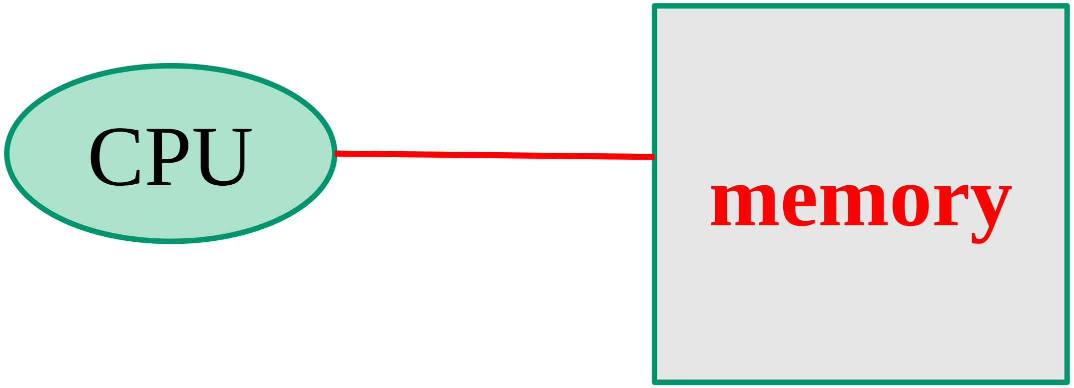
Memory

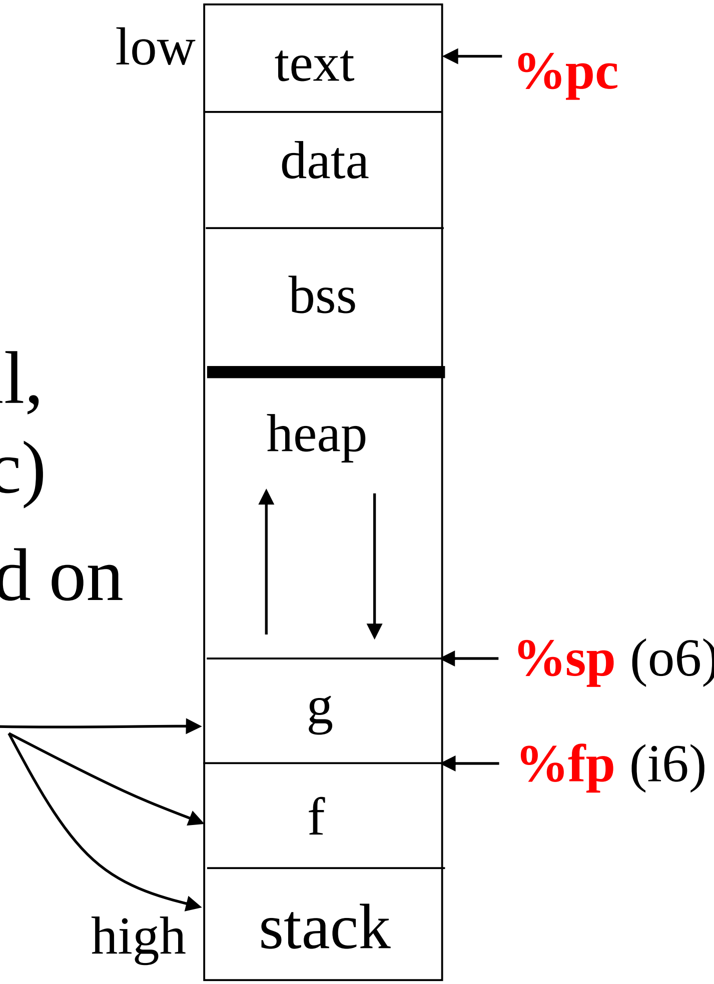


- Storage class
  - ✓ auto
  - ✓ static
  - ✓ extern
  - ✓ register

- Memory section
  - ✓ text section
  - ✓ stack section
  - ✓ data section

Memory space of SPARC executables

- static segments (compile-time)
  - ✓ **text** : program code
  - ✓ **data, bss** : global variables
- dynamic segments (run-time)
  - ✓ heap: allocated via function call, OS/library support (e. g., malloc)
  - ✓ **stack**: a stack frame is allocated on invocation of a function
    - main → call f →call g
    - local/automatic variable
    - needed for register saving



(32-bit) SPARC Memory

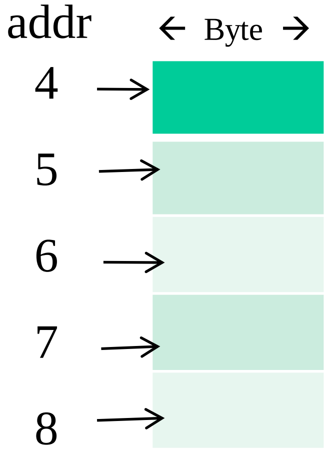
- Program and data is stored in memory
- Memory size: 2^{32} bytes for each process
  - ✓ **Addressing unit**: byte

- Data types

C	SPARC	size	unsigned	signed
char	byte	8	$0:2^8-1$	$-2^7:2^7-1$
short	half	16	$0:2^{16}-1$	$-2^{15}:2^{15}-1$
int, long	word	32	$0:2^{32}-1$	$-2^{31}:2^{31}-1$

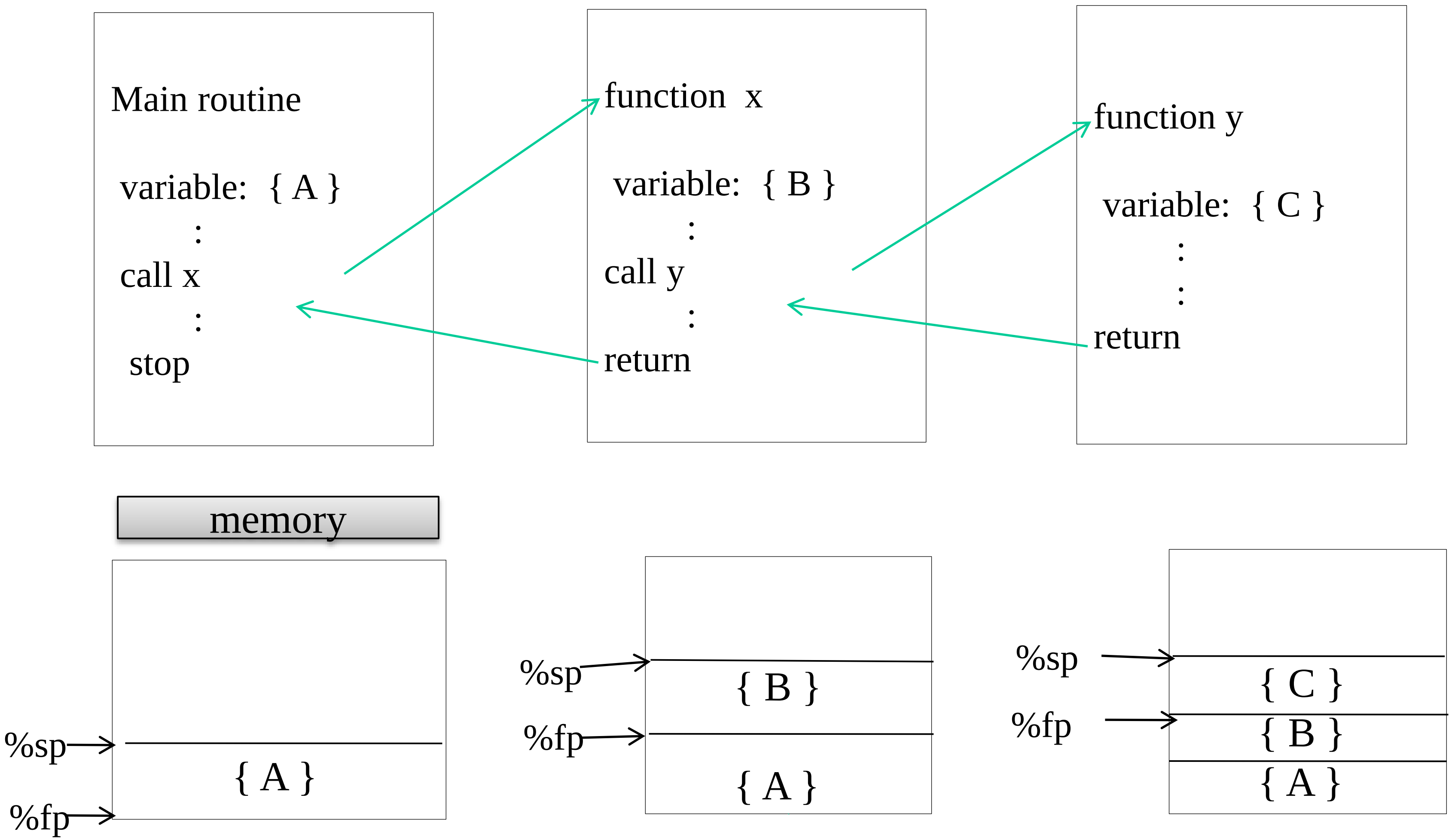
Alignment

- Data must be aligned on their natural boundaries. The starting address of a specific type of data must be a multiple of its size in bytes



  - ✓ char (1B): 0, 1, 2, 3, ...
  - ✓ short (2B): 0, 2, 4, 6, ...
  - ✓ int, long (4B): 0, 4, 8, ...
  - ✓ double (8B): 0, 8, 16, 24, 32, ...

