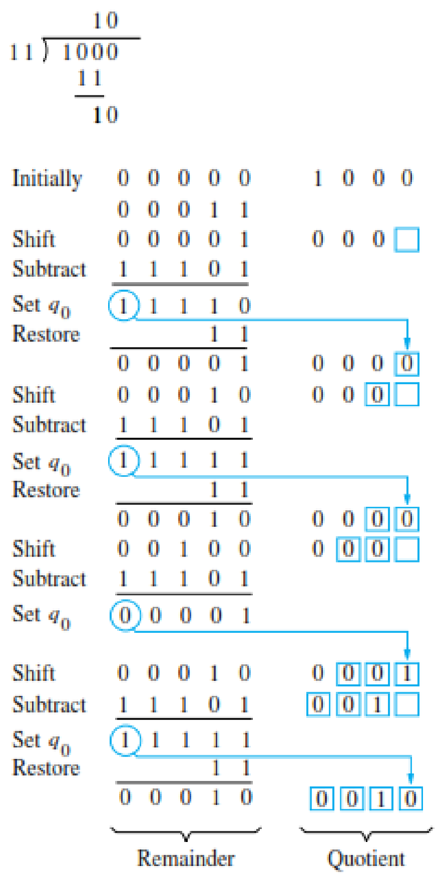
Restoring Division

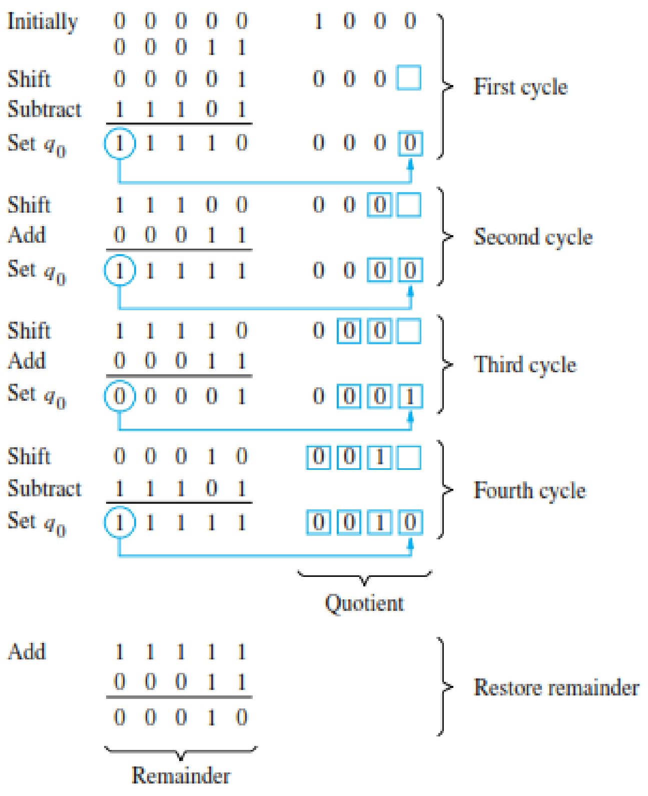
Figure 9.23 shows a logic circuit arrangement that implements the restoring division algorithm just discussed. Note its similarity to the structure for multiplication shown in Figure 9.7. An n-bit positive divisor is loaded into register M and an n-bit positive dividend

is loaded into register Q at the start of the operation. Register A is set to 0. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A. The required subtractions are facilitated by using 2's-complement arithmetic. The extra bit position at the left end of both A and M accommodates the sign bit during subtractions. The following algorithm performs restoring division.

Do the following three steps n times:

1. Shift A and Q left one bit position.
2. Subtract M from A, and place the answer back in A.
3. If the sign of A is 1, set q_0 to 0 and add M back to A (that is, restore A); otherwise, set q_0 to 1.

