Addressing modes in SPARC

	M68000
(1) immediate	#n
(2) register direct	Dn An
(3) memory direct	n
(4) indirect / indirect with offset	(An) , d16(An).

Discussion order:

(1) register direct

(2) immediate

(3) memory direct

(4) indirect.

Moving Data between registers — Register direct

- SPARC does not have any move instruction.
- To move data in register r1 to register r2, we use:

  add %g0, %[illegible] , %r2.

  Since register g0 is always 0, r1 + 0 = r1 and we effectively moved r1 to r2.

- We can also move small constants (between $(-2^{12}$ and $2^{12}-1)$ to a register:

  add %g0 , n , %r2 .

- Although there is no move instruction, the SPARC assembler recognizes the mnemonic mov:

  $$\text{mov} \quad \left\{\begin{matrix}\%r1 \\ \text{constant}\end{matrix}\right\}, \%r2$$

  It will translate it to

  $$\text{or} \quad \%g0 , \left\{\begin{matrix}\%r1 \\ \text{constant}\end{matrix}\right\}, \%r2$$

Q: How to move ~~constant~~ values that are larger than $2^{12}-1$ or smaller than -2^{12} to register ?

Ans: eg: 60000 to 06

```
SETHI   %hi(60000), %06
add     %06, %lo(60000), %06.
```

or:

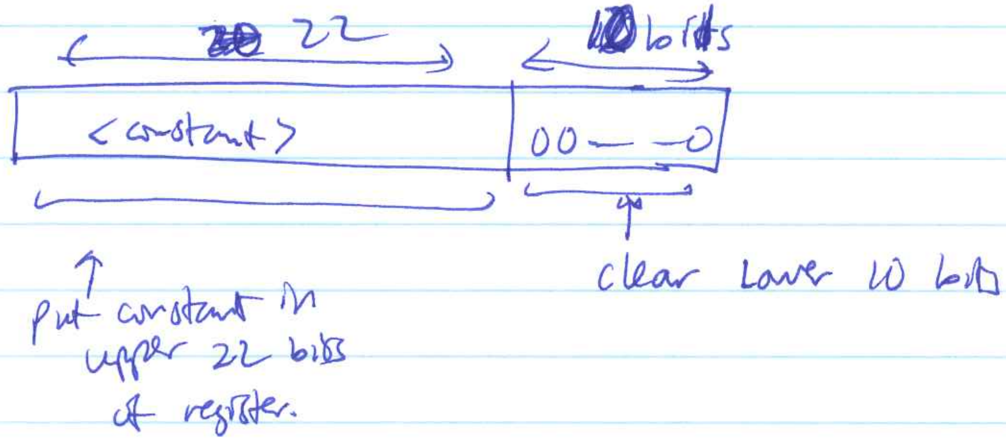
```
set 60000, %06.
```

Talk about SETHI first!

Instruction:

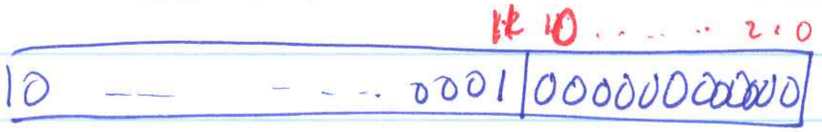
sethi <constant>, %reg.

effect:



eg: sethi 1, %l0

result:



value = ~~1,024~~ ie. 2^{10} !!!

= ~~4096~~ 1024.

Demo: sparc0-SETHI.s

Helpful operators: in assembler

%hi(value) = value / (2^{10}) = 1024

%lo(value) = value % 2^{10}

eg:

~~4050~~ ~~= 4096 + 4.~~

Demo: sparcØ-HI-LO.s

~~%hi(4050) = 1~~

~~%lo(4050) = 4.~~

1028 = 1024 + 4.

%hi(1028) = 1028 / 1024
= 1

%lo(1028) = 1028 % 1024
= 4.