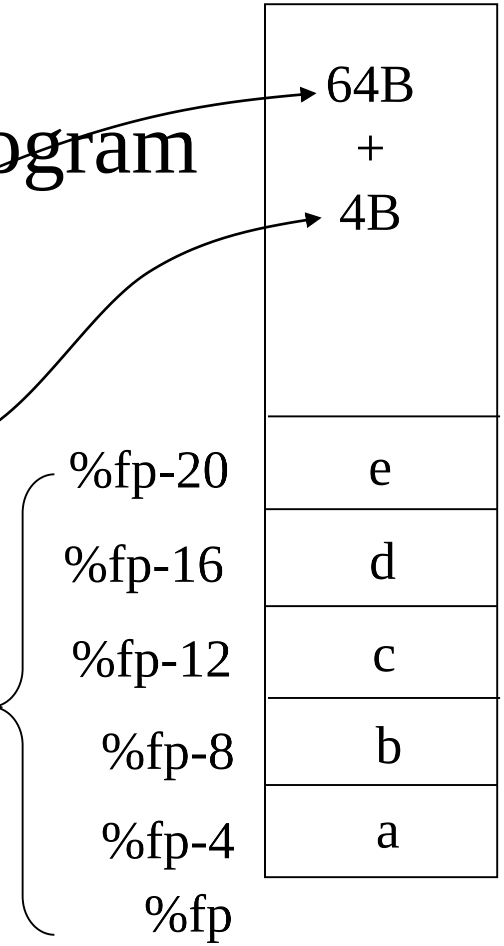
Local variables' lifetime



Stack frame

- **Minimum size: 64B/92B -** this is for system (storing i and l registers on interupt, trap, etc.), not for user program
- **Size: a multiple of 8**
- When 20B is needed for vars
  - save %sp, (-64 - (5*4)) & -8, %sp

```
int f(int x, int y) {
   int a, b, c, d, e;
   ...
}
```



SAVE instruction

- Effect of executing save instruction
  - ✓ stack frame allocation
  - ✓ new register set allocation

save %sp, (-64 - (5*4)) & -8, %sp

or save %sp, (-64 - (5*4)) & 0xfffffff8, %sp

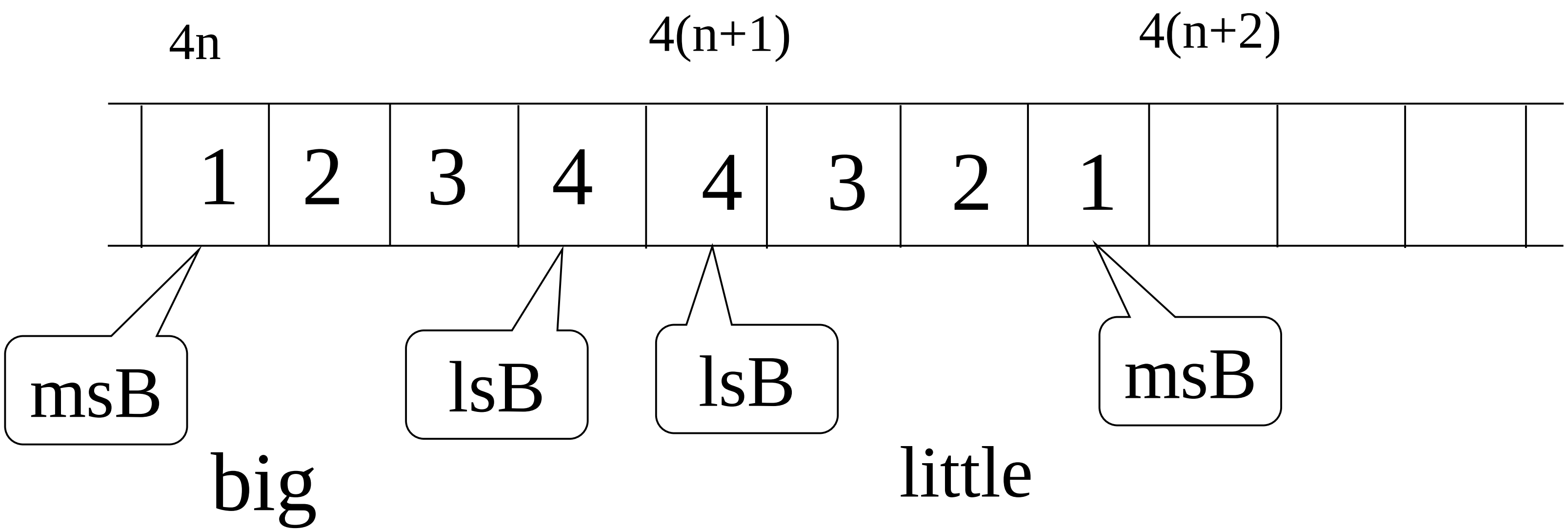
or save %sp, -88, %sp

result: $(\%fp)_{new} \leftarrow (\%sp)_{old}$

$(\%sp)_{new} \leftarrow (\%sp)_{old} - 88$

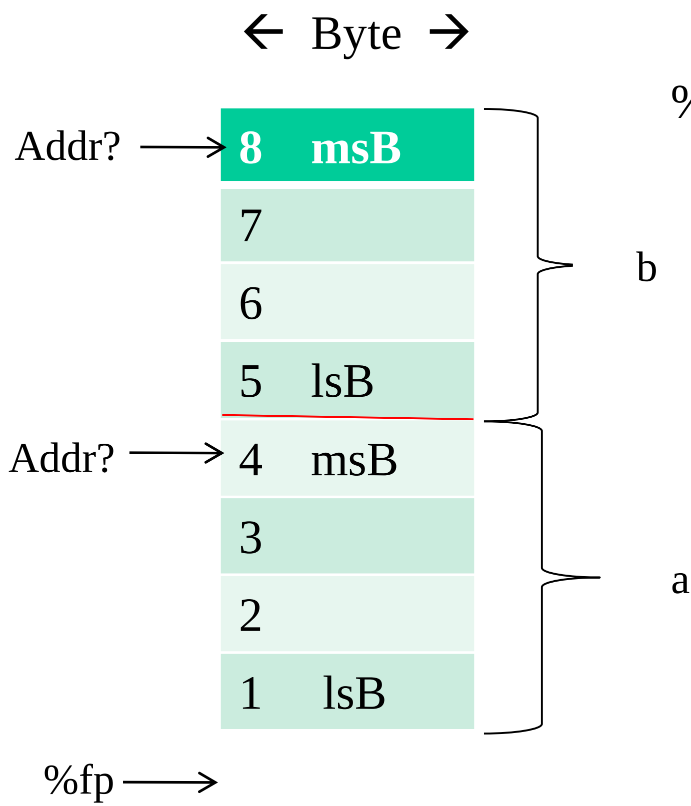
Data movement and endianness

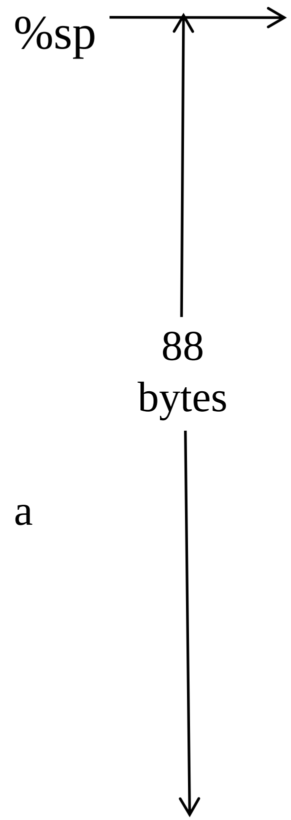
- Data movement available for 1B, 2B, 4B, 8B
- Moving 8B requires a register pair ($R_{2n}R_{2n+1}$)
- Big endian: address of a data block over multiple bytes is its msB's addr. addr(msB) < addr(lsB)



Deciding addresses of variables in stack

✓ int a, b, c, d, e



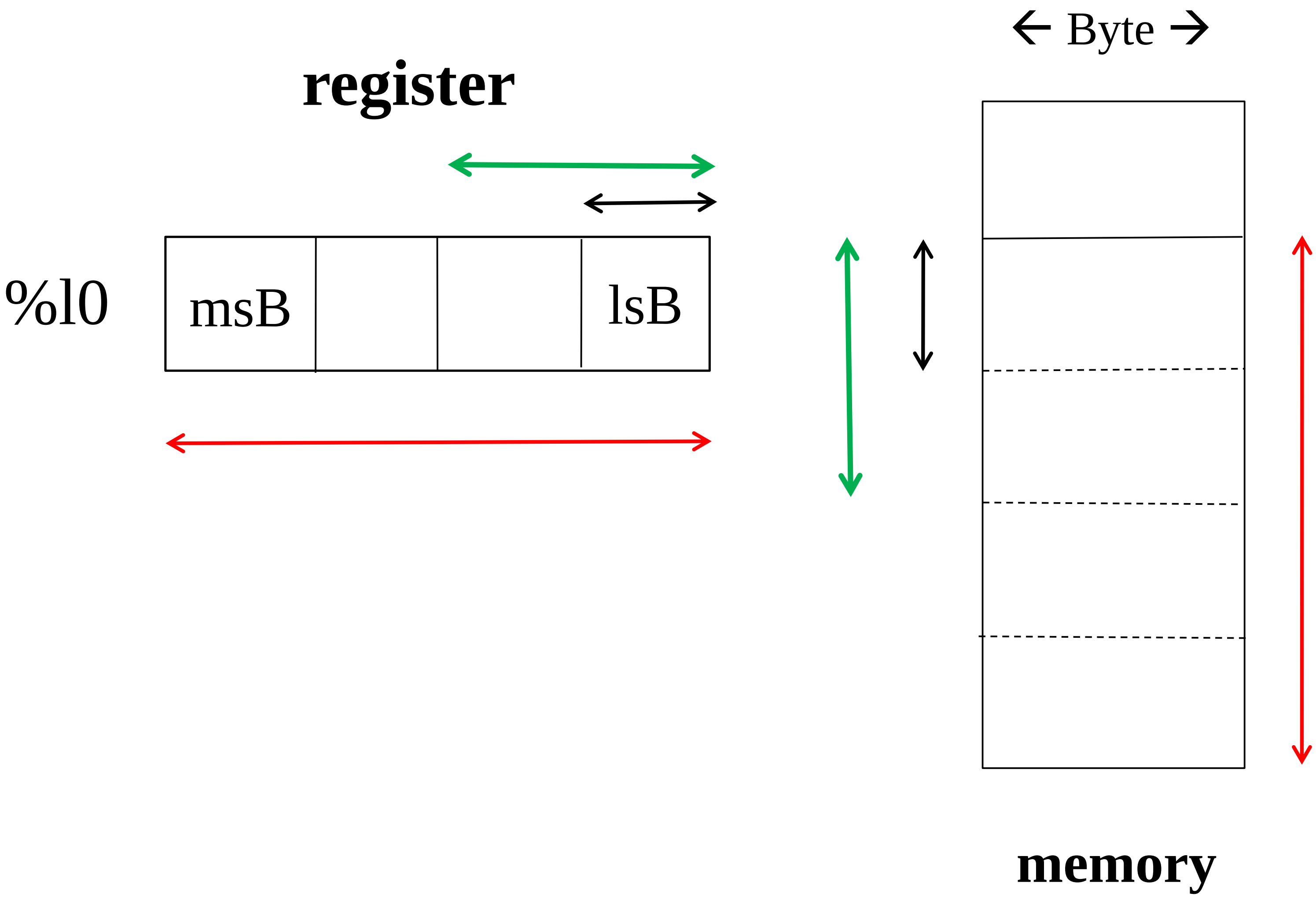


address		name
	64 bytes of Storage for system use (to save registers)	
%fp-20	20 19 18 17	e
%fp-16	16 15 14 13	d
%fp-12	12 11 10 9	c
%fp-8	8 7 6 5	b
%fp-4	4 3 2 1	a

