

CPS104 Computer Organization

Lecture 8

MIPS Instruction Set Architecture; MIPS Assembler

September 16 , 2009

Gershon Kedem

Administratrivia

- **Programming Puzzle-2 is on the Web, see:**
kedem.cs.duke.edu/cps104/ppz/PPZ2.html
- **Due: Today (11:59 pm).**
- **Submit the *.java file to cps104/PPZ2**
- **Use the Eclipse system to submit the file.**
- **Help is on-line.**
- **Reading: Chapter 2.**
- **Reading: Appendix B**
 - ◆ Available on: . . . cps104/Handouts/spim_appendix.pdf

Review: MIPS Arithmetic Instructions

<u>Instruction</u>	<u>Example</u>	<u>Meaning</u>	<u>Comments</u>
add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands
subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands
add immediate	addi \$1,\$2,100	$\$1 = \$2 + 100$	+ constant
add unsigned	addu \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands
subtract unsigned	subu \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands
add imm. unsign.	addiu \$1,\$2,100	$\$1 = \$2 + 100$	+ constant
multiply	mult \$2,\$3	Hi, Lo = $\$2 \times \3	64-bit signed product
multiply unsigned	multu \$2,\$3	Hi, Lo = $\$2 \times \3	64-bit unsigned product
divide	div \$2,\$3	Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3	Lo = quotient, Hi = remainder
divide unsigned	divu \$2,\$3	Lo = $\$2 \div \3 , Hi = $\$2 \bmod \3	Unsigned quotient Unsigned remainder
Move from Hi	mfhi \$1	$\$1 = \text{Hi}$	Used to get copy of Hi
Move from Lo	mflo \$1	$\$1 = \text{Lo}$	Used to get copy of Lo

Which add for address arithmetic? Which for integers?

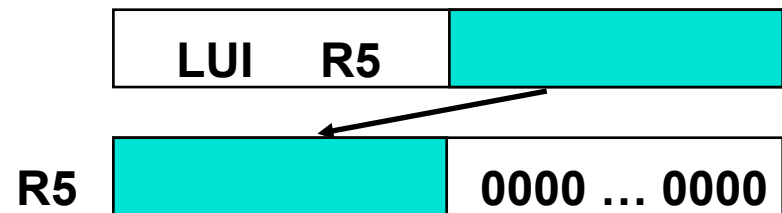
MIPS Logical Instructions

<u>Instruction</u>	<u>Example</u>	<u>Meaning</u>	<u>Comment</u>
and	and \$1,\$2,\$3	$\$1 = \$2 \& \$3$	Bitwise AND
or	or \$1,\$2,\$3	$\$1 = \$2 \mid \$3$	Bitwise OR
xor	xor \$1,\$2,\$3	$\$1 = \$2 \oplus \$3$	Bitwise XOR
nor	nor \$1,\$2,\$3	$\$1 = \sim(\$2 \mid \$3)$	Bitwise NOR
and immediate	andi \$1,\$2,10	$\$1 = \$2 \& 10$	Bitwise AND reg, const
or immediate	ori \$1,\$2,10	$\$1 = \$2 \mid 10$	Bitwise OR reg, const
xor immediate	xori \$1, \$2,10	$\$1 = \sim\$2 \& \sim 10$	Bitwise XOR reg, const
shift left logical	sll \$1,\$2,10	$\$1 = \$2 \ll 10$	Shift left by constant
shift right logical	srl \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right by constant
shift right arithm.	sra \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right (sign extend)
shift left logical	sllv \$1,\$2,\$3	$\$1 = \$2 \ll \$3$	Shift left by var
shift right logical	srlv \$1,\$2, \$3	$\$1 = \$2 \gg \$3$	Shift right by var
shift right arithm.	srav \$1,\$2, \$3	$\$1 = \$2 \gg \$3$	Shift right arith. by var

MIPS Data Transfer Instructions

<u>Instruction</u>	<u>Comment</u>
SW R3, 500(R4)	Store word
SH R3, 502(R2)	Store half
SB R2, 41(R3)	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load half word
LHU R1, 40(R3)	Load half word unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)

Why do we need LUI?



