Instruction Set Architecture, Assembly Code

Machine language

- Machine language:
  - registers store collections of bits
  - all data and instructions must be encoded as collections of bits (*binary*)
  - bits are represented as electrical charges (more or less)
  - control logic and arithmetic operations are implemented as circuits, which are driven by the movement of electrical charges
  - so, the instructions directly manipulate the underlying hardware
- The collection of all valid binary instructions is known as the *machine language*.

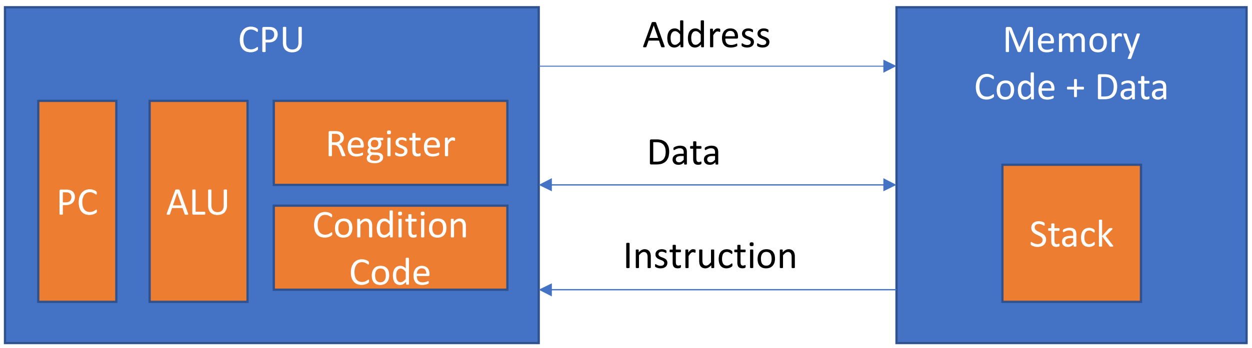
Instruction set Architecture

Layer of Abstraction

Above ISA- How to program a problem

Below ISA- what needs to be built- make it run fast with lower power

The Abstraction Machine



PC- Program counter

PC point to next instruction

Condition codes- helps in taking decision such as condition and loop

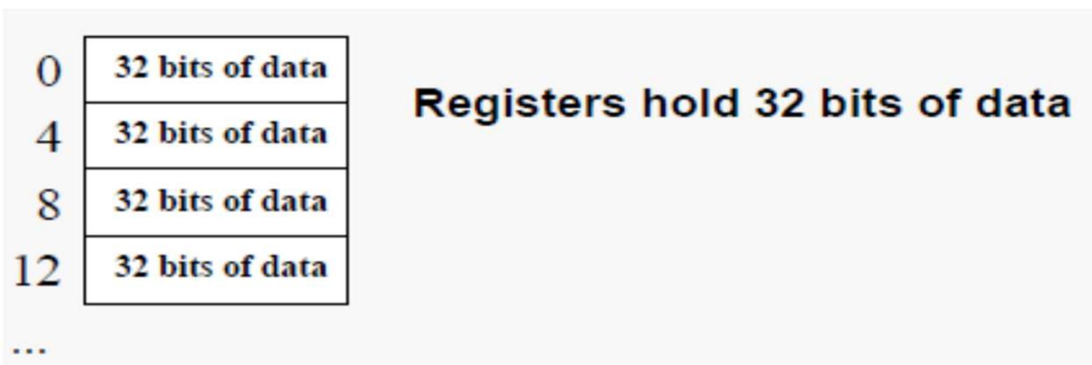
Example of instruction set

- MIPS
  - Real and simple to understand
  - Used by Sony play station, silicon graphics, NEC (National Electrical Code)

- .data
-         mycharecter: .byte 'h'
- .text
-         li $v0, 4
-         la $a0, mycharecter
-         syscall