Memory access instructions

load (register ← memory), **store** (memory ← register)

Opcode	operation
ldsb	signed byte, propagation of sign
ldub	unsigned byte, 0 padding
ldsh	signed half word, propagation of sign
lduh	unsigned half word, 0 padding
ld	word
ldd	double word
stb	low byte, no sign extension, no 0 padding
sth	low 2 bytes, no sign extension, no 0 padding
st	word, no sign extension, no 0 padding
std	double word, no sign extension, no 0 padding



← Byte →
register
%l0
msB
lsB
memory

Sign Extension

- 16 bit vs 32 bit representation

$\mathbf{0}000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

$0000\ 0000\ 0000\ 0010_{two}$

1's complement

$1111\ 1111\ 1111\ 1101_{two}$

$+ \quad 1_{two}$

$\mathbf{1}111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

$= 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

Memory access instruction format

- op-1 [R+A], S ! x = *p;
- op-2 S, [R+A] ! *p = x;
  - ✓ [...]: pointer dereferencing. *p in C language

    Memory address
  - ✓ R: register (containing memory address)
  - ✓ A: register or immediate (-4096~ 4095)
  - ✓ S: register (destination, source)

❖ op-1: load op-code

❖ op-2: store op-code

Example

- Local variables

```
int  a,b,c
  a = 5;
  b = 7;
  c = a + b;
```

- address of a, b, c

  a: %fp-4

  b: %fp-8

  c: %fp-12

- Assembly program

```
save  %sp, -80, %sp
mov  5, %l0
st     %l0, [%fp-4]

mov  7, %l0
st      %l0, [%fp-8]

ld     [%fp-4], %l0
ld     [%fp-8], %l1
add  %l0, %l1, %l0
st      %l0, [%fp-12]
```

Memory access instruction format

- ✓ op-1 [R+ A], S
- ✓ op-2 S, [R+A]

Bit index	31 30	25	24 19	18 14	13	12 5	4 0
Field	OP	S	OP-확장	R	**0**		A

Bit index	31 30	29 25	24 19	18 14	13	12 0
Field	OP	S	OP-확장	R	**1**	상 수

- ✓ OP : 11 OP-확장: 표 5.2 참조

- ✓ Example

```
  ld  [%o0-20],  %l0
→ 11 10000 000000 01000 1 1111111101100
  st  %l0, [%fp-24]
→ 11 10000 000100 11110 1 1111111101000
```

Calculating offset of variables in stack

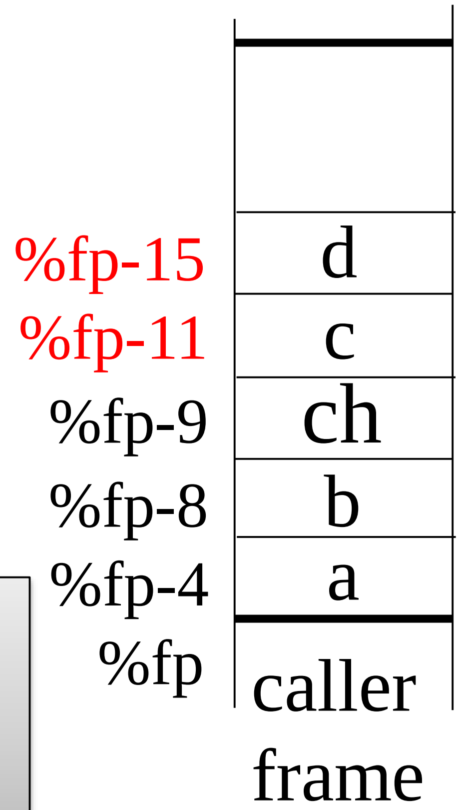
- Address boundary alignment is needed
- Example

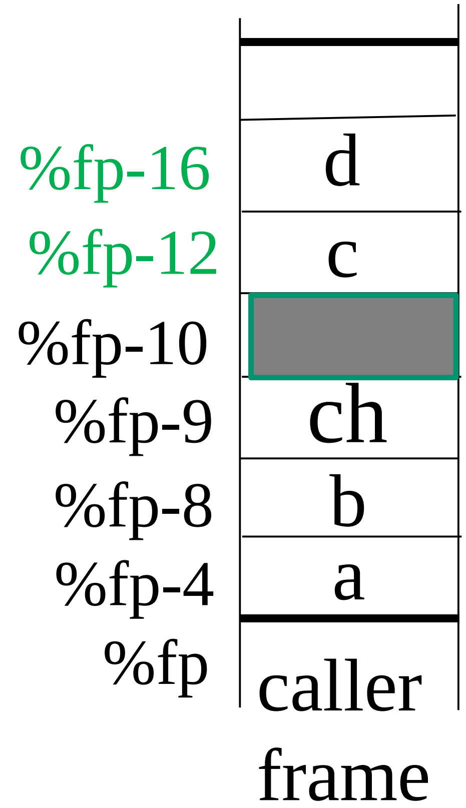
  int a, b;

  char ch;

  short c;

  int d





Any problem accessing c and d?

lduh [%fp-11], %l0

Example

```
int a, b;
char c1;
int c, d;
```

Offsets of local/stack variables
a: -4, b: -8, c1: -9, c: -16, d: -20

```
register int x, y , z;
```

Register variables

```
x = 17;
y = -5;

for(z = 1; z < x + y; z++)
  for(a = z; a >= z * y; a -= 10) {
    d = a + z;
    c1 = d * b;
    c = a + y / z;
  }
```

★ Storage types for variables

1. register

Register name		**Var name**
%l0		x
%l1		y
%l2		z

<u>Frame size</u>
A multiple of 8 greater than (20B + 64B)

2. memory

Addr	**Memory (stack)**	**Var name**
%sp →		
%fp-20	20 19 18 17	d
%fp-16	16 15 14 13	c
%fp-9	~~12 11 10~~ 9	c1
%fp-8	8 7 6 5	b
%fp-4	4 3 2 1	a
%fp →		

Assembly code

```
        a_s = -4
        b_s = -8
        c1_s = -9
        c_s = -16
        d_s = -20

        !  x :  %l0
        !  y :  %l1
        !  z :  %l2

        .global main
main:   save    %sp, -88, %sp  ! [ -64 + (-20)] & -8

        mov     17, %l0         !x = 17
        mov     -5, %l1         !y = -5
        ba      outer_test
        mov      1,  %l2        ! z = 1
```

- Memory inspection using gdb

✓ stack: x/d $fp-4
✓ data section: x/d &x

```
inner:                                  !code for inner loop
        add     %o0, %l2, %o0   !  a = a + z
        st      %o0, [%fp + d_s] ! d= a, d=a+z

        ld      [%fp + d_s], %o0
        call    .mul                    !  d * b
        ld      [%fp + b_s], %o1
        stb     %o0, [%fp + c1_s]    ! Store in c1

        mov     %l1, %o0
        call    .div            !  y/z
        mov     %l2, %o1
        ld      [%fp + a_s], %o1
        add     %o0, %o1, %o0
        st      %o0, [%fp + c_s]

inner_inc:                              !inner for increment statement
        ld      [%fp + a_s], %o0
        sub     %o0, 10, %o0        !  a-10
        st      %o0, [%fp + a_s]    ! a = a - 10
```

```
a: -4
b: -8
c1: -9
c: -16
d: -20
```

```
for(z = 1; z < x + y; z++)
  for(a = z; a >= z * y; a -= 10) {
    d = a + z;
    c1 = d * b;
    c = a + y / z;
  }
```

```
inner_test:                              !inner for test
        mov     %l2, %o0
        call    .mul        !  z*y
        mov     %l1, %o1
        ld      [%fp + a_s], %o1
        cmp     %o1, %o0     !   a  ?  z*y
        bge,a   inner
        ld      [%fp + a_s], %o0

outer_inc:                               !outer for increment statement
        add     %l2, 1, %l2

outer_test:                              !outer for test
        add     %l0, %l1, %o0
        cmp     %l2, %o0  ! z ? x+y
        bl,a    inner_test
        st      %l2, [%fp + a_s] !a = z

        mov     1, %g1
        ta      0
```

