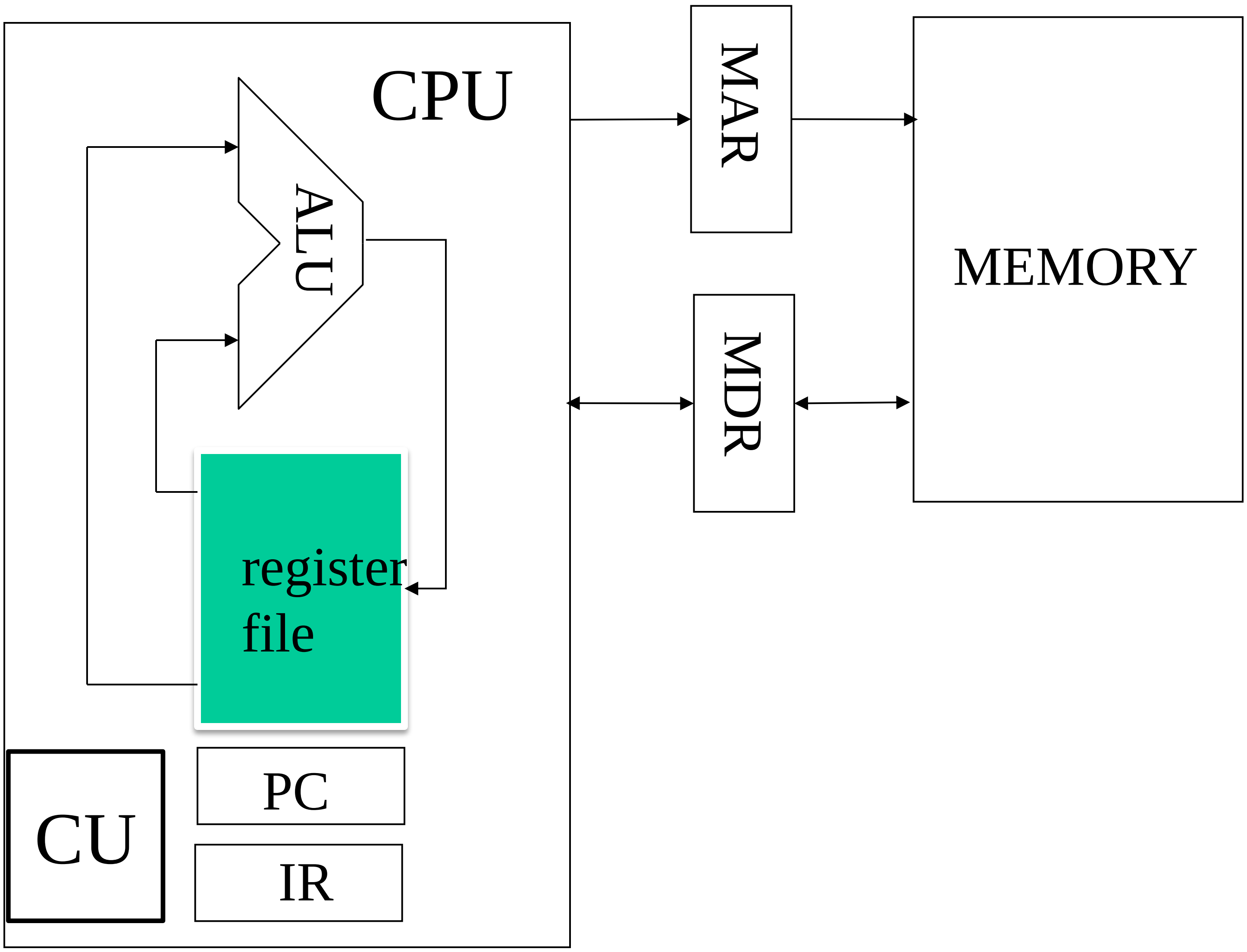
SPARC Instruction Set Architecture

Characteristics

- Load/Store architecture
- 32 registers are visible at any point
  - ✓ size of each register: 32 bits
- Byte addressable (memory addressing)
- 2^{30} instructions or integers

