* Architecture: Orgnaization of the hardware components of a computer; Software interface to get the components to carry out desired operation
* machine language： Binary representation used for communication within a computer system.   assembly language: A symbolic representation of machine instructions. GB = 10^9 GHz

clock = 10^9
* .text: Subsequent items(instructions) stored in Text segment at next available address.
  .globl: Declare the listed label(s) as global to enable referencing from other file.
  .data: Subsequent items stored in Data segment at next available address.
  .asciiz: Store the string in the Data segment and add null terminator.
* MIPS instructions: ▪ op: Basic operation of the instruction, traditionally called the opcode. ▪ rs: The first register source operand.
▪ rt: The second register source operand. ▪ rd: The register destination operand. It gets the result of the operation.
▪ shamt: Shift amount. (Section 2.6 explains shift instructions and this term; it will not be used until then, and hence the field contains zero in this section.)
▪ funct: Function. This field, often called the function code, selects the specific variant of the operation in the op field.
* sys call: part of the application Binary Interface(simplified interface to I/o
  li: load immediate move(move constants of one register to another)/ la: load address
  sbrk: unix system called used to implement malloc(allocate space= new in java) and some times free—>change the program "break" between the stack and heap
* call function: the calling program, or caller, puts the parameter values in $a0–$a3 and uses jal X to jump to procedure X (sometimes named the callee). The callee then performs the calculations, places the results in $v0 and $v1, and returns control to the caller using jr $ra.
* malloc() allocates space on the heap and returns a pointer to it. free() releases space on the heap to which the pointer points.
* assembly language: A symbolic language that can be trans lated into binary machine language.
* pseudoinstruction: A common variation of assembly language instructions often treated as if it were an instruction in its own right.
* Given a branch on register $s0 being equal to register $s1, beq $s0, $s1, L1. Replace it by a pair of instructions that offers a much greater branching distance.
Answer: These instructions replace the short-address conditional branch:

```
      bne $s0, $s1, L2
      j  L1
L2:
```

* Moore's Law. It states that integrated circuit resources double every 18–24 months.
* Amdahl's lay: if you speedup part of a system, you are not going to see proportional improvements overall.
* Application software->System software->Hardware
systems software : Software that provides services that are commonly useful, including operating systems, compilers, loaders, and assemblers.

- operating system: Supervising program that manages the resources of a computer for the benefit of the programs that run on that computer.
- compiler: A program that translates high-level language statements into assembly language statements.
- assembler: A program that translates a symbolic version of instructions into the binary version.

Hardware:including I/O, memory,datapath, control.(data path and control combined and called the processor)

- The datapath performs the arithmetic operations, and control tells the datapath, memory, and I/O devices what to do according to the wishes of the instructions of the program.
- memory: The storage area in which programs are kept when they are running and that contains the data needed by the running programs.

* High-level-language(e.g. C)—>(Compiler)—>Assembly language—>(Assembler)—>Binary machine language.
* Instruction set architecture: Also called architecture. An abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on.
* Application binary interface (ABI): The user portion of the instruction set plus the operating system interfaces used by application programmers. It defines a standard for binary portability across computers.
* Interpretations of performance: execution time, throughput, power consumption.

- 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.
  Performance = 1/ Execution time or execution time per program.

1. Wall clock time, response time, or elapsed time: These terms mean the total time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead—everything.
2. CPU execution time or simply CPU time: The time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. The actual time the CPU spends computing for a specific task.
3. 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.
4. Clock period: The length of each clock cycle. Clock rate (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period. = cycles / second
5. CPI(clock cycles per instruction): Average number of clock cycles per instruction for a program or program fragment.= clock cycles/ instruction count
6. IPC(instructions per clock cycle): Average number of instructions per clock cycle for a program or program fragment.
7. CPU time = Instruction Count * CPI * Clock cycle time = Instruction Count * CPI / Clock rate
   CPU time = Clock Cycles * Clock Cycle time = Clock cycles / Clock rate