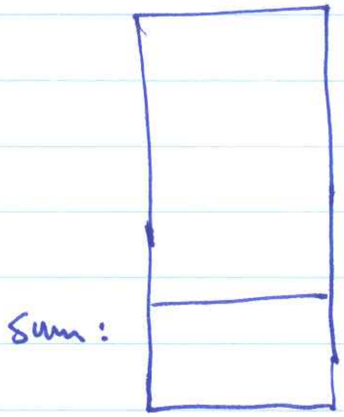
Demo: sparcØ-SETHI2.s

immediate addressing

this too

Another application of SETHI: move the ~~addr~~. to a register
(constant)

Example:



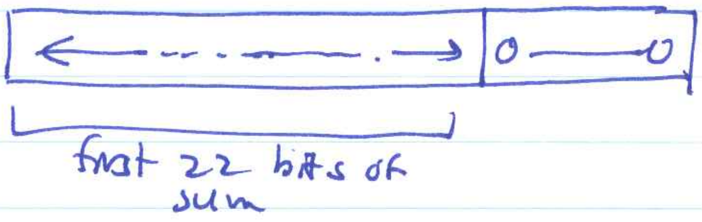
we want to move the address of sum to l0

In M68000, you would write: move.l #sum, a0

In SPARC:

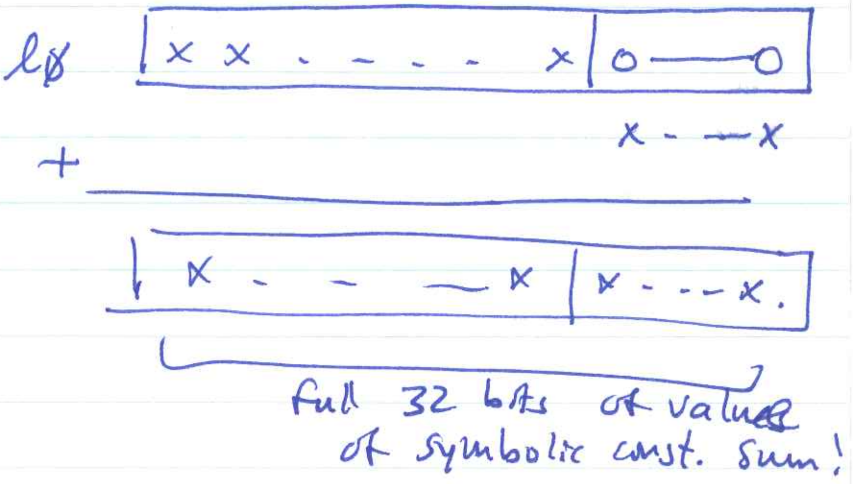
```
SETHI %hi(sum), %l0
```

result: l0



```
add %l0, %lo(sum), %l0
```

result: l0



· Because this is an important application, the assembler provides a macro for the programmer to move the address of a variable to a register:

```
set label, %r1
```