Load/store architecture



Registers

- 4 classes, 8 per class

- ***g*** (global) registers (%r0 - %r7)
  - %g0: always **0**,
  - %g1 - %g7: global data

- ***o*** (out) registers (%r8 - %r15)
  - %o0 - %o5: arguments, local data
  - %o6: %sp (stack pointer)
  - %o7: return address

Registers

- ***l*** (local) registers (%r16 - %r23)
  - %l0 - %l7: local variables
- ***i*** (in) registers (%r24 - %r31)
  - %i0 - %i5: arguments
  - %i6: %fp (frame pointer)
  - %i7: return address

SPARC registers

Register name	← 32 bit →
%g0	0
%g1	
:	
%g7	
%l0	
%l1	
:	
%l7	

Register name	← 32 bit →
%i0	
:	
%i6(=%fp)	mem. address
%i7	return address
%o0	
:	
%o6(=%sp)	mem. address
%o7	return address

Assembly programming (1)

- One instruction (or definition) per line
- Label
  - ✓ A string ends with colon(:)
- Comment
  - ✓ C style(/* … */) or starts with !
- Classification of instructions
  - ✓ Machine instructions
  - ✓ Synthetic instructions
  - ✓ Pseudo-ops: provide information to the assembler but do not generate instructions.

Assembly programming (2)

- Pseudo-op starts with period; Pseudo-ops provide information to the assembler but do not generate instructions
  - ✓ examples

    .word : memory allocation and initialization

    .global : enabling access from outside of program

- Synthetic instructions: translated into other machine instructions; exist for clearer/concise representation
  - ✓ ex: clr , mov

- Machine instructions: instructions supported by hardware
  - ✓ ALU instructions: add, sub, and
  - ✓ control inst.: call, ba
  - ✓ memory access inst.: ld, st

Instruction format (1)