MIPS Compare and Branch

Compare and Branch

beq *rs, rt, offset* if $R[rs] == R[rt]$ then PC-relative branch

bne *rs, rt, offset* \neq

Compare to zero and Branch

blez *rs, offset* if $R[rs] \leq 0$ then PC-relative branch

bgtz *rs, offset* $>$

bltz *rs, offset* $<$

bgez *rs, offset* \geq

bltzal *rs, offset* if $R[rs] < 0$ then branch and link (into R 31)

bgeal *rs, offset* \geq

- Remaining set of compare and branch take two instructions
- Almost all comparisons are against zero!

MIPS jump, branch, compare instructions

<u>Instruction</u>	<u>Example</u>	<u>Meaning</u>
branch on equal	<code>beq \$1,\$2,100</code>	if ($\$1 == \2) go to PC+4+100 Equal test; PC relative branch
branch on not eq.	<code>bne \$1,\$2,100</code>	if ($\$1 \neq \2) go to PC+4+100 Not equal test; PC relative
set on less than	<code>slt \$1,\$2,\$3</code>	if ($\$2 < \3) $\$1=1$; else $\$1=0$ Compare less than; 2's comp.
set less than imm.	<code>slti \$1,\$2,100</code>	if ($\$2 < 100$) $\$1=1$; else $\$1=0$ Compare < constant; 2's comp.
set less than uns.	<code>sltu \$1,\$2,\$3</code>	if ($\$2 < \3) $\$1=1$; else $\$1=0$ Compare less than; natural numbers
set l. t. imm. uns.	<code>sltiu \$1,\$2,100</code>	if ($\$2 < 100$) $\$1=1$; else $\$1=0$ Compare < constant; natural numbers
jump	<code>j 10000</code>	go to 10000 Jump to target address
jump register	<code>jr \$31</code>	go to \$31 For switch, procedure return
jump and link	<code>jal 10000</code>	$\$31 = PC + 4$; go to 10000 For procedure call

Signed v.s Unsigned Comparison

R1= 0...00 0000 0000 0000 0001

R2= 0...00 0000 0000 0000 0010

R3= 1...11 1111 1111 1111 1111

• After executing these instructions:

```
slt  r4,r2,r1
```

```
slt  r5,r3,r1
```

```
sltu r6,r2,r1
```

```
sltu r7,r3,r1
```

• What are values of registers r4 - r7? Why?

r4 = ; r5 = ; r6 = ; r7 = ;

Multiply / Divide

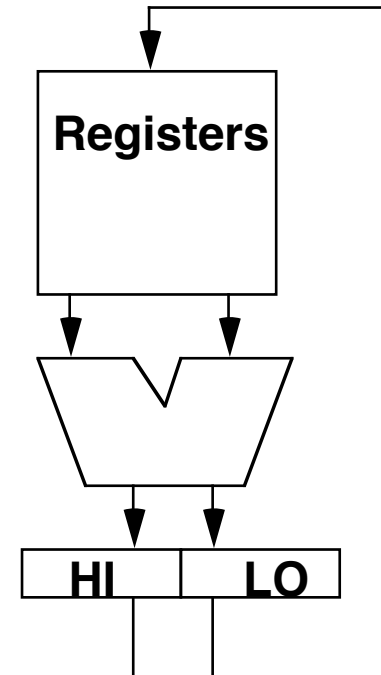
Start multiply, divide

mult rs, rt

mfhi rd

mflo rd

Move from HI or LO



Why not Third field for destination?

(Hint: how many clock cycles for multiply or divide vs. add?)