It translates to:

```
sethi %hi(label), %r1
or    %r1, %lo(label), %r1.
```

~~demos/sparc-sethi.s~~
sparc2-IMM-ADDR.s

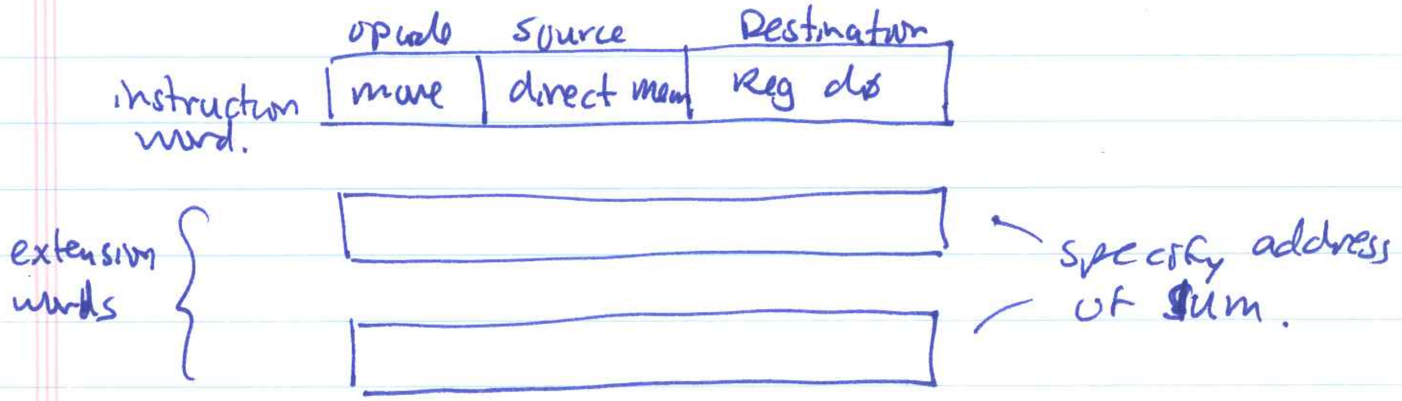
Direct memory addressing (this second)!   immediate

Recall: direct memory addressing in M68000:

```
        move.l sum, d0

sum:    ds.l   1
```

M68000 reppresent this instruction as:



	opcode	source	Destination
instruction word.	move	direct mem	Reg d0

extension words

specify address of sum.

- The extension words (a total of 32 bits) gives the address of the memory location of the source operand.

- Direct addressing trick in SPARC : use of SETHI instruction

The SPARC's instruction length is 32 bits.

The SPARC's memory size is 2^{32} bytes.
To specify a memory address you will need 32 bits.

So:

SPARC instruction:

There is no room to specify the operation code (opcode), nor destination register if the *full* address is given in the instruction.

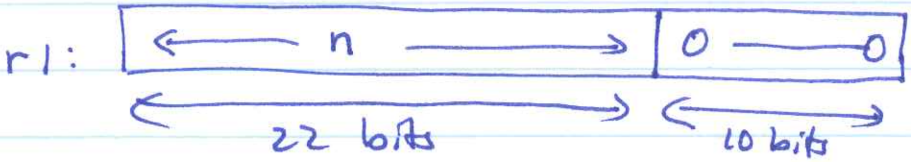
- In order to specify the address of any memory location, the SPARC breaks the address into 2 parts :

a "high" part
and a "low" part

and uses the SETHI (set high part) instruction to load a register with the high part of the address.

- Syntax: SETHI n, %r1

effect :



(1) put the value of n in the upper 22 bits of
(2) clear the lower 10 bits in registe r1.

- In addition, SPARC supplies 2 macros (routines) for computing the high & low part of a number:

$$\%hi(x) = x / 2^{10} \quad \text{(quotient)}$$
$$\%lo(x) = x \% 2^{10} \quad \text{(remainder).}$$

example:

label sum = 1111.0000.1010.1010.
$0101.01\underbrace{01.1111.1111}$
(32 bits).

then:

%hi(x) = 1111.0000.1010.1010.
0101.01

%lo(x) = 01.1111.1111

How to use SETHI in the direct addressing mode ~~to access memory locations?~~ ?

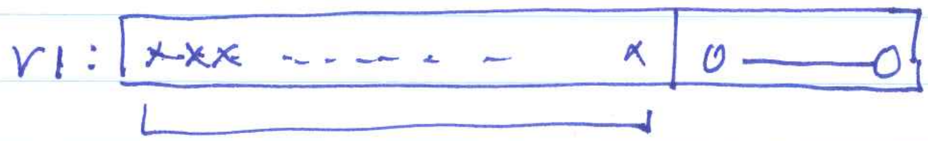
- This much you know:

$$\text{LD} \quad [\%r1 + \left\{\begin{matrix}\text{constant}\\ \%r2\end{matrix}\right\}], \%r3$$

will move the word at memory address given by $r1 + \left\{\begin{matrix}\text{constant}\\ r2\end{matrix}\right\}$ to register r3

SETHI %hi(sum) , %r1

will do the following:



left most 22 bits ~~of~~ of value of symbolic name "sum"

- Q : how do you move the word at label sum to a register ?

