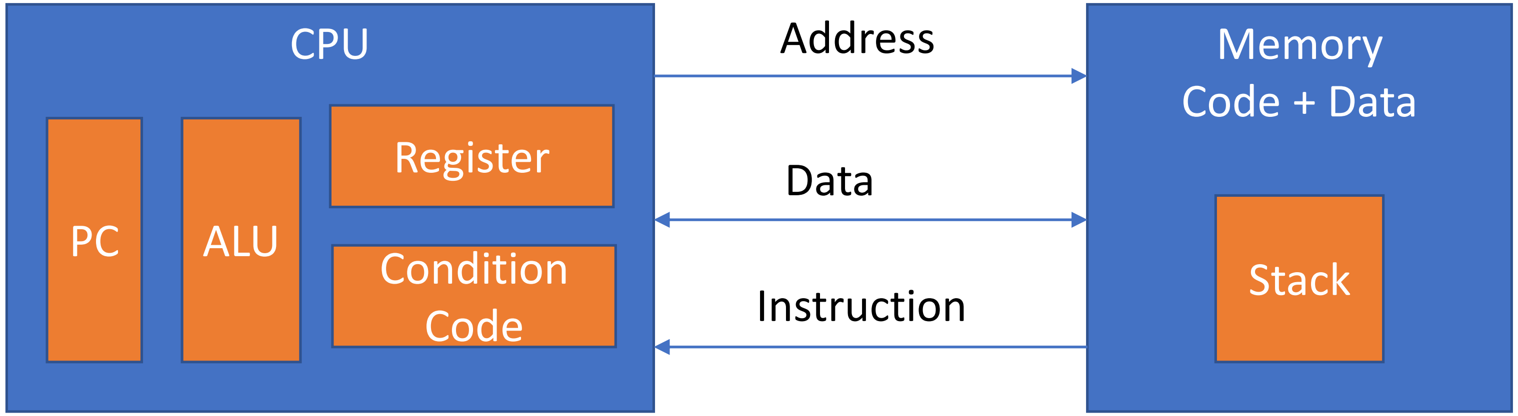
Assembly Code

Machine language

- Machine language:
  - All data and instructions must be encoded as collections of bits (*binary*)
  - Registers store collections of bits
  - Bits are represented as electrical charges (more or less)
  - Control logic and arithmetic operations are implemented as circuits
  - The instructions directly manipulate the underlying hardware
- The collection of all valid binary instructions is known as the *machine language*.

The Abstraction Machine



Program counter (PC): Point to next instruction
Condition codes- helps in taking decision such as condition and loop

https://www.youtube.com/watch?v=cNN_tTXABUA

Instruction set

- Language of machine
- Primitive compared to HLL
- Easily interpreted by hardware

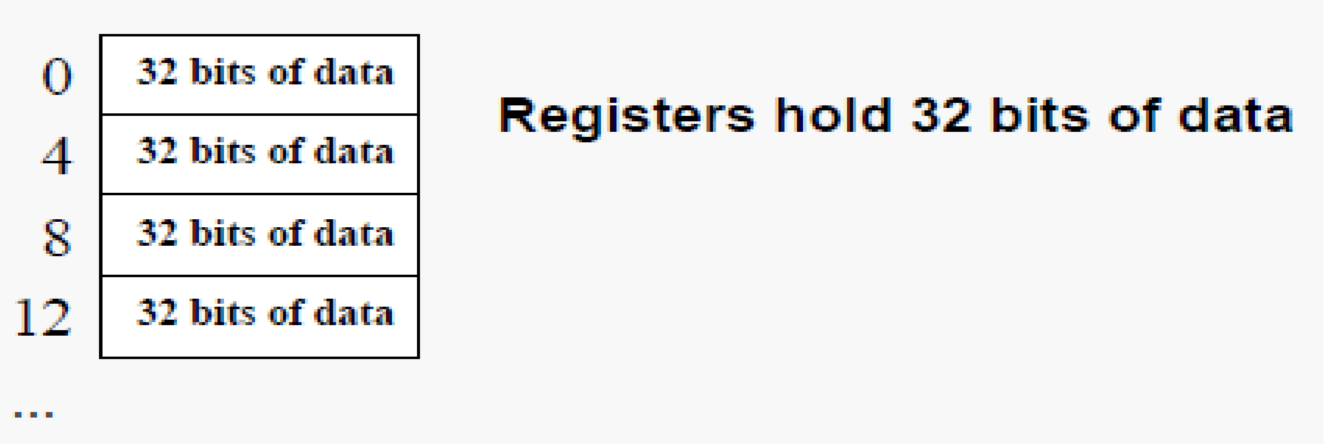
- Instruction set Design
  - Maximize performance
  - Minimize cost
  - Reduce design time

Example of instruction set

- MIPS
  - Real and simple to understand
  - Used by Sony play station, silicon graphics, NEC

MIPS Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a *word* is 32 bits or 4 bytes.
- 2^{32} bytes with byte addresses from 0 to 2^{32} - 1
- Words are *aligned*, that is, each has an address that is a multiple of 4.