Summary

- **MIPS has 5 categories of instructions**
 - ◆ **Arithmetic, Logical, Data Transfer, Conditional Branch, Unconditional Jump**
- **3 Instruction Formats: R-type, I-type, J-type.**

Next:

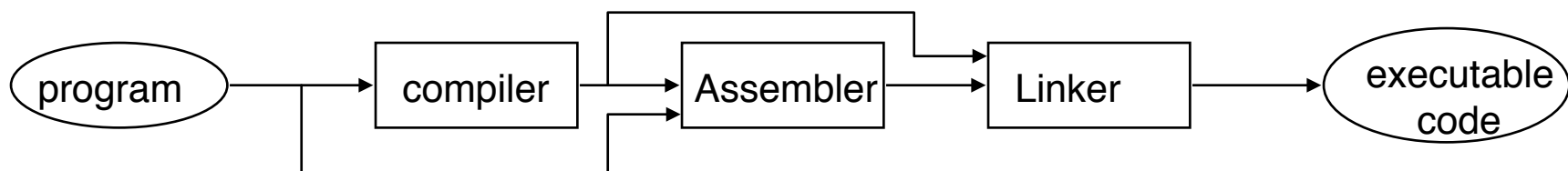
- **Assembly Programming**

Reading

- **Ch. 2, Appendix A**

Assembler and Assembly Language

- Machine language is a sequence of binary words.
- **Assembly language** is a **text representation** for machine language plus extras that make assembly language programming easier (more readable too!).



The SPIM Simulator

- **SPIM** is a simulator that let you run and debug MIPS assembler programs.
- **XSPIM** is an X-window version of the simulator.
- The simulator allows you to look at the content of registers and memory, and to single step through the simulation.
- Documentation in postscript format and PDF format are in:
<http://kedem.cs.duke.edu/cps104/Handouts.html>
- Also available a MAN page for **SPIM**.
- Try to run the provided example of an simple assembler program:
<http://kedem.cs.duke.edu/cps104/Handouts/MIPS.asm>

Assembly Language

- One instruction per line.
- **Numbers**: are base-10 integers or Hex.
- **Identifiers**: alphanumeric, `_`, string starting in a letter or `_`
- **Labels**: identifiers starting at the beginning of a line followed by “:”
- **Comments**: everything following `#` till end-of-line.
- **Instruction format**: Space and “,” separated fields.
 - ◆ `[Label:] <op> Arg1, [Arg2], [Arg3] [# comment]`
 - ◆ `[Label:] <op> arg1, offset(reg) [# comment]`
 - ◆ `.Directive [arg1], [arg2], ...`

Assembly Language (cont.)

- **Pseudo-instructions: extending the instruction set for convenience.**

- **Examples:**

- ◆ `move $2, $4`
Translates to:
`add $2, $4, $0`

- `# $2 = $4, (copy $4 to $2)`

- ◆ `li $8, 40`
`addi $8, $0, 40`

- `# $8 = 40, (load 40 into $8)`

- ◆ `sd $4, 0($29)`
`sw $4, 0($29)`
`sw $5, 4($29)`

- `# mem[$29] = $4; Mem[$29+4] = $5`

- ◆ `la $4, 0x1000056c`
`lui $4, 0x1000`
`ori $4, $4, 0x056c`

- `# Load address $4 = <address>`

Assembly Language (cont.)

- The assembler is “smart enough” to figure out what instructions to use depending on the arguments’ types.
 - ◆ **For Example:**
 - ◆ `add $1, $2, -345` will be translated into
`addi $1, $2, -345`
 - ◆ `div $1, $2, $3` will be translated into
`div $2, $3`
`mflo $1`
- The assembler also translates labels into addresses.
- **For Example** one can use:
 - ◆ `la $1, array1` to load the address that the label `array1` is attached to.
 - ◆ `bne $2, $4, loop1` to branch into the instruction that has the label `loop1`

Assembly Language; Directives

✱ **Directives:** `"."<string> [arg1], [arg2] . . .`

✱ **Examples:**

- ◆ `.align n # align datum on 2n byte boundary.`
- ◆ `.ascii "<string>" # store a string in memory.`
- ◆ `.asciiz "<string>" # store a null terminated string in memory`
- ◆ `.data [address] # start a data segment.
 # [optional beginning address]`
- ◆ `.text [address] # start a code segment.`
- ◆ `.word w1, w2, . . . , wn # store n words in memory.`