- Sol: SETHI %hi(sum), %l0
  LD [%l0 + %lo(sum)], %l1

- Similarly: how to store a word in register i7 to memory location at label max?

```
SETHI  %hi(max), %g1
ST   %i7 , [ %g1 + %lo(max)]
```

Moving data between IU & memory

Indirect (with offset)

- SPARC has only 2 instructions that move data between CPU & memory:

  LD - load, moves data from memory to CPU.
  ST - store, moves data from CPU to memory.

- LOAD :

  syntax: ① LD [%R1 + %R2], %R3

  R1, R2, R3 } any of global, local, input or output registers.

  moves the word at memory location R1 + R2 into register R3

  ② LD [%R1 + const], %R2

  moves the word at memory location R1 + const into register R2.

  $2^{12}-1 \leq \text{Const} \leq -2^{12}$.

- STORE :

  syntax : ① ST %R3, [%R1 + %R2]

  ② ST %R2, [%R1 + const]

Memory addressing instructions

- Only load & store instructions access memory.

- Syntax:

13 bit signed 2's compl. constant.

$$\mathrm{LD}\quad [\%r1 + \begin{Bmatrix} \text{constant} \\ \%r2 \end{Bmatrix}],\ \%r3$$

13 bit signed constant

$$\mathrm{ST}\quad \%r3,\ [\%r1 + \begin{Bmatrix} \text{constant} \\ \%r2 \end{Bmatrix}]$$

Effect: the effective addr. is computed as

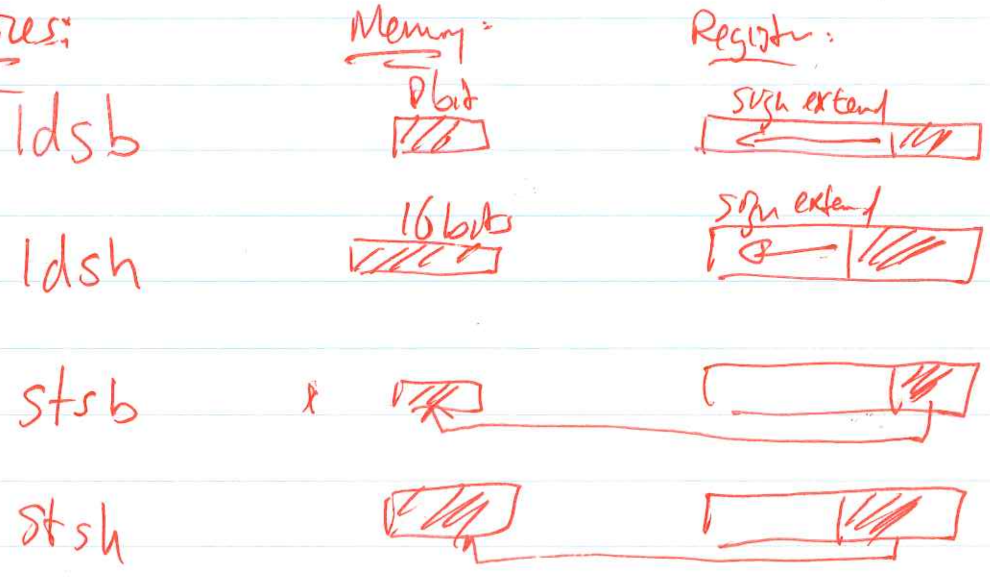
$$\%r1 + \text{constant}$$

or $\%r1 + \%r2$

and the ~~content~~ long word of this location is moved to register r3.

(we only discuss long word operations).