MIPS Assembly Hello World

```
# PROGRAM: Hello, World!
        .data               # Begin a data declaration section
msg:    .asciiz   "\nHello, World!\n"

        .text               # Begin a section of assembly language instructions
main:                       # Execution begins with next instruction

        li      $v0, 4        # system call code for printing string = 4
        la      $a0, msg      # load address of string to be printed into $a0
        syscall               # call operating system to perform operation in $v0
                              #     syscall takes its argument from $a0
        li      $v0, 10       # system call code for terminating execution
        syscall
```

Table: System services.

Service	System Call Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_float	6		float (in `$f0`)
read_double	7		double (in `$f0`)
read_string	8	`$a0` = buffer, `$a1` = length	
sbrk	9	`$a0` = amount	address (in `$v0`)
exit	10		
print_character	11	`$a0` = character	
read_character	12		character (in `$v0`)
open	13	`$a0` = filename,	file descriptor (in `$v0`)
		`$a1` = flags, `$a2` = mode	
read	14	`$a0` = file descriptor,	bytes read (in `$v0`)
		`$a1` = buffer, `$a2` = count	
write	15	`$a0` = file descriptor,	bytes written (in `$v0`)
		`$a1` = buffer, `$a2` = count	
close	16	`$a0` = file descriptor	0 (in `$v0`)
exit2	17	`$a0` = value	

```
        .data
str:    .asciiz "the answer = "
        .text
        li $v0, 4           # system call code for print_str
        la $a0, str         # address of string to print
        syscall             # print the string

        li $v0, 1           # system call code for print_int
        li $a0, 5           # integer to print
        syscall             # print it
```

MIPS Register Names

- MIPS assemblers support standard symbolic names for the 32 general-purpose registers:
- $zero		stores value 0; cannot be modified
- $v0-1		used for system calls and procedure return values
- $a0-3		used for passing arguments to procedures
- $t0-9		used for local storage; caller saves
- $s0-7		used for local storage; procedure saves
- $sp		stack pointer
- $fp		frame pointer; primarily used during stack manipulations
- $ra		used to store return address in procedure call
- $gp		pointer to area storing global data (data segment)
- $at		reserved for use by the assembler
- $k0-1		reserved for use by OS kernel

MIPS Arithmetic Instructions

- All arithmetic and logical instructions have 3 operands
- Operand order is fixed (destination first):

  <opcode> <dest>, <leftop>, <rightop>

- Example:

  C code: a = b + c;

  MIPS code: add $s0, $s2, $s3

Example

- C code: a = b + c + d;
- MIPS pseudo-code:

```
add   $s0, $s1, $s2
add   $s0, $s0, $s3
```

- Operands must be registers (or immediate), only 32 registers are provided
- Each register contains 32 bits

Example

- C code:

```
a = b + c - d;
a = a+5;
e = a*3
```

Immediates

- In MIPS assembly, *immediates* are literal constants.
- Many instructions allow immediates to be used as parameters.

```
addi   $t0, $t1, 42        # note the opcode
li     $t0, 42            # actually a pseudo-instruction
```

MIPS Logical Instructions

- Examples:

```
and   $s0, $s1, $s2 # bitwise AND
andi  $s0, $s1, 42
or    $s0, $s1, $s2 # bitwise OR
ori   $s0, $s1, 42
nor   $s0, $s1, $s2 # bitwise NOR (i.e., NOT OR)
sll   $s0, $s1, 10 # logical shift left
srl   $s0, $s1, 10 # logical shift right
```

Conditional Instructions

- MIPS conditional instructions:

```
slt     $t0, $s0, $s1          # $t0 = 1 if $s0 < $s1
                                 (set if less than)
                               # $t0 = 0 otherwise
slti    $t0, $s0, <imm>        # $t0 = 1 if $s0 < imm
                               # $t0 = 0 otherwise
bne     $t0, $t1, <label>      # branch on not-equal
                               # Label if $t0 != $t1
beq     $t0, $t1, <label>      # branch on equal
```

Conditional Instructions

- MIPS conditional instructions:

```
ble   $t0, $t1, <label>      # branch on less or equal
                             # Label if $t0 <= $t1
bge   $t0, $t1, <label>      # branch greater or equal
                             # Label if $t0 >= $t1
```

Unconditional Branch Instructions

- MIPS unconditional branch instructions:

```
j  Label     # PC = Label
b Label      # PC = Label
jr $ra       # PC = $ra
```

- These are useful for building loops and conditional control structures.