The routine written in C

```
#include <stdio>

int main ( )
{
    int i;
    int sum = 0;
    for(i=0; i <= 100; i++) sum = sum + i*i ;
    printf("The sum from 0 .. 100 is %d\n", sum) ;
}
```

Assembly Language Example1:

```
.text
.align 2
main:
    la    $10, Temp
loop:
    lw    $14, 4($10)
    mul   $15, $14, $14
    lw    $24, 0($10)
    add   $25, $24, $15
    sw    $25, 0($10)
    addi  $8, $14, 1
    sw    $8, 4($10)
    ble  $8, 100, loop
    la    $4, str
    lw    $5, 0($10)
    jal   printf
```

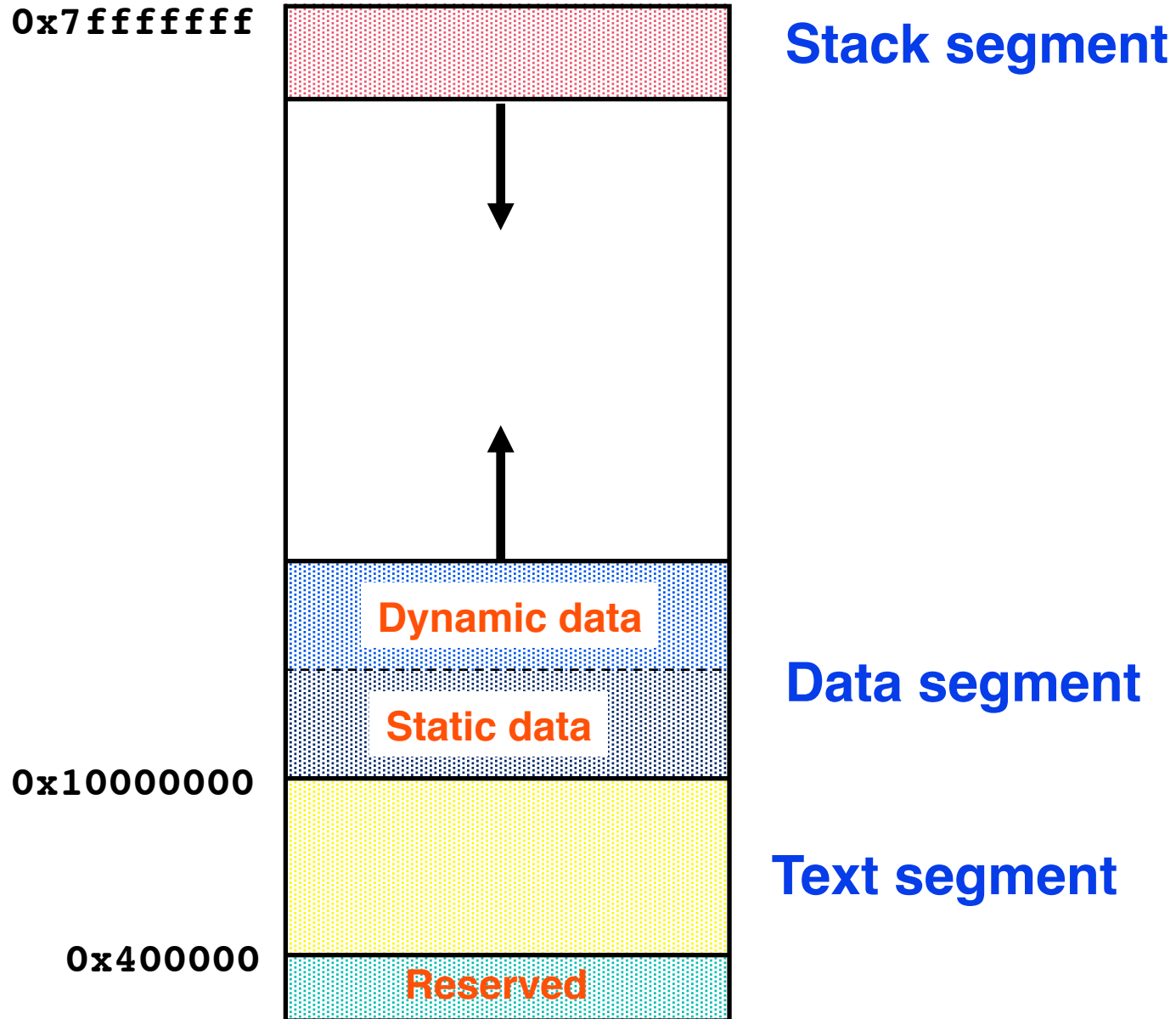
```
li    $2, 10
syscall    # Exit
```

```
.data
.align 2
Temp: .word 0, 0
str:
    .asciiz "The sum from 0 .. 100 is
%d\n"
```