Operand sizes:



(Ldd - double word)

(std - double word)

Demo: sparc1.s

new-sparc1.s

```
int a, b, c;

main()
 { printf("Enter a= "); scanf("%d", &a);
   printf("Enter b= "); scanf("%d", &b);

   c = a+b;

   printf("Sum c = %d \n", c);
 }.
```

- First some explanation:

printf ("Enter a= ") what is it?

In C: a string "Enter a =" evaluate to the <u>address</u> of a string.

eg: str1: .ascii ~~.string~~ "Enter a = \0"

Printf gets the address (symb. const) str1 as input value !!!

.section.

~~.seg~~ "text" ———— start instructions.

.global _main ———— like xdef

_main: save %sp, 128, %sp — save space on stack & ~~[illegible]~~ decrement ~~po~~ CWP by 1.

set str1, %o0 - pass addr. of str1 (1^{e} parameter) in O0

call _printf

⋮

str1: .~~string~~ ascii "Enter a = \0".

```
! File: sparc1.s
! SPARC assembler program for the following equivalent C program:
! -------------------------------------------------------------
! int a, b, c;
!
! main()
! {
!    printf("Enter a = "); scanf("%d", &a);
!    printf("Enter b = "); scanf("%d", &b);
!    c = a + b;
!    printf("Sum c = %d\n", c);
! }
! ==============================================================
        .section ".text"                ! Begin main()
        .align 4                        ! Align to word boundary
        .global main                    ! M68000 xdef equivalent
main:
        save    %sp,-96,%sp             ! Pull down window and save space
                                        ! on stack - ignore this instruction
                                        ! for now

! ------------------------------------ Call printf with address of string Str1
! ------------------------------------ C's library func's: params in o0, o1,..
        sethi   %hi(Str1),%o0           ! Pass address of string Str1 in o0
        or      %o0,%lo(Str1),%o0       !
        call    printf                  ! Call printf
        nop                             ! Delay slot - fill with No Op for now

! ------------------------------------ Call scanf with addresses of InStr and a
        sethi   %hi(InStr),%o0          ! Pass address of string InStr in o0
        or      %o0,%lo(InStr),%o0      !
        sethi   %hi(a),%o1              ! Pass address of a in o1
        or      %o1,%lo(a),%o1          !
        call    scanf                   ! Call scanf
        nop                             ! Delay slot - fill with No Op for now

! ------------------------------------ Call printf with address of string Str2
        sethi   %hi(Str2),%o0           ! Pass address of string Str2 in o0
        or      %o0,%lo(Str2),%o0       !
        call    printf                  ! Call printf
        nop                             ! Delay slot - fill with No Op for now

! ------------------------------------ Call scanf with addresses of InStr and b
        sethi   %hi(InStr),%o0          ! Pass address of string InStr in o0
        or      %o0,%lo(InStr),%o0      !
        sethi   %hi(b),%o1              ! Pass address of b in o1
        or      %o1,%lo(b),%o1          !
        call    scanf                   ! Call scanf
        nop                             ! Delay slot - fill with No Op for now

! ------------------------------------ c = a + b
        sethi   %hi(a),%l0              ! Get a in l0
        ld      [%l0+%lo(a)],%l0        !
        sethi   %hi(b),%l1              ! Get a in l1
        ld      [%l1+%lo(b)],%l1        !
        add     %l0,%l1,%l2             ! Add a and b in l2
        sethi   %hi(c),%l3              ! Put l2 in c
        st      %l2,[%l3+%lo(c)]        !

! ------------------------------------ Call printf with address of Str3 and c
        sethi   %hi(Str3),%o0           ! Pass address of string Str2 in o0
        or      %o0,%lo(Str3),%o0       !
        sethi   %hi(c),%o1              ! Pass address of string c in o1
        ld      [%o1+%lo(c)],%o1        !
        call    printf                  ! Call printf
        nop                             ! Delay slot - fill with No Op for now
```

Compare & branching

- SPARC does not have a compare instruction.

  To do compare, SPARC subtract 2 values with subcc and store result in %g0.