Example

```c
if ( i == j )
   h = i + j;
```

```
#$s0 == i, $s1 == j, $s3 == h
bne   $s0, $s1, skip          # test negation of C-test
add   $s3, $s0, $s1           # if-body
skip: ....
```

```
# $s0 == i, $s1 == j, $s3 == h
beq     $s0, $s1, doif        # if-test
b       skip                  # skip if
doif:                         # if-body
        add   $s3, $s0, $s1
skip: ....
```

Example

```
if ( i < j )
        i++;
else
        j++;
```

```
# $s3 == i, $s4 == j
    bge  $s3, $s4, doelse
    addi $s3, $s3, 1                # if-body
    b endelse                       # skip else
doelse:
        addi $s4, $s4, 1            # else-body
endelse:
```

```
# $s3 == i, $s4 == j
    blt  $s3, $s4, do
    b else                          # skip else
do:
        addi $s3, $s3, 1
        b Endif

else:
        addi $s4, $s4, 1            # else-body
Endif:
```

Division operation

div $t1,$t2

Division with overflow : Divide $t1 by $t2 then set LO to quotient and HI to remainder

Use mfhi to access HI,

Use mflo to access LO

Division

Example-

- Write an assembly code to check weather given number is even and odd.

Assembly Code for even odd

Loops

```
int N = 100;
int i = 0;
while ( N > 0 ) {
        N = N / 2;          // N = N >> 1;
        i++;
}
```

While loop

```
int N = 100;
int i = 0;
while ( N > 0 ) {
        N = N / 2;          // N = N >> 1;
        i++;
}
```

First way

```
#$s0 == N, $t0 == i
        li      $s0, 100            # N = 100
        li      $t0, 0              # i = 0
loop:   ble     $s0, $zero, done    # loop test
        srl     $s0, $s0, 1         # calculate N / 2
        addi    $t0, $t0, 1         # i++
        b       loop                # restart loop
done:
```

Second way

```
#$s0 == N, $t0 == i
        li      $s0, 100            # N = 100
        li      $t0, 0              # i = 0
        ble     $s0, $zero, done    # see if loop is necessary
loop:
        srl     $s0, $s0, 1         # calculate N / 2
        addi    $t0, $t0, 1         # i++
        bgt     $s0, $zero, loop    # check whether to restart
done:
```

For loop

```
int Sum = 0;
SLimit = 100;
for (int i = 1; i <= Limit; ++i) {
        Sum = Sum + i*i;
}
```

For loop

```
int Sum = 0, Limit = 100;
for (int i = 1; i <= Limit; ++i)
{
        Sum = Sum + i*i;
}
```

```
#$s0 == Sum, $s1 == Limit, $t0 == i
        li      $s0, 0              # Sum = 0
        li      $s1, 100            # Limit = 0
        li      $t0, 1              # i = 1
loop:   bgt     $t0, $s1, done      # loop test
        mul     $t1, $t0, $t0       # calculate i^2
        add     $s0, $s0, $t1       # Sum = Sum + i^2
        addi    $t0, $t0, 1         # ++i
        b       loop                # restart loop
done:
```

Example:

- Write an assembly code to print number 1-10.

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	See note below table
exit (terminate execution)	10		
print character	11	$a0 = character to print	See note below table
read character	12		$v0 contains character read

Taking input from user

```
.data
msg:        .asciiz     "Enter a number\n"
msg2:      .asciiz     "Your Number"
.text
# Print message (syscall 4)
li              $v0,         4
la              $a0,        msg
syscall
# Read number (syscall 5)
li              $v0,        5              #integer input
syscall
move       $s0,        $v0
li              $v0,        4
la              $a0,        msg2
syscall
# Print number (syscall 0)
move       $a0,        $s0
li $v0, 1
 syscall
li $v0 10
syscall
```

Taking string as user input

```
.data
  ask:  .asciiz "Enter string: "
  ret:  .asciiz "You wrote: "
buffer:   .space 100
 .text
        la     $a0,        ask
        li     $v0,        4
        syscall
        li     $v0,        8
        la     $a0,         buffer
        li     $a1,        100
        syscall
        move $t0,     $a0
        la     $a0,        ret
        li     $v0,        4
        syscall
        li     $v0,        4
        move $a0,$t0
        syscall
        li     $v0         10
        syscall
```

Example:

- Write a MIPS code sum number between given range?
- Example- a=10, b=15
- Then print 75 which is (10+11+12+13+14+15)

```
.data
prompt: .asciiz "enter number"
.text
li          $v0         4
la          $a0,        prompt        # prompt for user input
syscall
#receive input
li           $v0,        5
syscall
add         $s1,        $v0,        $zero     # $s1 = user input
li          $v0,         5
syscall
add         $s2,        $v0,        $zero
loop:
bgt         $s1,        $s2         quit
add         $s0          $s0        $s1
addi        $s1,        $s1,         1
b            loop
quit:
Li           $v0        1
la          $a0,         ($s0)
syscall
li          $v0,        10
syscall
```

Array Declaration and Storage Allocation

- The first step is to reserve sufficient space for the array:

```
.data
list:      .space       1000          # reserves a block of 1000 bytes
```

This yields a contiguous block of bytes of the specified size.