- Throughput: Also called bandwidth. Another measure of performance, it is the number of tasks completed per unit time.
- Power = Energy * Capacitive load * Voltage^2

Components affecting performance: 1. algorithm: Instruction count, (CPI)2. programming language: Instruction count, CPI 3. compiler: Instruction count, CPI 4.processor 5. I/O 6.Instruction set architecture: Instruction count, CPI, clock rate
* SPEC (System Performance Evaluation Cooperative) is an effort funded and supported by a number of computer vendors to create standard sets of benchmarks for modern computer systems.
  Dividing the execution time of a reference processor by the execution time of the measured computer normalizes the execution time measurements; this normalization yields a measure, called the SPECratio, which has the advantage that bigger numeric results indicate faster performance.
* MIPS (million instructions per second): A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and 10^6 = Instruction count / (Execution time*10^6) = Clock rate / (CPI*10^6)
* MFLOPS: millions of floating point ops per sec

Deci	bin	hex	oct
0	0000	0	00
1	0001	1	01
2	0010	2	02
3	0011	3	03
4	0100	4	04
5	0101	5	05
6	0110	6	06
7	0111	7	07
8	1000	8	10
9	1001	9	11
10	1010	a	12
11	1011	b	13
12	1100	c	14
13	1101	d	15
14	1110	e	16
15	1111	f	17

```
# Purpose:  Implement recursive binary search in MIPS32
# Input:    n:     number of elements in list; list:  sorted list of n ints; val:   int to search for
# Output:   Subscript of val if val is in list ; Otherwise -1
        .text
        .globl main
main:
        addi    $sp, $sp, -416          # Make additional stack space.  4 words for $ra, $s0, $s1, $s2.100 words for list
        sw      $ra, 412($sp)           # Put contents of $ra on stack
        sw      $s0, 408($sp)           # Put $s0 on stack
        sw      $s1, 404($sp)           # Put $s1 on stack
        sw      $s2, 400($sp)           # Put $s2 on stack
        move    $s0, $sp                # $s0 = stores start address of list = $sp
        # Ask the OS to read a number and put it in $s1 = n
        li      $v0, 5                  # Code for read int.
        syscall                         # Ask the system for service.
        move    $s1, $v0                # Put the input value (n) in a safe  place
        # Now read in the list
        move    $a0, $s0                # First arg is list
        move    $a1, $s1                # Second arg is n
        jal     rd_lst
        # Read the value to search for, val
        li      $v0, 5                  # Code for read int.
        syscall                         # Ask the system for service.
        move    $s2, $v0                # Put the input value (val) in a safe place
        # Now carry out the binary search
        move    $a0, $s2                # First arg is val
        move    $a1, $s0                # Second arg is list
        move    $a2, $zero              # Third arg is 0
```

```
        addi    $a3, $s1, -1            # Fourth arg is n-1
        jal     binsrch
        # Now print the result
        move    $a0, $v0                # Arg is return val from bin_srch
        li      $v0, 1                  # Code for print int
        syscall
        # Prepare for return
        lw      $ra, 412($sp)           # Retrieve return address
        lw      $s0, 408($sp)           # Retrieve $s0
        lw      $s1, 404($sp)           # Retrieve $s1
        lw      $s2, 400($sp)           # Retrieve $s2
                addi            $sp, $sp, 416           # Make additional stack space.
        li      $v0, 10                 # For MARS
        syscall
        ### Read_list function
        #    $a0 is the address of the beginning of list (In/out); $a1 is n (In)
rd_lst:
        # Setup
        addi    $sp, $sp, -4            # Make space for return address
        sw      $ra, 0($sp)             # Save return address
        # Main for loop
        move    $t0, $zero              # $t0 = i = 0
rd_tst: bge     $t0, $a1, rddone        # If  i = $t0 >= $a1 = n branch out of loop. Otherwise continue.
                li              $v0, 5                                  # Code for read int.