- SPARC assembler recognizes a cmp mnemonic:

$$\text{cmp } \%r1, \begin{Bmatrix} \text{constant} \\ \%r2 \end{Bmatrix}$$

which is translated to:

$$\text{subcc } \%r1, \begin{Bmatrix} \text{constant} \\ \%r2 \end{Bmatrix}, \%g0 .$$

This will compute $\%r1 - \begin{Bmatrix} \text{const} \\ \%r2 \end{Bmatrix}$ & set the flags.

Notice Difference:

M68000 : cmp <ea>, Dn

Performs: Dn − <ea>

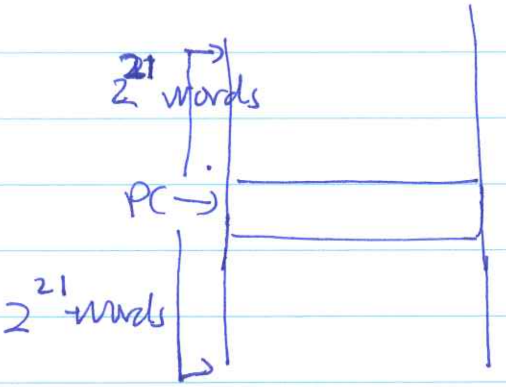
SPARC : cmp %r1, <ea>

performs: r1 − <ea>

reversed!!!

- Branching

Relative to PC
max. offset :



max offset : 2^{21} words $= 2^{32}$ bytes.

↑

22 bits offset

- Using branch with compare

The construct :

```
cmp %r1, %r2
bge label
```

branch if $r1 \geq r2$
(Note: quite the reverse to M68000 ! ☹)

will branch to label if $r1 \geq r2$.
(Note that M68000 cmp d0, d1 subtracts : d1 − d0 & set flags. Therefore, SPARC & M68000 are exactly opposite).

it's unfortunately the reverse of M68000 !!!

- Branch mnemonics.

ba label - branch always to label

bn - branch never

signed

only this

bne - branch not equal

be - branch equal

bg - branch greater than

bge - branch greater than or equal

bl - branch less than

ble - branch less than or equal

unsigned

bgu - branch greater unsigned

bleu - branch less than or equal unsigned

bcc - branch carry clear
or branch greater or equal unsigned

bcs - branch carry set
or branch less than unsigned.

bpos - branch positive (N = 0)

bneg - branch negative (N = 1)

bvc - branch overflow clear (V = 0)

bvs - branch over flow set (V = 1)

- Example: write a program that sums an array.

```
int a[] = {5, 2, 6, 7, 1}
int  i, sum;

main()
 {
     sum = 0;

     for (i=0;  i<5 ; i++ )
          sum  += a[i];

     printf ("sum = %d\n", sum);
 }
```

~~sparc2.s~~

sparc-example2b.s

use Egtapi to illustrate

Delayed Branching

- The SPARC CPU uses a "pipeline" to execute instructions (see CS355).

  This causes the branch (or any instructions that change PC) to have "one slot delay":

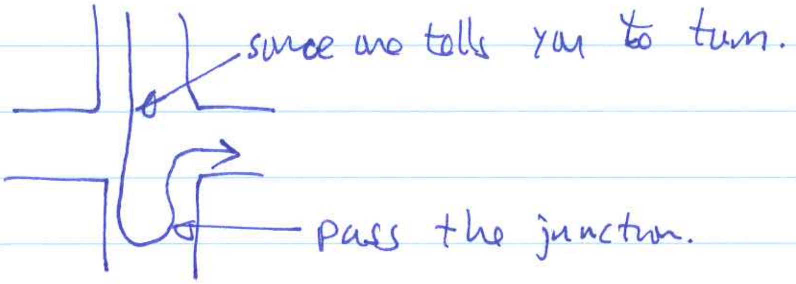
  The branch instruction does not take effect immediately, but after one instruction.

  In other words:

  After "executing" the branch (call, rts) instruction, the SPARC CPU will fetch & execute ONE more instruction after the branch (call, rts) instruction, and THEN jump to the location indicated by the branch (call, rts) instruction.