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
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32} - 4
- 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
```

- .align - Align next data item on specified byte boundary (0=byte, 1=half, 2=word, 3=double)
- .ascii - Store the string in the Data segment but do not add null terminator
- .asciiz - Store the string in the Data segment and add null terminator
- .byte - Store the listed value(s) as 8 bit bytes
- .data - Subsequent items stored in Data segment at next available address
- .double - Store the listed value(s) as double precision floating point
- .end_macro - End macro definition. See .macro
- .eqv - Substitute second operand for first. First operand is symbol, second operand is expression (like #define)
- .extern - Declare the listed label and byte length to be a global data field
- .float - Store the listed value(s) as single precision floating point
- .globl - Declare the listed label(s) as global to enable referencing from other files
- .half - Store the listed value(s) as 16 bit halfwords on halfword boundary
- .include - Insert the contents of the specified file. Put filename in quotes.
- .kdata - Subsequent items stored in Kernel Data segment at next available address
- .ktext - Subsequent items (instructions) stored in Kernel Text segment at next available address
- .macro - Begin macro definition. See .end_macro
- .set - Set assembler variables. Currently ignored but included for SPIM compatability
- .space - Reserve the next specified number of bytes in Data segment
- .text - Subsequent items (instructions) stored in Text segment at next available address
- .word - Store the listed value(s) as 32 bit words on word boundary

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

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	

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

Types of memory

- Reserved - This is memory which is reserved for the MIPS platform.

- Program text - (Addresses 0x0040 0000 - 0x1000 00000) This is where the machine code representation of the program is stored. Each instruction is stored as a word (32 bits or 4 byte) in this memory.

- Static data - (Addresses 0x1001 0000 - 0x1004 0000) This is data which will come from the data segment of the program. The size of the elements in this section are assigned when the program is created (assembled and linked) and cannot change during the execution of the program.

Types of memory

- Heap - (Addresses 0x1004 0000 - until stack data is reached, grows upward) Heap is dynamic data which is allocated on an as-needed basis at run time (e.g. with a new operator in Java).

- Stack – (Addresses 0x7fff fe00 - until heap data is reached, grows downward) The program stack is dynamic data allocated for subprograms via push and pop operations. All method local variables are stored here. Because of the nature of the push and pop operations, the size of the stack record to create must be known when the program is assembled.

- Kernel - (Addresses 0x9000 0000 - 0xffff 0000) - Kernel memory is used by the operating system, and so is not accessible to the user

MIPS instruction formats

Every assembly language instruction is translated into a machine code instruction in one of three **formats**

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits = 32 bits
R	000000	rs	rt	rd	shamt	funct
I	op	rs	rt	address/immediate		
J	op	target address				

- **R**egister-type
- **I**mmediate-type
- **J**ump-type

Example instructions for each format

Register-type instructions

```
# arithmetic and logic
add   $t1, $t2, $t3
or    $t1, $t2, $t3
slt   $t1, $t2, $t3

# mult and div
mult  $t2, $t3
div   $t2, $t3

# move from/to
mfhi  $t1
mflo  $t1

# jump register
jr    $ra
```

Immediate-type instructions

```
# immediate arith and logic
addi  $t1, $t2, 345
ori   $t1, $t2, 345
slti  $t1, $t2, 345

# branch and branch-zero
beq   $t2, $t3, label
bne   $t2, $t3, label
bgtz  $t2, label

# load/store
lw    $t1, 345($t2)
sw    $t2, 345($t1)
lb    $t1, 345($t2)
sb    $t2, 345($t1)
```

Jump-type instructions

```
# unconditional jump
j     label
```

```
# jump and link
jal   label
```

Components of an instruction

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R	000000	rs	rt	rd	shamt	funct
I	op	rs	rt	address/immediate		
J	op	target address				

Component	Description
op, funct	codes that determine operation to perform
rs, rt, rd	register numbers for args and destination
shamt, imm, addr	values embedded in the instruction

Components of an R-type instruction

R:	000000	rs	rt	rd	shamt	funct

R-type instruction

• op	6 bits	always zero!
• rs	5 bits	1st argument register
• rt	5 bits	2nd argument register
• rd	5 bits	destination register
• shamt	5 bits	used in shift instructions (for us, always 0s)
• funct	6 bits	code for the operation to perform
	32 bits	

Note that the destination register is third in the machine code!

Assembling an R-type instruction

`add $t1, $t2, $t3`

000000	rs	rt	rd	shamt	funct

rs	=	10	(`$t2` = `$10`)
rt	=	11	(`$t3` = `$11`)
rd	=	9	(`$t1` = `$9`)
funct	=	32	(look up function code for `add`)
shamt	=	0	(not a shift instruction)

000000	10	11	9	0	32
000000	01010	01011	01001	00000	100000

`0000 0001 0100 1011 0100 1000 0010 0000`

`0x014B4820`

Exercises

R:

0	rs	rt	rd	sh	fn

Name	Number
`$zero`	0
`$v0–$v1`	2–3
`$a0–$a3`	4–7
`$t0–$t7`	8–15
`$s0–$s7`	16–23
`$t8–$t9`	24–25
`$sp`	29
`$ra`	31

Assemble the following instructions:

- `sub  $s0, $s1, $s2`
- `mult $a0, $a1`
- `jr   $ra`

Instr	fn
`add`	32
`sub`	34
`mult`	24
`div`	26
`jr`	8

Components of an I-type instruction

I:	op	rs	rt	address/immediate

I-type instruction

- op — 6 bits — code for the operation to perform
- rs — 5 bits — 1st argument register
- rt — 5 bits — destination or 2nd argument register
- imm — 16 bits — constant value embedded in instruction

32 bits

Note the destination register is second in the machine code!

Assembling an I-type instruction

`addi $t4, $t5, 67`

op	rs	rt	address/immediate

op = 8 (look up op code for `addi`)
rs = 13 (`$t5` = `$13`)
rt = 12 (`$t4` = `$12`)
imm = 67 (constant value)

8	13	12	67
001000	01101	01100	0000 0000 0100 0011

`0010 0001 1010 1100 0000 0000 0100 0011`

`0x21AC0043`

Exercises

R:	0	rs	rt	rd	sh	fn

I:	op	rs	rt	addr/imm

Assemble the following instructions:

- `or  $s0, $t6, $t7`
- `ori $t8, $t9, 0xFF`

Name	Number
`$zero`	0
`$v0–$v1`	2–3
`$a0–$a3`	4–7
`$t0–$t7`	8–15
`$s0–$s7`	16–23
`$t8–$t9`	24–25
`$sp`	29
`$ra`	31

Instr	op/fn
`and`	36
`andi`	12
`or`	37
`ori`	13

Conditional branch instructions