                syscall                                                 # Ask the system for service.
        sll     $t1, $t0, 2             # Words are 4 bytes:  use 4*i, not i
        add     $t1, $a0, $t1           # $t1 = list + i
                sw      $v0, 0($t1)             # Put the input value in $v0 in list[i]
        addi    $t0, $t0, 1             # i++
        j       rd_tst                  # Go to the loop test
        # Prepare for return
rddone: lw      $ra, 0($sp)             # retrieve return address
        addi    $sp, $sp 4              # adjust stack pointer
        jr      $ra                     # return
        ### Print_list function (Only for Debugging):$a0 is the address of the beginning of list (In), $a1 is n  (In).
pr_lst:
        # Setup
        addi    $sp, $sp, -4            # Make space for return address
        sw      $ra, 0($sp)             # Save return address
        # Main for loop
        move    $t2, $a0                # Need $a0 for syscall:  so copy to t2
        move    $t0, $zero              # $t0 = i = 0
pr_tst: bge     $t0, $a1, prdone        # If  i = $t0 >= $a1 = n  branch out of loop.Otherwise continue.
        sll     $t1, $t0, 2             # Words are 4 bytes:  use 4*i, not i
        add     $t1, $t2, $t1           # $t1 = list + i
                lw      $a0, 0($t1)             # Put the value to print in $a0
                li              $v0, 1                                  # Code for print int.
                syscall
        # Print a space
        la      $a0, space              #
        li      $v0, 4                  # Code for print string
        syscall
        addi    $t0, $t0, 1             # i++
        j       pr_tst                  # Go to the loop test
        # print a newline
prdone:
        la      $a0, newln
        li      $v0, 4                  # code for print string
        syscall
        # Prepare for return
        lw      $ra, 0($sp)             # retrieve return address
        addi    $sp, $sp 4              # adjust stack pointer
        jr      $ra                     # return
        ### Bin_srch function
        # $a0 is the value to be searched for (val),$a1 is the list, $a2 is the current lower bound (lo),$a3 is the current upper bound (hi)
        # All args are input args:  $a0 and $a1 are unchanged in the recursive calls.  $a2 and $a3 are changed, but after a call starts a recursive call, their current values aren't used again.
binsrch:                                # Bin_srch function
        # Setup
        addi    $sp, $sp, -4            # Make space for return address
        sw      $ra, 0($sp)             # Save return address
        bgt     $a2, $a3, not_in        # If lo = $a2 > $a3 = hi, val isn't in the list, quit
        add     $t0, $a2, $a3           # $t0 = hi + lo
        srl     $t0, $t0, 1             # $t0 = mid = $t0/2
        sll     $t1, $t0, 2             # $t1 = byte offset of list[mid]
        add     $t1, $t1, $a1           # $t1 = absolute addr of list[mid]
        lw      $t2, 0($t1)             # $t2 = list[mid]
        bne     $a0, $t2, not_eq        # If $a0 = val != list[mid] = $t2, go to not_eq
        # We found val
        move    $v0, $t0                # val = list[mid], set return value to mid.
        j       done
        # We didn't find val this time
not_eq: bgt     $a0, $t2, grter         # if val > list[mid] go to grter
        # val < list[mid], update hi
less:   addi    $a3, $t0, -1            # hi = mid-1
        jal     binsrch
        j       done                    # Don't forget this!
        # val > list[mid], update lo
grter:  addi    $a2, $t0, 1             # lo = mid+1
        jal     binsrch
        j       done                    # Don't forget this!
        # val is not in list
not_in: li      $v0, -1                 # val not in list: return -1
        # Prepare for return
done:   lw      $ra, 0($sp)             # Retrieve return address
        addi    $sp, $sp, 4             # Adjust stack pointer
        jr      $ra                     # return
        .data
space:  .asciiz " "
newln:  .asciiz "\n"
```