- This phenomenon is called "delayed branching".

- Analogy:



- Example

```
        mov   %g0 ,%l1
        add   %l1 , 1 , %l1
        ba    label
        add   %l1, 2, %l1
        add   %l1, 4, %l1
label:  add   %l1 , 0, %l1 .
```

delay-branching.s

- The following instructions will be executed:

```
mov   %g0 , %l1
add   %l1, 1, %l1
ba    label
add   %l1, 2, %l1
add   %l1, 0, %l1.
```

- The ~~instruction~~ location that follows the branching instruction is called "delay slot".

The instruction in the delay slot is always executed (unless annulled - later).

The instruction "add %l1, 2, %l1" is in the delay slot.

- Programming with branch delay

The trivial way to overcome the "strange" delayed branch behaviour is to put a NOP instruction after the branch:

```
      ⋮
branch instruction
NOP
      ⋮
```

- A more sophristicated way is to "move" a useful instruction into the delay slot.

- Example: same program as sparc2.s, now with filled delay slots.

```
int a[] = {5, 2, 6, 7, 1}
int i, sum;

main()
  {  sum = 0;

       for (i=0; i<5; i++)
           sum = sum + a[i];

       printf("Sum = %d \n", sum);
  }
```

sparc2a.s

```
! File: sparc2.s
! ---------------------------------------------------
! int a[] = {5, 2, 6, 7, 1};
! int i, sum;
!
! main()
! {
!    sum = 0;
!    for (i = 0; i < 5; i++)
!       sum += a[i];
!    printf("Sum = %d\n", sum);
! }
! ====================================================
        .section ".text"
        .global main
main:
        save    %sp,-96,%sp

        sethi   %hi(sum),%o0
        st      %g0,[%o0+%lo(sum)]      ! sum = 0

        sethi   %hi(i),%o0
        st      %g0,[%o0+%lo(i)]        ! i = 0

for:
        sethi   %hi(i),%o0
        ld      [%o0+%lo(i)],%o0        ! Get i
        cmp     %o0,5
        bge     fordone
        nop                             ! Delay slot !

        sethi   %hi(a), %o1             ! Get address of array a
        or      %o1, %lo(a), %o1
        smul    %o0,4,%o0               ! 4*i
        ld      [%o1+%o0],%o2           ! Get a[i]

        sethi   %hi(sum),%o3            ! Get sum in o6
        ld      [%o3+%lo(sum)],%o3

        add     %o3,%o2,%o3             ! Add a[i] to sum

        sethi   %hi(sum),%o0            ! Put sum back in memory
        st      %o3,[%o0+%lo(sum)]

        sethi   %hi(i),%o0
        ld      [%o0+%lo(i)],%o1        ! Get i
        add     %o1,1,%o1               ! Add 1 to i
        st      %o1,[%o0+%lo(i)]        ! Put i back in memory

        ba      for                     ! Go to for test
        nop                             ! Delay slot....

fordone:
                                        ! Print sum with printf
        set     Str,%o0                 ! Pass first parameter o0
        sethi   %hi(Str), %o0           ! Pass address of Str in o0
        or      %o0, %lo(Str), %o0
        sethi   %hi(sum),%o1            ! Pass sum in o1
        ld      [%o1+%lo(sum)],%o1
        call    printf                  ! Call printf
        nop                             ! Delay slot

                                        ! main done - return to OS
        ret
        restore
! ---------------------------------------------------------------
! ------------------------------------- main done - return to OS
        ret                             ! return
        restore                         ! delay slot - push back window


        .section ".data1"
Str1:   .ascii  "Enter a = \0"
Str2:   .ascii  "Enter b = \0"
Str3:   .ascii  "Sum c = %d\n\0"
InStr:  .ascii  "%d\0"

        .section ".bss"
        .align 4
a:      .skip   4
b:      .skip   4
c:      .skip   4
```