MIPS: Software conventions for Registers

0	zero	constant 0	16	s0	callee saves
1	at	reserved for assembler	...		
2	v0	expression evaluation &	23	s7	
3	v1	function results	24	t8	temporary (cont'd)
4	a0	arguments	25	t9	
5	a1		26	k0	reserved for OS kernel
6	a2		27	k1	
7	a3		28	gp	Pointer to global area
8	t0	temporary: caller saves	29	sp	Stack pointer
...			30	fp	frame pointer
15	t7		31	ra	Return Address (HW)

Memory Layout



Example2

```
# Example for CPS 104
# Program to add together list of 9 numbers.

        .text                # Code
        .align 2
        .globl main

main:                    # MAIN procedure
Entrance
sw      $s0, 20($sp)      # /
        move      $v0, $0      #/ initialize exit
code to 0

        move      $s1, $0      #\
        la       $s0, list      # \ Initialization
        la       $s2, msg       # /
        la       $s3, list+36   #/
```

Example2 (cont.)

```
#                               Main code segment

again:                            #   Begin main loop
    lw      $t6, 0($s0)          #\
    add     $s1, $s1, $t6       #/   Actual "work"
                                       #   SPIM I/O
    li      $v0, 4              #\
    move    $a0, $s2            # >   Print a string
    syscall                               #/
    li      $v0, 1              #\
    move    $a0, $s1            # >   Print a number
    syscall                               #/
    li      $v0, 4              #\
    la      $a0, nln            # >   Print a string (eol)
    syscall                               #/

    addiu   $s0, $s0, 4         #\   index update and
    bne     $s0, $s3, again     #/   end of loop
```

Example2 (cont.)

```
#                               Exit Code

    li    $v0, 10                #\    load exit code ;
    syscall                # |    exit
    .end    main                #/    end of program

#                               Data Segment

    .data                        # Start of data segment
list:    .word    35, 16, 42, 19, 55, 91, 24, 61, 53
msg:     .asciiz  "The sum is "
nl:      .asciiz  "\n"
```

System call

- * System call is used to communicate with the system and do simple I/O.
- * Load system call code into Register **\$v0**
- * Load arguments (if any) into registers **\$a0**, **\$a1** or **\$f12** (for floating point).
- * do: **syscall**
- * Results returned in registers **\$v0** or **\$f0**.

code	service	Arguments	Result	comments
1	print integer	\$a0		
2	print float	\$f12		
3	print double	\$f12,\$f13		
4	print string	\$a0		(address)
5	read integer		integer in \$v0	
6	read float		float in \$f0	
7	read double		double in \$f0	
8	read string	\$a0=buffer, \$a1=length		
9	sbrk	\$a0=amount	address in \$v0	
10	exit			