	 Initially Shift Subtract Set q_0 Restore First cycle Second cycle Third cycle Fourth cycle Remainder Quotient **Figure 9.24** A restoring division example.		
5	Explain the algorithm for non-restoring integer division with suitable example. **Stage 1:** Do the following two steps n times: 1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A. 2. Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0. **Stage 2:** If the sign of A is 1, add M to A. Stage 2 is needed to leave the proper positive remainder in A after the n cycles of Stage 1. The logic circuitry in Figure 9.23 can also be used to perform this algorithm, except that	CO2	K2

	 Initially 0 0 0 0 0 1 0 0 0 0 0 0 1 1 Shift 0 0 0 0 1 0 0 0 Subtract 1 1 1 0 1 Set q_0 1 1 1 1 0 0 0 0 0 First cycle Shift 1 1 1 0 0 0 0 0 Add 0 0 0 1 1 Set q_0 1 1 1 1 1 0 0 0 0 Second cycle Shift 1 1 1 1 0 0 0 0 Add 0 0 0 1 1 Set q_0 0 0 0 0 1 0 0 0 1 Third cycle Shift 0 0 0 1 0 0 0 1 Subtract 1 1 1 0 1 Set q_0 1 1 1 1 1 0 0 1 0 Fourth cycle Quotient Add 1 1 1 1 1 0 0 0 1 1 0 0 0 1 0 Restore remainder Remainder **Figure 9.25** A non-restoring division example.		

6	Explain with an algorithm how to do addition and subtraction of two floating-point binary numbers.	CO2	K2

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary.

Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed.

The hardware implementation of floating-point operations involves a considerable amount of logic circuitry. These operations can also be implemented by software routines. In either case, the computer must be able to convert input and output from and to the user's decimal representation of numbers. In many general-purpose processors, floating-point operations are available at the machine-instruction level, implemented in hardware.

An example of the implementation of floating-point operations is shown in Figure 9.28. This is a block diagram of a hardware implementation for the addition and subtraction of 32-bit floating-point operands that have the format shown in Figure 9.26*a*. Following the Add/Subtract rule given in Section 9.7.1, we see that the first step is to compare exponents to determine how far to shift the mantissa of the number with the smaller exponent. The shift-count value, *n*, is determined by the 8-bit subtractor circuit in the upper left corner of the figure. The magnitude of the difference $E'_A - E'_B$, or *n*, is sent to the SHIFTER unit. If *n* is larger than the number of significant bits of the operands, then the answer is essentially the larger operand (except for guard and sticky-bit considerations in rounding), and shortcuts can be taken in deriving the result. We do not explore this in detail.