```
beq $t0, $t1, label
```

I:	op	rs	rt	address/immediate

I-type instruction

• op	6 bits	code for the comparison to perform
• rs	5 bits	1st argument register
• rt	5 bits	2nd argument register
• imm	16 bits	**jump offset** embedded in instruction
	32 bits	

Calculating the jump offset

Jump offset

Number of instructions from the **next instruction**

(`nop` is an instruction that does nothing)

```
      beq  $t0, $t1, skip
      nop  # 0 (start here)
      nop  # 1
      nop  # 2
skip: nop  # 3!
      ...
```

offset = 3

```
loop: nop  # -5
      nop  # -4
      nop  # -3
      nop  # -2
      beq  $t0, $t1, loop
      nop  # 0 (start here)
```

offset = -5

Assembling a conditional branch instruction

```
        beq  $t0, $t1, label
        nop
        nop
label:  nop
```

op	rs	rt	address/immediate

op = 4 (look up op code for **beq**)
rs = 8 (**$t0** = **$8**)
rt = 9 (**$t1** = **$9**)
imm = 2 (jump offset)

4	8	9	2
000100	01000	01001	0000 0000 0000 0010

0001 0001 0000 1001 0000 0000 0000 0010

0x11090002

Exercises

R:	0	rs	rt	rd	sh	fn
I:	op	rs	rt	addr/imm		

Name	Number
`$zero`	0
`$v0–$v1`	2–3
`$a0–$a3`	4–7
`$t0–$t7`	8–15
`$s0–$s7`	16–23
`$t8–$t9`	24–25
`$sp`	29
`$ra`	31

Assemble the following program:

```
# Pseudocode:
#   do {
#     i++
#   } while (i != j);
loop:  addi  $s0, $s0, 1
       bne   $s0, $s1, loop
```

Instr	op/fn
`add`	32
`addi`	8
`beq`	4
`bne`	5

Program Termination

- Unlike the high-level languages you are familiar to, MIPS assembly does not include an instruction, or block syntax, to terminate the program execution.
- MIPS programs can be terminated by making a system call:

```
## Exit
        li $v0, 10          # load code for exit system call in $v0
        syscall             # make the system call to exit
```

- Without such code, the system could attempt to continue execution into the memory words that followed the final instructions of the program.

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)
- 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
                               # $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

```
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

```
.data
msg: .asciiz "\n"
.text
li          $t0 25
li          $t1 4
div         $t0 $t1    # division operation
mfhi        $t0        # transfer remainder  from Hi register
mflo        $t1         # transfer quotient  from Lo register
Li          $v0, 1     # printing remainder
la          $a0, ($t0)
syscall
li          $v0, 4     # print \n,  go to next line
la          $a0, msg
syscall
li          $v0, 1      # printing remainder
la $a0, ($t1)
syscall
li $v0, 10             # exit program
syscall
```

Example-

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

Assembly Code for even odd

```
.data
even:       .asciiz      "\nEven"
odd:        .asciiz      "\nOdd"
.text
li          $t0          25
li          $t1          2
div         $t0          $t1
mfhi        $t0
li          $v0,         1
la          $a0,         ($t0)
syscall
li          $v0,         4
bne         $t0          $zero       printodd
la          $a0,         even
b           endif
printodd:
            la           $a0         odd
endif:
syscall
li          $v0,         10
syscall
```

Loops

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

While loop

```c
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;
Limit = 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.

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

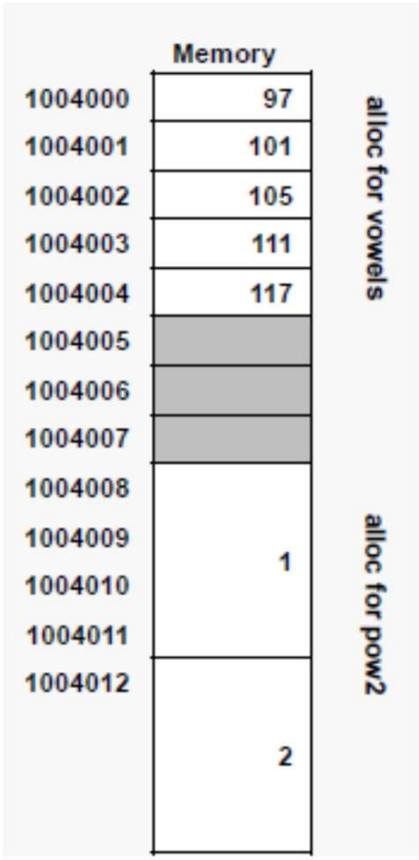
```
.data
vowels: .byte 'a', 'e', 'i', 'o', 'u'
pow2: .word 1, 2, 4, 8, 16, 32, 64, 128
```

`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

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

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

– ($a0-$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:   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.