The size of the array is specified in bytes… could be used as:

- array of 1000 char values (ASCII codes)
- array of 250 int values
- array of 125 double values

Array Declaration with Initialization

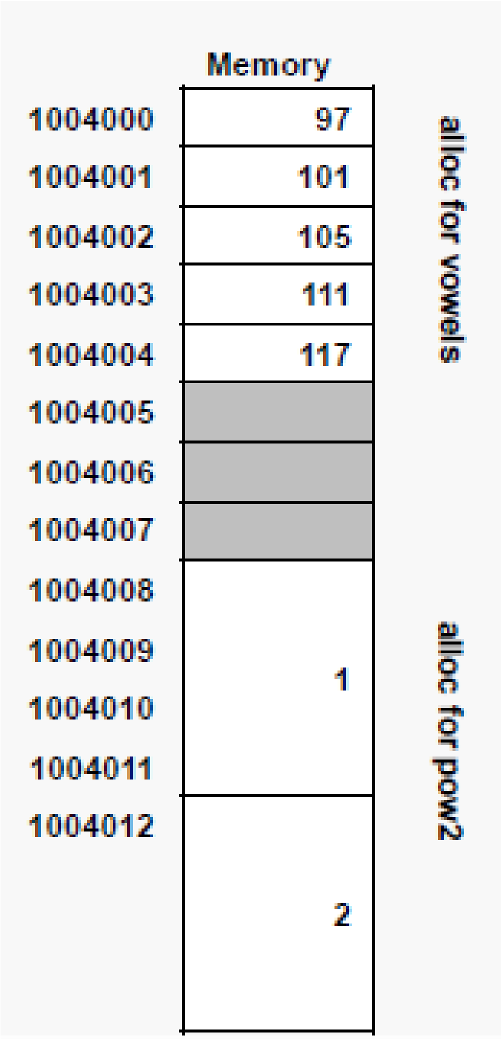
```
.data
vowels: .byte 'a', 'e', 'i', 'o', 'u'
pow2: .word 1, 2, 4, 8, 16, 32, 64, 128
```

`vowels` names a contiguous block of 5 bytes, set to store the given values; each value is stored in a single byte.

Address of `vowels[k] == vowels + k`

`pow2` names a contiguous block of 32 bytes, set to store the given values; each value is stored in a word (4 bytes)

Address of `pow2[k] == pow2 + 4 * k`



Store elements into Array

```
.data
list: .space 1000
listsz: .word 25            # using as array of integers
.text
main: lw $s0, listsz        # $s0 = array dimension
la $s1, list                # $s1 = array address
li $t0, 0                   # $t0 = # elems init'd
beginL: beq $t0, $s0, endL
sw $t0, ($s1)               # list[i] = $to
addi $s1, $s1, 4            # step to next array cell
addi $t0, $t0, 1            # count elem just init'd
b beginL
endL:
li $v0, 10
syscall
```

Retrieve elements from array

```
.data
pow2:  .word 1, 2, 4, 8, 16
.text
li        $s0       5
li        $t0      0
la        $s1       pow2
beginL: beq     $t0       $s0      endL
lw        $s2      ($s1)
li        $v0      1
move   $a0      $s2
syscall
addi     $s1      $s1       4         # step to next array cell
addi     $t0,     $t0      1         # loop count
b beginL
endL:
li        $v0,     10
syscall
```

Procedure

- ($a0-$a3): used to pass arguments
- ($v0-$v1): used to pass return values
- ($ra): used to store the addr of the instruction which is to be executed after the procedure returns

```
main()
{
   int a, b;
   sum(a,b);
    …
}

int sum(int x, int y) {
            return(x+y);
}
```

```
main:   move  $a0,$s0          # x = a
        move  $a1,$s1          # y = b
        jal   sum              #$ra= jump to sum
                 ...
sum:    add $v0,$a0,$a1
        jr  $ra
```

MIPS provides a single instruction called ‘jal’ to
1.Load $ra with addr of next instruction
2.Jump to the procedure.