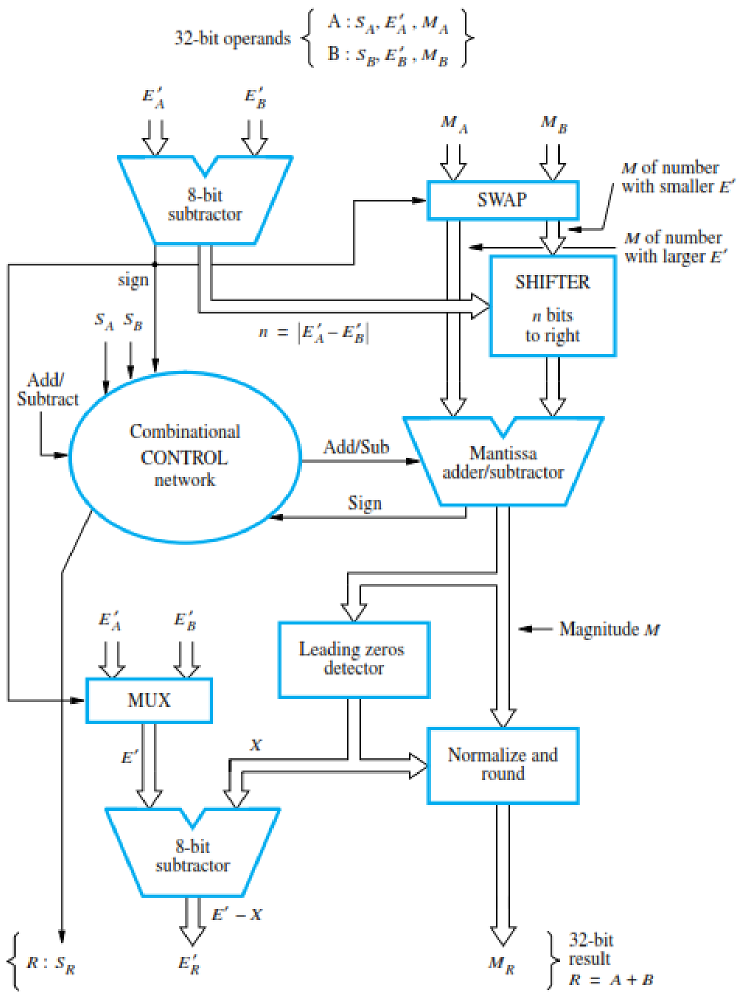
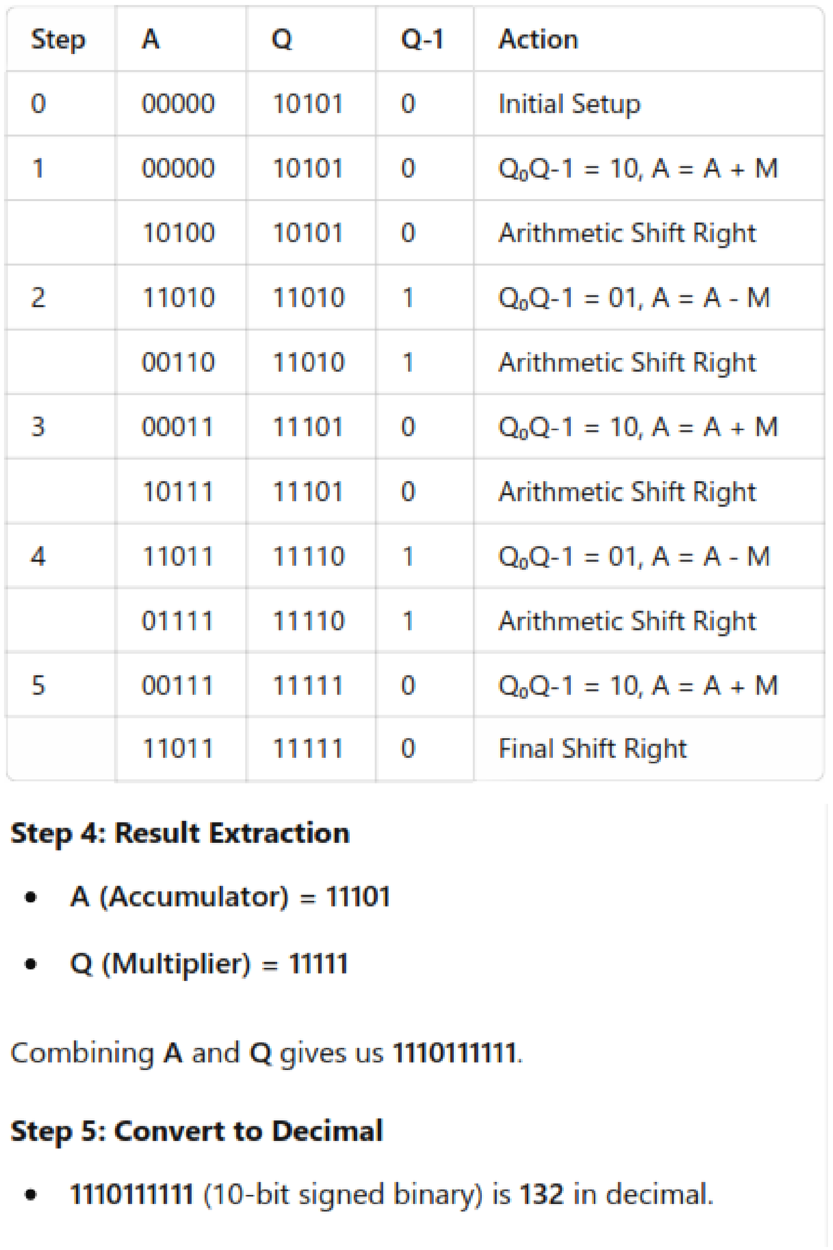


Figure 9.28 Floating-point addition-subtraction unit.