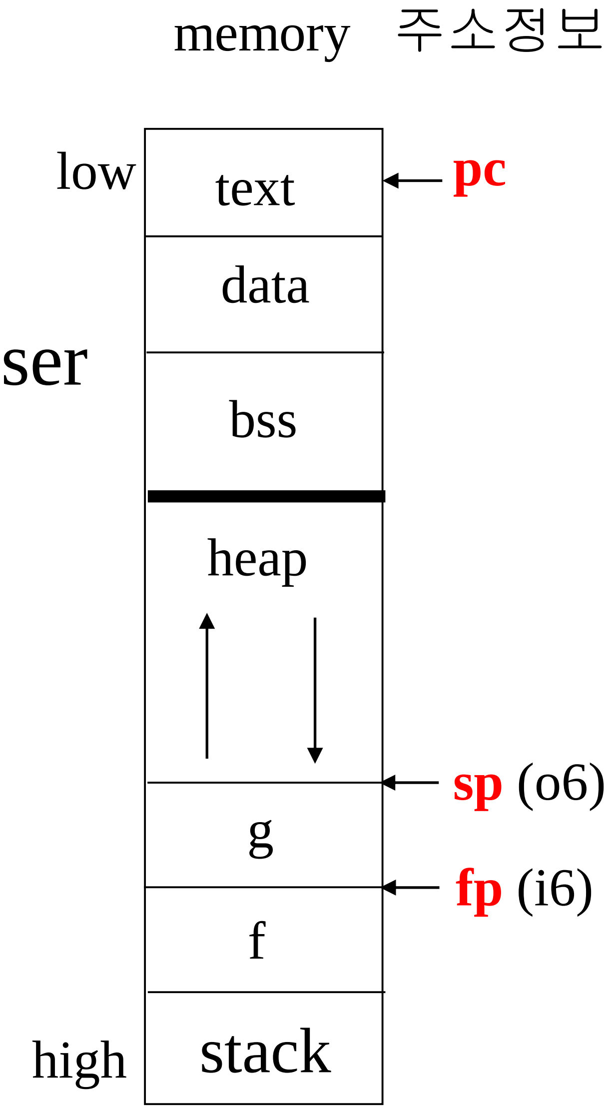
```
a: -4
b: -8
c1: -9
c: -16
d: -20
```

```
for(z = 1; z < x + y; z++)
  for(a = z; a >= z * y; a -= 10) {
    d = a + z;
    c1 = d * b;
    c = a + y / z;
  }
```

External Data and Text

- Memory is for program & data
- static area: allocated by compiler
  - ✓ text segment: code, read-only data
  - ✓ **data** segment: initialized vars by user
  - ✓ **bss** segment: zero-initialized vars

  – **extern, static** variable in C lang.



text segment (section)

- **.section “.text”**

- Stores program
- Stores read-only data
  - ✓ Ex: arguments passing

- Using .global main is exposed externally
  - ✓ .global main

Data Segment

- **.section “.data”**
- Pseudo-op for data initialization

  .word **.skip**

  .half **.align**

  .byte **.ascii**

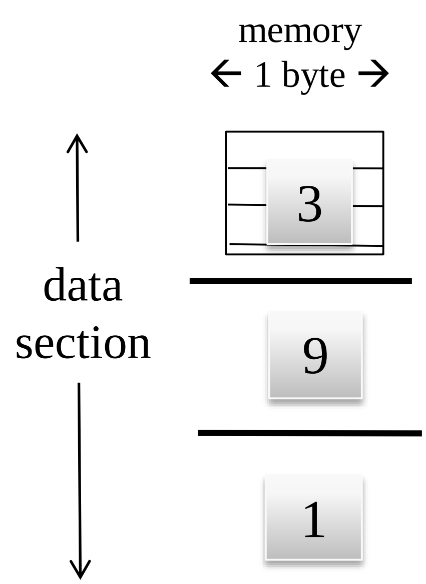
  .asciz

- Exanoke

  .section “.data”

  .word 3, 3*3, 3*3 >> 3

  → Stores each of 3, 9, 1 in 4 B area

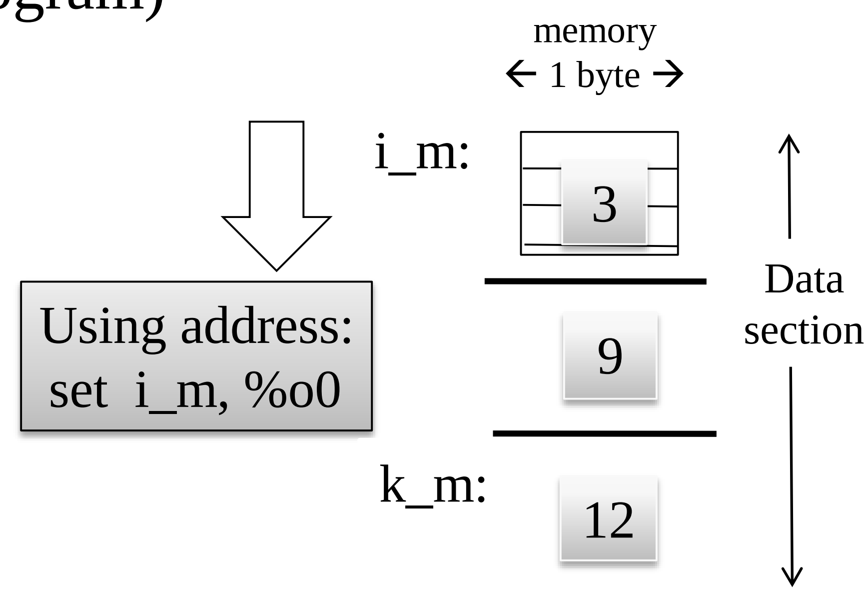


Address Reference

- Labels (for easy use in program)

  Example –

```
        .section ".data"
i_m:    .word   3
j_m:    .word   9
k_m:    .word   3 + 9
```



- In C lang:

```c
static int i = 3;
static int j = 9;
static int k;
k = i + j;
```

32 bit constant and sethi instruction

•In arithmetic/logical, memory access instructions immediate value field size is 13 bits → how to use value greater than 13-bit number?

• sethi format

op	31:30 = 00
rd	29:25
op2	24:22 = 100
imm	**21:0**

• Usage

✓ `sethi 0x30cf0034 >> 10, %l0`

→ 00 10000 100 0011000011001111000000

• Results

(Rd) ← upper 22 bits are set to imm, lower 10 bits are set to 0

%l0: 0011000011001111000000 0000000000

Storing 32 bit numbers

- sethi 0x30cf0034 >> 10, %o0

  or   %o0, 0x30cf0034 & 0x3ff, %o0

- sethi %hi(0x30cf0034), %o0

  or   %o0, %lo(0x30cf0034), %o0

- set 0x30cf0034, %o0

      results: %o0 ← 0x30cf0034

- %hi, %lo operator
  - ✓ %hi(x): x >> 10 (right shift) ! high 22 bits
  - ✓ %lo(x): x & 0x3ff ! low 10 bits
- set usage

```
main:  save  %sp, -96, %sp
          :
       mov  1,  %o0
       mov  2,  %o1
       call  foo
       nop
          :
```

```
set  foo,  %l0
jmpl  %l0, %o7
nop
```

Data Segment

Data labels

Start of data segment

Export labels

4B storage

Start of text segment

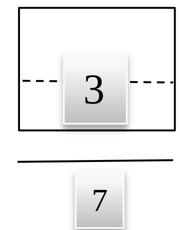
```
        .section    ".data"
        .global     i_m, j_m, k_m
i_m:    .word       3
j_m:    .word       9
k_m:    .word       0
        .section    ".text"
        .global     main
main:   save        %sp, -96, %sp
        sethi       %hi(i_m), %o0
        ld          [%o0 + %lo(i_m)], %l0
        sethi       %hi(j_m), %o0
        ld          [%o0 + %lo(j_m)], %l1
        add         %l0, %l1, %o0
        set         k_m, %o1
        st          %o0, [%o1]
```

```
set  i_m, %o0
ld  [%o0], %l0
set  j_m, %o0
ld  [%o0], %l1
```

Varying sizes when memory allocation

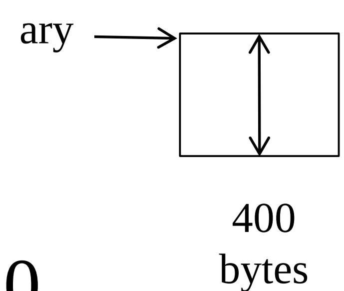
- byte, half-word

```
.half    3
.byte    7
```



- Allocation without initialization

```
ary:      .skip    4*100      ! int  ary[100]
```



Using start address: set ary, %l0

Alignment

- Without alignment

Relative addr.	.section ".data"	
0	a: .word	3
4	b: .byte	5
5	c: .half	5
7	d: .byte	6
8	e: .word	17

- With alignment

Relative addr.		
0	a: .word	3
4	b: .byte	5
	.align	**2**
6	c: .half	5
8	d: .byte	6
	.align	**4**
12	e: .word	17

ASCII data

- .byte 0150, 0145, 0154, 0154, 0157
- .byte "h", "e", "l", "l", "o"

- .ascii "hello" ! Same as upper two cases
- .byte 0 ! null char. (\0: end of string)

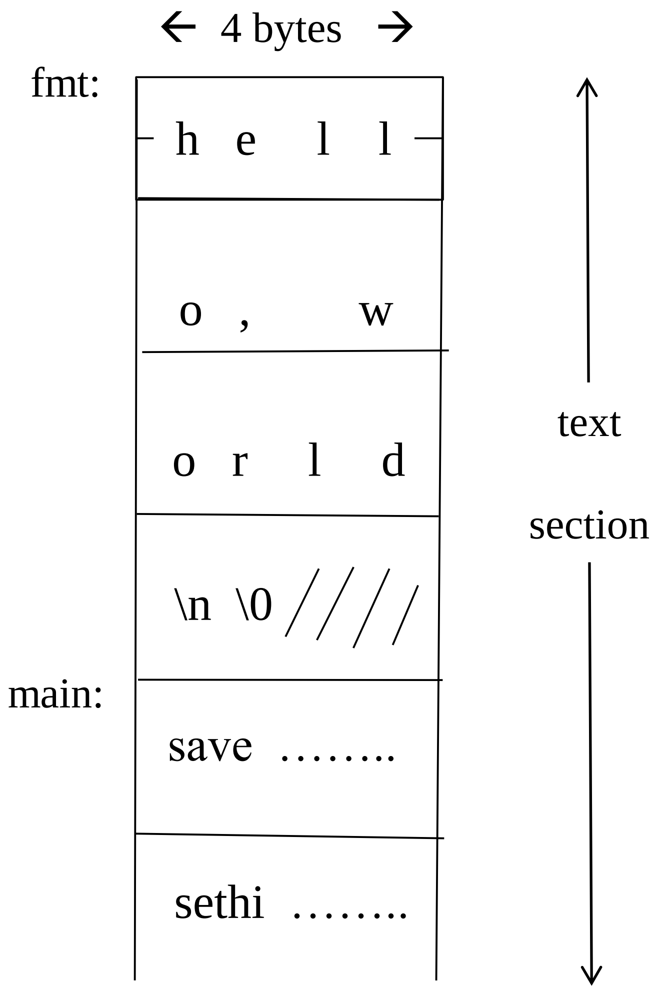
- .asciz "hello" ! Add null char at the end