Details of the MIPS instruction set

- Register zero always has the value zero (even if you try to write it)
- Link instructions put the return address PC+4 into the link register
- All instructions change all 32 bits of the destination register (including `lui`, `lb`, `lh`) and all read all 32 bits of sources (`add`, `sub`, `and`, `or`, ...)
- Immediate arithmetic and logical instructions are extended as follows:
 - ♦ logical immediate are zero extended to 32 bits
 - ♦ arithmetic immediate are sign extended to 32 bits
- The data loaded by the instructions `lb` and `lh` are extended as follows:
 - ♦ `lbu`, `lhu` are zero extended
 - ♦ `lb`, `lh` are sign extended
- Overflow can occur in these arithmetic and logical instructions:
 - ♦ `add`, `sub`, `addi`
 - ♦ it cannot occur in: `addu`, `subu`, `addiu`, `and`, `or`, `xor`, `nor`, `shifts`, `mult`, `multu`, `div`, `divu`

Miscellaneous MIPS Instructions

- **break** **A breakpoint trap occurs, transfers control to exception handler**
- **syscall** **A system trap occurs, transfers control to exception handler**
- **coprocessor instrs.** **Support for floating point.**
- **TLB instructions** **Support for virtual memory: discussed later**
- **restore from exception** **Restores previous interrupt mask & kernel/user mode bits into status register**
- **load word left/right** **Supports misaligned word loads**
- **store word left/right** **Supports misaligned word stores**

Assembly Examples: code fragments

★ **If (...) { ... };**

Example

The C++ code

```
a=b=c=0;  
if (i < 5) {  
  a = 1;  
  b = 2;  
  c = 3;  
}  
d = 5;
```

The Assembler code

```
move $a0,0    # a=0  
move $a1,0    # b=0  
move $a2,0    # c=0  
bge  $s0,5,flbl # if i >=5  
                        # goto flbl  
  
move $a0,1    # a=1  
move $a1,2    # b=2  
move $a2,3    # c=3  
  
flbl: move $a3,5 # d=5
```

Assembly Examples: code fragments

* If (...) { ... } else { ... } ; Example

The C++ code

```
a=b=c=0;
if (i < 5) {
a = 1;
b = 2;
c = 3;
}
else{
a = 4;
b = 5;
c = 6;
}
d = 5;
```

The Assembler code

```
move $a0,0 # a=0
move $a1,0 # b=0
move $a2,0 # c=0
bge $s0,5,lb11 # if i >=5
# goto lb11

move $a0,1 # a=1
move $a1,2 # b=2
move $a2,3 # c=3
j lb12 # goto lb12
lb11: move $a0,4 # a=4
move $a1,5 # b=5
move $a2,6 # c=6
lb12: move $a3,5 # d=5
```

Assembly Examples: code fragments

★ `for(i=0; i<100; i++) sum=sum+A[i];` **Example-1:**

The C++ code

```
int a[100];  
.  
.  
.  
sum = 0;  
for(i=0; i<100; i++)  
    sum=sum+a[i];
```

The Assembler code

```
la    $t0,a           # $t0 = &a[0]  
move  $a0,0          # sum=0  
move  $a1,0          # i=0  
move  $a3,100        # $a3 = 100  
loop: mult $a2,$a1,4  # $a2=i*4  
      addu $t1,$t0,$a2 # $t1=&a[i]  
      lw   $t2,0($t1)  # $t2=a[i]  
      add  $a0,$a0,$t2 # sum=sum+a[i]  
      addi $a1,$a1,1   # i++  
      blt  $a1,$a3,loop # if i<100  
                          # goto loop
```

Assembly Examples: code fragments

★ `for(i=0; i<100; i++) sum=sum+A[i];` **Example-2:**

The C++ code

```
int a[100];
int* p=&a[0];
sum = 0;
for(i=0;i<100;i++)
{sum=sum+*p;
  p++;
}
```

The Assembler code

```
la    $t0,a           # p = &a[0]
move  $a0,0           # sum=0
move  $a1,0           # i=0
move  $a3,100         # $a3 = 100
loop: lw    $t2,0($t0) # $t2=a[i]
      add   $a0,$a0,$t2 # sum=sum+a[i]
      addi  $a1,$a1,1   # i++
      addui $t0,4       # p++
      blt  $a1,$a3,loop # if i<100
                          # goto loop
```