7	Describe in detail about Booth's multiplication algorithm and perform the booth's operation for the 5-bit signed operand, -12 is the multiplicand, and it's multiplied by –11 the multiplier.	CO2	K3



Step	A	Q	Q-1	Action
0	00000	10101	0	Initial Setup
1	00000	10101	0	Q_0Q-1 = 10, A = A + M
	10100	10101	0	Arithmetic Shift Right
2	11010	11010	1	Q_0Q-1 = 01, A = A - M
	00110	11010	1	Arithmetic Shift Right
3	00011	11101	0	Q_0Q-1 = 10, A = A + M
	10111	11101	0	Arithmetic Shift Right
4	11011	11110	1	Q_0Q-1 = 01, A = A - M
	01111	11110	1	Arithmetic Shift Right
5	00111	11111	0	Q_0Q-1 = 10, A = A + M
	11011	11111	0	Final Shift Right

Step 4: Result Extraction

- **A (Accumulator) = 11101**
- **Q (Multiplier) = 11111**

Combining **A** and **Q** gives us **1110111111**.

Step 5: Convert to Decimal

- **1110111111** (10-bit signed binary) is **132** in decimal.

8	Mr. John has been assigned a project by his team leader in ALS Technologies. His project is to design an algorithm for 2's complement division using addition and subtraction operations. Help Mr. John is in designing an algorithm by sketching the flowchart for non-restoring division and also check the working of it with the following numbers: 24÷4	CO2	K3

Stage 1: Do the following two steps *n* times:

1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
2. Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

Stage 2: If the sign of A is 1, add M to A.

Stage 2 is needed to leave the proper positive remainder in A after the *n* cycles of Stage 1. The logic circuitry in Figure 9.23 can also be used to perform this algorithm, except that

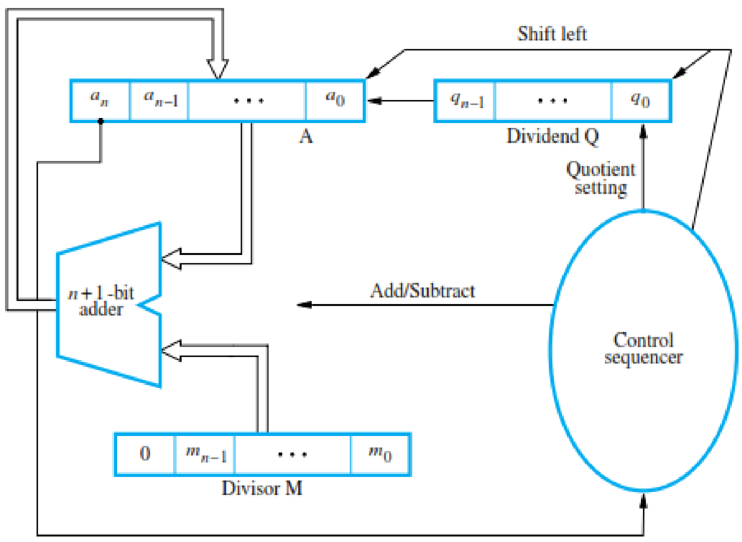


Figure 9.23 Circuit arrangement for binary division.

Iteration	Shift (R, Q)	R Update (Add/Subtract D)	Q_0
1	00000 11000	R = R - D → 11100	0
2	11001 10000	R = R + D → 11101	0
3	11011 00000	R = R + D → 11111	0
4	11110 00000	R = [illegible] D → 00010	1
5	00100 00000	R = R - D → 00000	1

Step 3: Final Correction

Since R = 00000 (positive), no need for correction.

Final Result:

- Quotient (Q) = 01100_2 → 6 in decimal
- Remainder (R) = 00000_2 → 0 in decimal