Format string

```
        .section  ".text"
        .global   printf
fmt:    .asciz    "hello, world\n"
        .align    4   !  why?
        .global   main
main:   save      %sp, -96, %sp
        sethi     %hi(fmt), %o0
        call      printf
        or        %o0, %lo(fmt), %o0
        ret
        restore
```

```
main()
{
 printf("hello, world \n");
}
```

```
set  fmt, %o0
call  printf
nop
```


4 bytes
fmt:
h e l l
o , w
o r l d
\n \0
main:
save ....
sethi ....
text
section

Pointer

- Pointer to local/stack variables

```
add    %fp, x_s, %o0   ! %o0 ← %fp  +  x_s
```

- Pointer to external variables/data

```
set    x_m, %o0
```

Or

```
sethi  %hi(x_m), %o0
or     %o0, %lo(x_m), %o0
```

bss section

- Initialized with 0
- Example

```
        .section    ".bss"
        .align 4
ary:    .skip       4*100
i_m:    .skip       4
```

1-dimensional array

- Stored in consecutive memory

  int a;

  char c1;

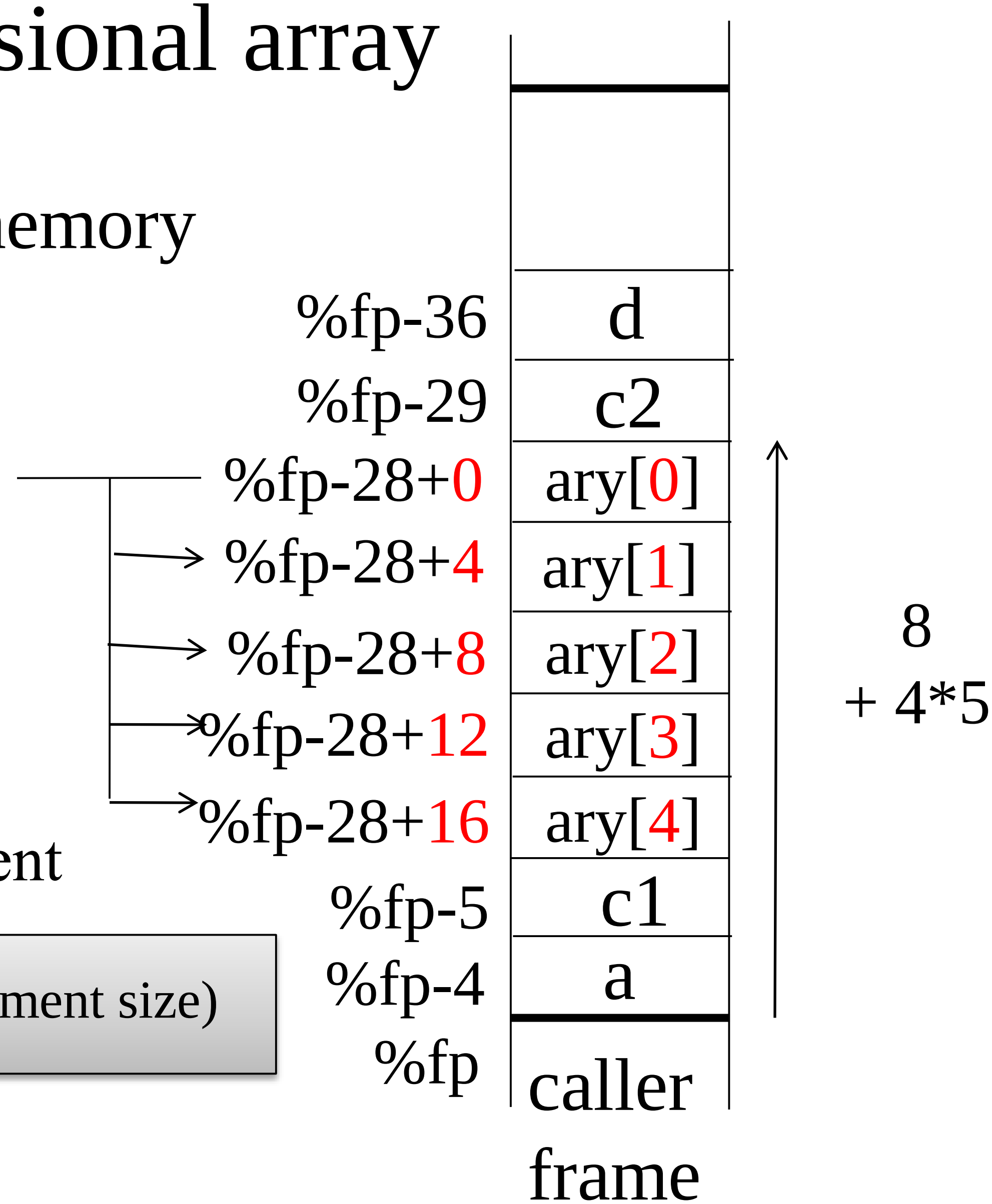
  int ary[5];

  char c2;

  int d;

- Finding address of i's element

Address of 1st element + i * (element size)



Address calculation

- Storing ary[i] in %o0

  (assumption: i is stored in %l0)

  a_s = -4

  c1_s = -5

  ary_s = -28

  c2_s = -29

  d _s = -36

```
sll    %l0,  2,  %o0       ! i * 4
add   %fp,  %o0,  %o0   ! add the frame pointer
ld     [%o0 + ary_s],   %o0
                 ↑
       %fp - 28 + i * 4
```

Finding min number in nums

```
void main(){

  int nums[20] = {17, 11, -161, -32, -893, 566, 25, 88, 67, -90};
  int n = 10;                            /* number of elements in array */
  int min;                               /* to hold the minimum element */
  register int i;                       /* for index */

  min = nums[0];                        /* initialize min to first element */

  for (i = 1; i < n; i++)               /* run through rest of array */

    if (nums[i] < min)                  /* storing smallest number */
      min = nums[i];
 }
```

Variable arrangement

address		variables
%sp →	register saving	
%fp-88 %fp-84		min n
%fp-80		nums[0]
%fp-76		nums[1]
%fp-72		nums[2]
%fp-4		nums[19]
%fp →		

17

11

-161

:

- Using register

  index i : %l0

```
!local variables

        nums_s = -80  !  4 * 20
        n = -84
        min = -88

!  index i  in  %l0

        .global main
main: save    %sp, -184, %sp   ! -92 -88 & -8

      mov     17, %o0                        ! initialization
      st      %o0, [%fp + nums_s + 0]
      mov     11, %o0
      st      %o0, [%fp + nums_s + 4]
      mov     -161, %o0
      st      %o0, [%fp + nums_s + 8]
      mov     -32, %o0
      st      %o0, [%fp + nums_s + 12]
      mov     -893, %o0
      st      %o0, [%fp + nums_s + 16]
```

```
        mov     566, %o0
        st      %o0, [%fp + nums_s + 20]
        mov     25, %o0
        st      %o0, [%fp + nums_s + 24]
        mov     88, %o0
        st      %o0, [%fp + nums_s + 28]
        mov     67, %o0
        st      %o0, [%fp + nums_s + 32]
        mov     -90, %o0
        st      %o0, [%fp + nums_s + 36]

        mov     10, %o0                          ! n = 10
        st      %o0, [%fp + n]                   ! Store in stack
        ld      [%fp + nums_s], %l1              ! min initialization
        st      %l1, [%fp + min]                 ! min = nums[0]
        ba       fortest
        mov     1, %l0                  ! Set loop index I to 1

for:    sll     %l0, 2, %o0             ! o0 = i * 4
        add     %fp, %o0, %o0           ! o0 = %fp + i * 4
```

```
        ld       [%o0 + nums_s], %o0      ! load  nums[i]
        ld       [%fp + min], %l1         ! load  min
        cmp      %o0, %l1                 ! nums[i]  ?  min
        bge      keep                     !     >=  then branch
        nop
        st     %o0, [%fp + min]           !     min update
keep:
         add     %l0, 1, %l0              ! i++
fortest:
         ld      [%fp + n], %o0           !  Loading n
         cmp     %l0, %o0                 !  i  ?  n
         bl      for                      !     <  then  branch
         nop
         mov     1, %g1                   !     >= then  exit
         ta      0
```

Performance tuning: using registers

If we change to

```
register int min;
register int n=10;
```

```
         nums_s = -80

        !  index i   in  %l0
        !  min       in  %l1
        !  n         in  %l2

       .global main
main:  save    %sp, -176, %sp  ! ( -92 - 80 ) & -8

      mov     17, %o0                       ! Initialization
```

```
st      %o0, [%fp + nums_s + 0]
mov     11, %o0
st      %o0, [%fp + nums_s + 4]
mov     -161, %o0
st      %o0, [%fp + nums_s + 8]
mov     -32, %o0
st      %o0, [%fp + nums_s + 12]
mov     -893, %o0
st      %o0, [%fp + nums_s + 16]
mov     566, %o0
st      %o0, [%fp + nums_s + 20]
mov     25, %o0
st      %o0, [%fp + nums_s + 24]
mov     88, %o0
st      %o0, [%fp + nums_s + 28]
mov     67, %o0
st      %o0, [%fp + nums_s + 32]
mov     -90, %o0
st      %o0, [%fp + nums_s + 36]
```

```
        mov     10, %l2                 ! n:   reg.  l2
        ld      [%fp + nums_s], %l1     ! min: reg.  l1
        ba       fortest
        mov     1, %l0
for:    sll     %l0, 2, %o0             ! o0 = i * 4
        add     %fp, %o0, %o0           ! o0 = %fp + i * 4
        ld      [%o0 + nums], %o0
        cmp     %o0, %l1                ! No need for accessing min
        bge     keep
        nop
        mov     %o0, %l1                !  min update
keep:
        add     %l0, 1, %l0             ! i++
fortest:
        cmp     %l0, %l2            !  Elimination of load for n
        bl      for
        nop

        mov     1, %g1
        ta       0
```

Optimizing address calculation

Address calculation using array index:
%fp + nums +   %l0 * 4

Use pointer:
%l3 = %fp + nums

%l3 = %l3 + 4

```
        mov     10, %l2             ! n:   reg.  l2
        add     %fp, nums_s, %l3    !  Pointer in %l3
        ld      [%l3], %l1          ! min: reg.  l1
        ba       fortest
        mov     1, %l0
For:
        add     %l3, 4, %l3         ! %l3 = %l3 + 4
        ld      [%l3], %o0
        cmp     %o0, %l1
        bge     keep
        nop
        mov     %o0, %l1            !  min update
keep:
        add     %l0, 1, %l0         ! i++
fortest:
        cmp     %l0, %l2
        bl      for
        nop
        mov     1, %g1
        ta       0
```

Filling-up delay slots

```
        mov     10, %l2             ! n:   reg.  l2
        add     %fp, nums_s, %l3    !  Pointer in %l3
        ld      [%l3], %l1          ! min: reg.  l1
        ba       fortest
        mov     1, %l0
For:
        ld      [%l3], %o0
        cmp     %o0, %l1
        bge     keep
        add     %l0, 1, %l0         ! i++
        mov     %o0, %l1            !  min update
keep:
fortest:
        cmp     %l0, %l2
        bl,a      for
        add     %l3, 4, %l3         ! %l3 = %l3 + 4
        mov     1, %g1
        ta       0
```

Altering loop control

```
min = nums[0];

for (i = 1; i < 10; i++)

  if (nums[i] < min)
    min = nums[i];
```

```
min = nums[0];

for( i = 9, i > 0; i-- )

  if( nums[i] < min )
    min = nums[i];
```

```
        ld       [%fp+nums_s], %l1    ! min: %l1
        mov      9, %l0               !  index = 9
        add     %fp, nums_s+ 4*9, %l3 !  Address of last element
        ba        fortest
        tst      %l0                  ! i > 0
For:
        ld       [%l3], %o0
        cmp      %o0, %l1
        bge      keep
        nop
        mov      %o0, %l1            !  min update
keep:
        subcc      %l0, 1, %l0       ! i--
        sub        %l3, 4, %l3       ! Pointer - 4
fortest:
        bg       for
        nop
        mov      1, %g1
        ta        0
```

```
        ld        [%fp+nums_s], %l1     ! min: %l1
        orcc      %g0, 9, %l0            !  index = 9 & i > 0
        add      %fp, nums_s+ 4*9, %l3
        ba          fortest
        nop
For:
        ld        [%l3], %o0
        cmp       %o0, %l1
        bge       keep
        nop
        mov       %o0, %l1              !  min update
keep:
        subcc       %l0, 1, %l0         ! i--
        sub         %l3, 4, %l3         ! Pointer - 4
fortest:
        bg        for
        nop
        mov       1, %g1
        ta          0
```

Replace with bg,a   for

Load delay slot

```
        ld      [%fp+nums_s], %l1
        orcc  %g0, 9, %l0
        add %fp, nums_s+ 4*9, %l3
        ba        fortest
        nop
For:
        ld        [%l3], %o0
        cmp       %o0, %l1
        bge       keep
        nop
        mov       %o0, %l1
keep:
        subcc       %l0, 1, %l0
        sub         %l3, 4, %l3
fortest:
        bg        for
        nop
        mov       1, %g1
        ta         0
```

```
 ld     [%fp+nums_s], %l1
        orcc  %g0, 9, %l0
        add %fp, nums_s+ 4*9, %l3
        ba        fortest
        nop
For:
        ld        [%l3], %o0
        sub         %l3, 4, %l3
        cmp       %o0, %l1
        bge       keep
        nop
        mov       %o0, %l1
keep:
        subcc       %l0, 1, %l0

fortest:
        bg        for
        nop
        mov       1, %g1
        ta         0
```

static declaration

```
void main(){

  static int nums[20] = {17,11,-161,-32,-893,566,25,88,67,-90};
  int n = 10;           /* number of elements in array */

  register int i;               /* for index */
  register int min;             /* to hold the minimum element */

  min = nums[0];                /* initialize min to first element */

  for (i = 1; i < n; i++)       /* run through rest of array */

    if (nums[i] < min)          /* storing smallest number */
      min = nums[i];
   }
```

```
       .section “.data"
 nums: .skip 4*20

       .section ".text"

        !local variables

         n = -4

        !  index i  in  %l0
        !  max     in  %l1

        .global main
main: save    %sp, -96, %sp   ! -92 -4 & -8

      set     nums, %l2

      mov     17, %o0
      st      %o0, [%l2 + 0]
      mov     11, %o0
      st      %o0, [%l2 + 4]
```

```
mov     -161, %o0
st      %o0, [%l2 + 8]
mov     -32, %o0
st      %o0, [%l2 + 12]
mov     -893, %o0
st      %o0, [%l2 + 16]
mov     566, %o0
st      %o0, [%l2 + 20]
mov     25, %o0
st      %o0, [%l2 + 24]
mov     88, %o0
st      %o0, [%l2 + 28]
mov     67, %o0
st      %o0, [%l2 + 32]
mov     -90, %o0
st      %o0, [%l2 + 36]

mov     10, %o0                     ! n = 10
st      %o0, [%fp + n]
```

```
      ld      [%l2], %l1            ! min initialization
                                    ! min = nums[0]
      ba       fortest              !
      mov     1, %l0                !

for:   sll     %l0, 2, %o0          ! o0 = i * 4
      add     %l2, %o0, %o0         ! o0 = %l2 + i * 4
      ld      [%o0], %o0            ! load  nums[i]
      cmp     %o0, %l1              ! nums[i]  ?  min
      ble     keep                  !     <=  then branch
      nop
      mov     %o0, %l1              !      min update
keep:
       add     %l0, 1, %l0          ! i++
fortest:
       ld      [%fp + n], %o0       !  n   loading
       cmp     %l0, %o0             !  i  ?  n
       bl      for                  !     <  then  branch
       nop
       mov     1, %g1               !     >= then  exit
       ta      0
```

Multi-dimensional array

int a[3][4];

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Row no. Column no.

Row major: C, Pascal
Column major: Fortran

memory

Row major
a[0][0]
a[0][1]
a[0][2]
a[0][3]
a[1][0]
a[1][1]
a[1][2]
…
a[2][2]
a[2][3]

Column major
a[0][0]
a[1][0]
a[2][0]
a[0][1]
a[1][1]
a[2][1]
a[0][2]
…
a[1][3]
a[2][3]

Address calculation in multi-dimensional array

- Address of a[i][j]? $0 \le i < R,\ 0 \le j < C$

  R: num. rows C: num. column W: element size

- Row major case

  addr a[i][j] = addr a[0][0] + (i * C + j) * W

- Column major case

  addr a[i][j] = addr a[0][0] + (j * R + i) * W

- addr a[0][0] = %fp + a_s

- Address of element with index i,j,k in 3D array of int arr [di][dj][dk] =>

  %fp + arr_s + [{i * dj + j} * dk + k]*(element size)

- What is the address of a[1,2,3] in 3D array a[2][3][4] (where dj =3, dk =4)?

  1: Num. of elements with different 1st dimension index before a[1,2,3] : i * dj * dk

  2: Num. of elements with different 2nd dimension index before a[1,2,3]: j * dk

  3: Num. of elements with different 3rd dimension index before a[1,2,3]: k

a[0,0,0] a[1,0,0]
a[0,0,1] a[1,0,1]
a[0,0,2] a[1,0,2]
a[0,0,3] a[1,0,3]

a[0,1,0] a[1,1,0]
a[0,1,1] a[1,1,1]
a[0,1,2] a[1,1,2]
a[0,1,3] a[1,1,3]

a[0,2,0] a[1,2,0]
a[0,2,1] a[1,2,1]
a[0,2,2] a[1,2,2]
a[0,2,3] a[1,2,3]

Address calculation example

- Load 2 bytes to a register from the element with index i,j,k in the three dimensional array of short ary [2] [3] [4]

※ Address calculation

%fp + ary_s + {(i * dj + j) * dk + k } * 2

- ✓ ary_s : -2*3*4 * 2
- ✓ dj : 3
- ✓ dk: 4

```
ary_s = -2*3*4 * 2
di = 2
dj = 3
dk = 4
   !  i →  %i_r   j → %j_r   k →  %k_r
mov   %i_r, %o0
call     .mul
mov   dj,   %o1              ! %o0 = i * dj

add   %j_r, %o0, %o0         ! %o0 = i * di + j
call     .mul
mov   dk,   %o1              ! (i * di + j) * dk

add   %k_r, %o0, %o0         ! (i * di + j) * dk + k
sll  %o0,  1,  %o0           ! (i * di + j) * dk + k * 2
add    %fp,  %o0,  %o0
ldsh     [%o0 + ary_s],  %o0   ! %o0 = ary [i][j][k]
 :
```

Program example

```
main()
{
  int score[3][4]={{68,55,90,88},{78,77,89,91},
                   {93,95,89,98}};
    int  sum[3] = {0};

    register int  i,j;

    for (i=0; i<3; i++)
    for (j=0; j<4; j++)
        sum[i] = sum[i] + score[i][j];
}
```

```
     score_s = -48              !  3*4*4
     sum_s = -60
                 ! i : %l0, j : %l1
     .global main
main: save  %sp, -152, %sp  ! -(92 + 3*4*4 + 3*4) & -8
       :                        ! Initialize score & sum
       :
      ba     outer_test    ! branch to outer loop test
      mov   0, %l0         ! use delay slot for initialization
                           ! statement   z = 1
inner:                              !code for inner loop
      sll     %l0, 2, %o0        ! i * 4
      add    %fp, %o0, %o0       ! %fp + i *4
      ld      [%o0 + sum_s], %l2 ! load  sum[i]  in  %l2
      sll     %l0, 2, %o0        ! i * 4
      add    %o0, %l1, %o0       ! (i * 4) + j
      sll      %o0, 2, %o0       !  (i * 4) + j  * 4
      add  %fp, %o0, %o0         ! %fp +  (i * 4) + j  * 4
      ld  [%o0 + score_s], %l3   !load  score[i][j]  in %l3
```

```
        add  %l2, %l3, %l4      ! sum[i] = sum[i] + score[i][j]
        sll    %l0, 2, %o0           ! i * 4
        add    %fp, %o0, %o0         ! %fp + i *4
        st     %l4, [%o0 + sum_s]  ! store %l4 into sum[i]
inner_inc:                      !inner for increment statement
        add     %l1, 1, %l1          ! j++
inner_test:                     !inner for test
        cmp     %l1, 4               !  j  ?  4
        bl       inner               !   <  then  branch
        nop
outer_inc:                      !outer for increment statement
        add     %l0, 1, %l0          !   < then i++
outer_test:                     !outer for test
        cmp     %l0, 3               ! i  ?  3
        bl,a      inner_test         !     <   then branch
        mov     0, %l1               !  j = 0
        mov     1, %g1               !   >= then  exit
        ta       0
```

Boundary test

- What if min index is not 0? To locate a element we need to calculate distance from the min index.

- Example.

  ary[−2..3]

  → ary[−2] ary[−1] ary[0]

  ary[1] ary[2] ary[3]

  → distance of ary[1]?

  $1-(-2) = 3$

Example.

int ary[l1..u1, l2..u2, l3..u3]

dimension size : $d1 = u1 - l1 + 1$

$d2 = u2 - l2 + 1$

$d3 = u3 - l3 + 1$

memory size: $\underbrace{d1 * d2 * d3}_{\text{Num. elements}} * 4$

- Address of element with index i, j, k :

$$\%fp + b\_s + [\{(i - l1) * d2 + (j - l2)\} * d3 + (k - l3)] * 4$$

or

$$\%fp + b\_s + (i * d2 + j) * d3 + k * 4$$
$$- (l1 * d2 + l2) * d3 + l3 * 4$$

- Boundary test: check if the indices are within their valid ranges

$$\text{lower\_bound} <= \text{index} <= \text{upper\_bound}$$
$$0 <= \text{index} - \text{lower\_bound} <= \text{upper\_bound} - \text{lower\_bound}$$
$$0 <= \text{index} - \text{lower\_bound} < \text{dimension}$$
$$[= (\text{upper\_bound} - \text{lower\_bound} + 1)]$$

- Example

  int arr [−2..3, 0..9, 2..4] declaration

  indices are stored in %l0(i) %l1(j) %l2(k)

  do boundary test when accessing arr[i][j][k]

```
    l_1 = −2      u_1 = 3
    l_2 = 0       u_2 = 9
    l_3 = 2       u_3 = 4
    d1 = 6                    ! u_1 – l_1 + 1
    d2 = 10
    d3 = 3
    arr_s = − 6*10*3 * 4
  :
subcc  %l0, l_1, %o1     !  i − l_1
bl       error             ! (i − l_1) < 0  then branch
cmp     %o1, d1           ! (i − l_1) ? d1
bge      error             !          >=   then branch
```

```
   add      %o1, %g0, %o0
   call     .mul                        ! (i - l_1) * d2  in %o0
   mov      d2, %o1
   subcc    %l1, l_2, %o1               ! j - l_2
   bl       error                       ! (j - l_2) < 0 then branch
   cmp      %o1, d2                     ! (j - l_2) ? d2
   bge      error                       !           >=   then branch
   add      %o1, %o0, %o0               ! (i - l_1) * d2 +  (j - l_2)
   call     .mul                        ![                          ] * d3 in %o0
   mov      d3, %o1
   subcc    %l2, l_3, %o1               ! ( k - l_3)
   bl       error                       !           < 0 then  branch
   cmp      %o1, d3                     ! (k - l_3) ? d3
   bge      error                       !           >=   then  branch
   add      %o1,%o0,%o0               !    [(i-l_1)*d2+(j-l_2)]*d3 + (k-l_3)
   sll      %o0, 2, %o0                 !    {            } * 4
   add      %fp, %o0, %o0
   ld       [%o0 +  arr_s], %o0  ! %o0 = ary[i][j][k]
                :
error:
                :
```

Methods for fast address calculation

- Strength reduction
  - ✓ replace multiplication with shift & add

    example: i * 4

- $\%o0 * 5_{10} = \%o0 * (4 + 1) = \%o0 * 2^2 + \%o0$

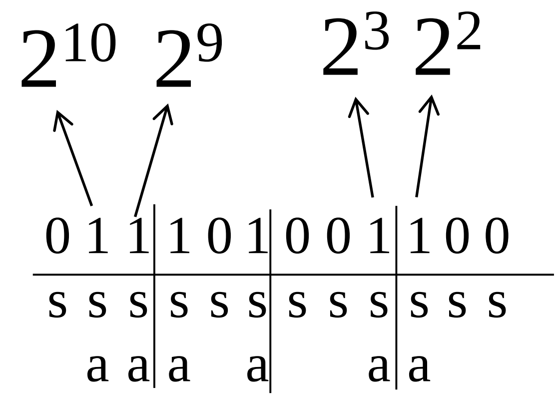
  $(5_{10} = 101_2)$

```
mov     %o0, %o1
sll     %o0, 2, %o0
add     %o0, %o1, %o0
```

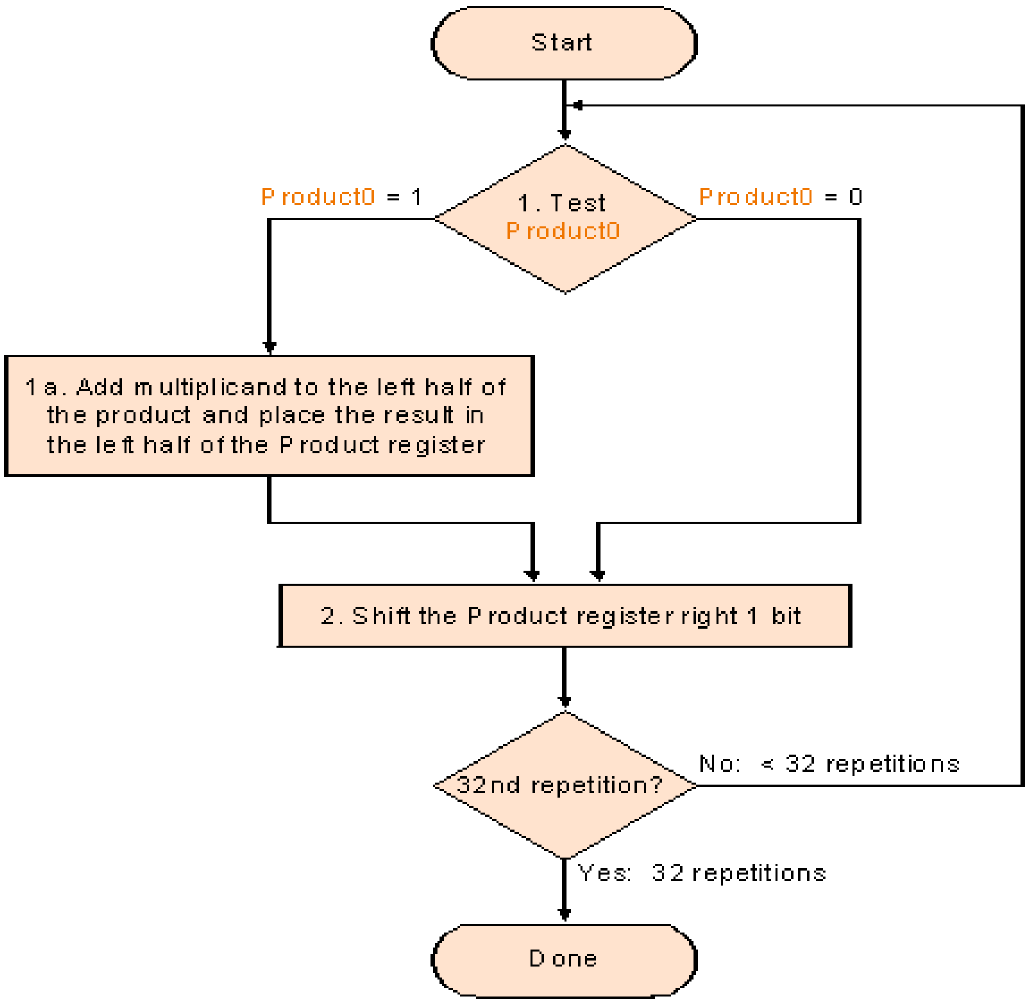
```
add   %o0, %g0, %o0
call  .mul
mov   5,  %o1
```

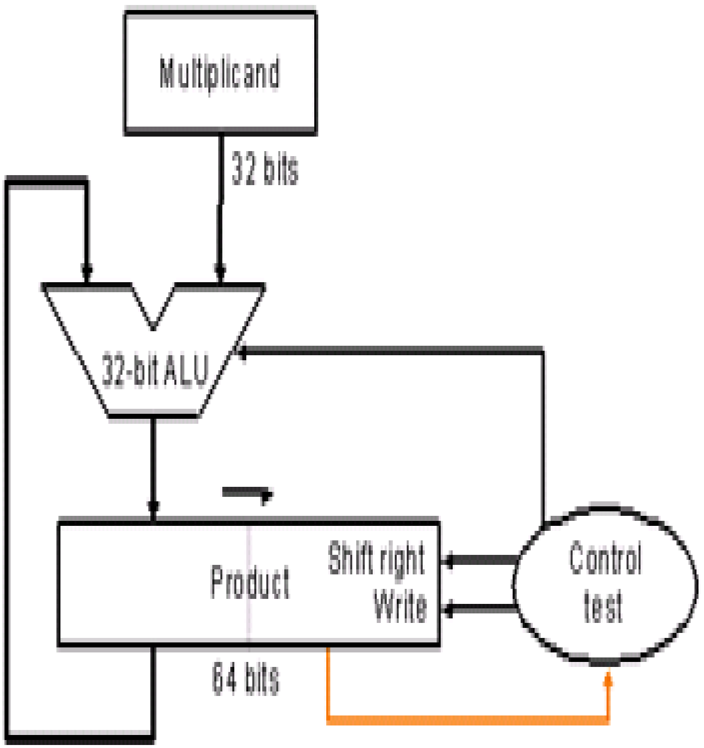
%o0 * 03514

```
clr   %o1              ! %o1 = 0
sll   %o0, 2, %o0      ! %o0 = %o0 * 4
add   %o0, %o1, %o1
sll   %o0, 1, %o0      ! %o0 = %o0 * 8
add   %o0, %o1, %o1
sll   %o0, 3, %o0      ! %o0 = %o0 * 64
add   %o0, %o1, %o1
sll   %o0, 2, %o0      ! %o0 = %o0 * 256
add   %o0, %o1, %o1
sll   %o0, 1, %o0      ! %o0 = %o0 * 512
add   %o0, %o1, %o1
sll   %o0, 1, %o0      ! %o0 = %o0 * 1024
add   %o0, %o1, %o1
```



Multiplication algorithm/hardware





Booth Recoding: reduce additions

$A * 01110_2$ $\quad 14_{10}$

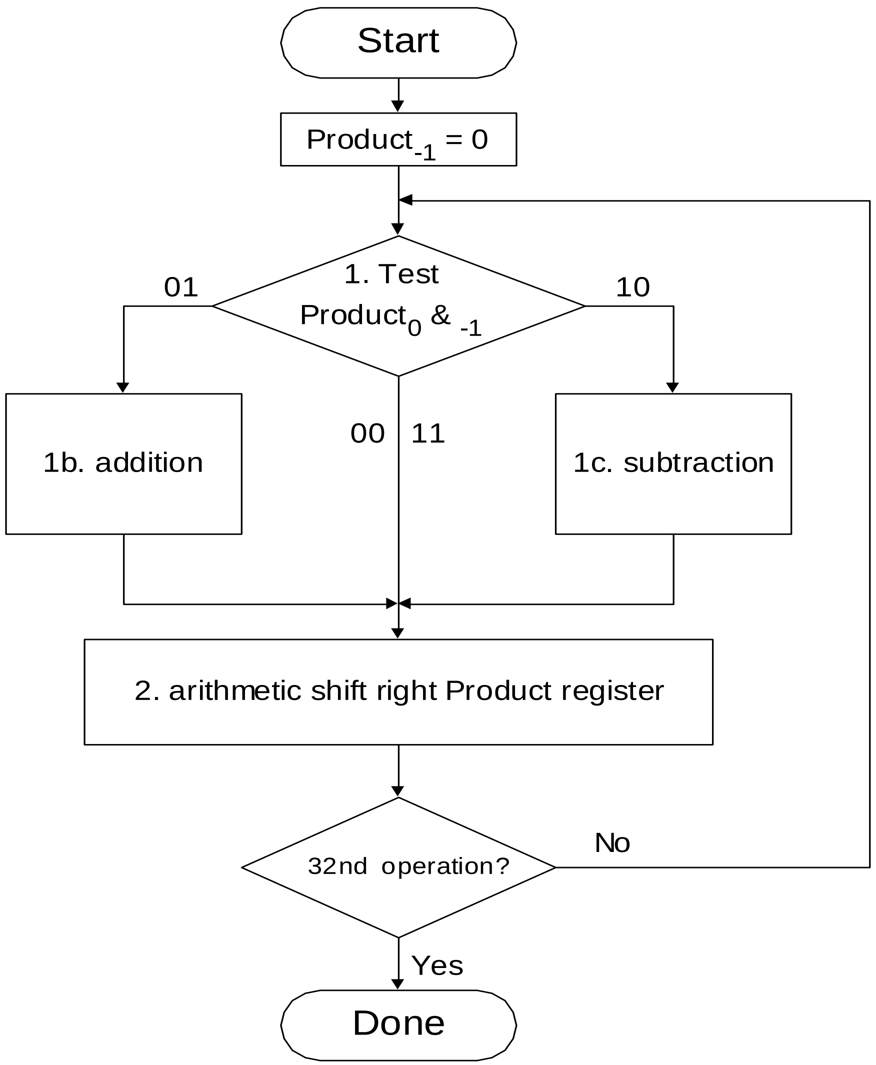
$= A * (10000 - 10)$ $\quad = 16_{10} - 2_{10}$

$= A * 10000 - A * 10$ $\quad = 2^4 - 2^1$

change	operation
$0 \leftarrow 0$	shift
$1 \leftarrow 1$	shift
$1 \leftarrow 0$	sub, shift
$0 \leftarrow 1$	add, shift

prev.	cur.	operation	meaning
	0	P = 0	Initial state
0	0	A << 1	
1	0	P = P - A, A << 1	P = -A * 10
1	1	A << 1	
1	1	A << 1	
0	1	P = P + A, A << 1	P = A*10000-A*10

Booth's Algorithm



- Example

$\%o0 * 7_{10} \;\; => \;\; \%o0 * 0111_2\,0$

(assume there is 0 right to lsb)

```
sub  %g0, %o0, %o1
sll  %o0, 3, %o0
             ! shift-left, %o0 * 2^3
add  %o0, %o1, %o1
             ! %o1 = %o0 * 7
```

Structure

- Alignment problem
  - ✓ How to align fields?
    → use positive offset
  - ✓ How to align structure itself?
    → consider biggest field
- Example

```
struct example {
    int a, b;
    char d;
    short x, y;
    int u, v;        }
```

24 bytes

offset	member/field
0	example.a
4	example.b
8	example.d
10	example.x
12	example.y
16	example.u
20	example.v

offset	← byte →	field
0		
1		a
2		
3		
4		
5		b
6		
7		
8		d
9		
10		x
11		
12		y
13		
14		
15		
16		
17		u
18		
19		
20		
21		v
22		
23		

1. Alignment of each member/field
2. Structure alignment? multiple of 4

- Address of a **field**: starting address + **offset**

- Code example (%l0 : pointer to the first element)

```
example_a = 0
example_b = 4
example_d = 8
example_x = 10
example_y = 12
example_u = 16
example_v = 20
ld      [%l0 + example_a] , %o0  ! %o0 = example.a
ld      [%l0 + example_b] , %o1  ! %o1 = example.b
ldub    [%l0 + example_d] , %o2  ! %o2 = example.d
ldsh    [%l0 + example_x] , %o3  ! %o3 = example.x
ldsh    [%l0 + example_y] , %o4  ! %o4 = example.y
ld      [%l0 + example_u] , %o5  ! %o5 = example.u
ld      [%l0 + example_v] , %l1  ! %l1 = example.v
```

- When alignment is needed for each structure variables

```
struct example {
    int a, b;
    char d;
    short x, y;
    int u;
   short v;        }
```

offset	member
0	example.a
4	example.b
8	example.d
10	example.x
12	example.y
16	example.u
20	example.v

22 bytes

offset	← byte →	field
0		
1		a
2		
3		
4		
5		b
6		
7		
8		d
9		
10		x
11		

12		y
13		
14		
15		
16		
17		u
18		
19		
20		
21		v

What should be the starting address of each structure variable?

```
main()
{
struct student{
    char name[20];
    int  scoremath;
    int  scoreengli;
    int  average;
                };

struct student s = {"kim", 95, 98, 0};

s.average = (s.scoremath + s.scoreengli)/2;

    :
}
```

```
        scoremath_s = 20
        scoreengli_s = 24
        average_s = 28
        student_s = -32
        .global main
main:  save  %sp, -128, %sp   ! -(92+32) & -8
        :
        mov  95, %o0
        st  %o0, [%fp + student_s + scoremath]
        mov  98, %o0
        st  %o0, [%fp + student_s + scoreengli]
        mov   0, %o0
        st  %o0, [%fp + student_s + average]
        :
        ld  [%fp + student_s + scoremath], %l0
        ld  [%fp + student_s + scoreengli], %l1
        add  %l0, %l1, %l0   !  sum
        srl  %l0, 1, %l0      !  /2
        st  %l0, [%fp + student_s + average]
         :
```

Alignment Map

```
struct date {
    char day0, month1;
    short year2;
} d1, d2;
struct person {
    char name[21]0;
    int ss24;
    struct date birth28, marriage32;
    char married36, sex37
} p1, p2;
```

addr	field	size
-88	p2.name	21
-64	p2.ss	4
-60	p2.birth	4
-56	p2.marriage	4
-52	p2.married	1
-51	p2.sex	1
-48	p1.name	21
-24	p1.ss	4
-20	p1.birth	4
-16	p1.marriage	4
-12	p1.married	1
-11	p1.sex	1
-8	d2.day	1
-7	d2.month	1
-6	d2.year	2
-4	d1.day	1
-3	d1.month	1
-2	d1.year	2

- fields in date

0		day
1		month
2		year
3		

✓ structure size: 4 bytes

✓ biggest field: 2 bytes

- fields in person

0		name[0]
20		name[20]
21		
24		ss
28		birth
32		marriage
36		married
37		sex
38		
39		

✓structure size: 38

✓ biggest field: 4

→ revised size: 40

Example

```
!define structure date
date_day = 0
date_month = 1
date_year = 2

! align_of_date, 2 bytes
! size_of_date, 4 bytes

!define structure person
person_name = 0
person_ss = 24
person_birth = 28
person_marriage = 32
person_married = 36
person_sex = 37
```

```
struct date d1, d2;
struct person p1, p2;

d1.day = 13;
d1.month = 5;
d1.year = 1997;

p1.birth = d1;
p2.marriage.day = 3;
p1.sex = p2.sex;
```

```
        ! align_of_person, 4 bytes
        ! size_of_person, 40 bytes
        ! local variables
        d1 = -4
        d2 = -8
        p1 = -48
        p2 = -88

        .global main
main:   save    %sp, -184, %sp  !  [ (-92)-(40*2+4*2)] & -8
        mov     13, %o0                         !d1.day = 13;
        stb      %o0, [%fp + d1 + date_day]
        mov     5, %o0                          !d1.month = 5;.
        stb     %o0, [%fp + d1 + date_month]
```

```
mov     1967, %o0                       !d1.year = 1967;
sth     %o0, [%fp + d1 + date_year]

ld      [%fp + d1], %o0                 !p1.birth = d1
st      %o0, [%fp + p1 + person_birth]
!all four bytes will fit into a single register
mov     3, %o0                          !p2.marriage.day = 3;
stb     %o0, [%fp + p2 + person_marriage + date_day]

ldub    [%fp + p2 + person_sex], %o0    !p1.sex = p2.sex;
stb     %o0, [%fp + p1 + person_sex]
```