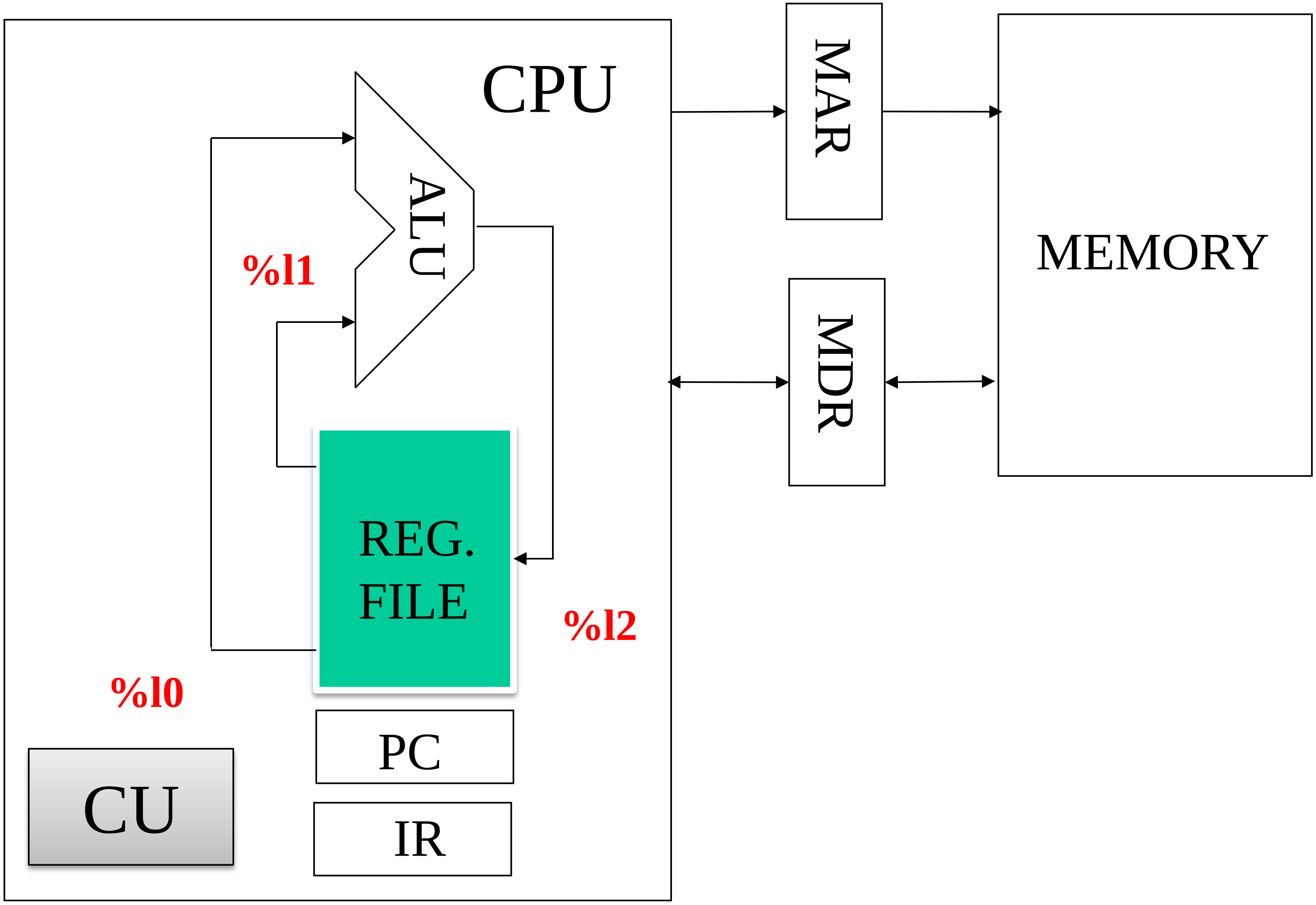
1. OP S, A, R ! (S) op (A) $\rightarrow$ (R)
   - S: source register 1
   - A: source register 2 or immediate value k
   - R: destination register
   - OP: arithmetic/logic operation symbol
     - ✓ 예: add, sub

❖ range of immediate value k : $-4096 \le k < 4096$

($\because$ 13-bit signed number)

✓ ex: add %l0, %l1, %l2

add %l0, -5, %l2


CPU
ALU
%l1
REG. FILE
%l2
%l0
PC
IR
CU
MAR
MDR
MEMORY

Instruction format (2)

2. OP A, R ! (A) → (R)
   - A: source register or immediate value
   - R: destination register
   - ✓ ex: mov %l0, %l1
     mov 3, %l1

3. OP R
   - R: destination register or address(label)
   - ✓ ex: clr %l0
     call .mul

Program Example (1)

- < c-type lang.> < Assemble >

c-type lang.	Assemble
x = 5	mov 5, %l1
y = 3	mov 3, %l2
w = x + y	add %l1, %l2, %l3
z = x - y + w	sub %l1, %l2, %l5
	add %l5, %l3, %l4

- Mapping assumption:

variable	register
x	%l1
y	%l2
w	%l3
z	%l4

Program example (2)

```
        A = 10
        B = 15
        C = 20
        .global    main
main:   save     %sp,    -96,    %sp     ! Allocate stack frame
        mov      A,      %o0             ! Put 10 into %o0
        add      %o0,    B,      %o0     ! Add 15 to %o0
        add      %o0,    C,      %o0     ! Add 20 to %o0
        restore                          ! Deallocate stack frame
        ret                              ! return
        nop
```

Arithmetic operations

- Addition(add) and subtraction(sub) are hardware instructions
- Multiplication and division are subroutines
  - ✓ Arguments passing: use out registers

✓ b * c calculation

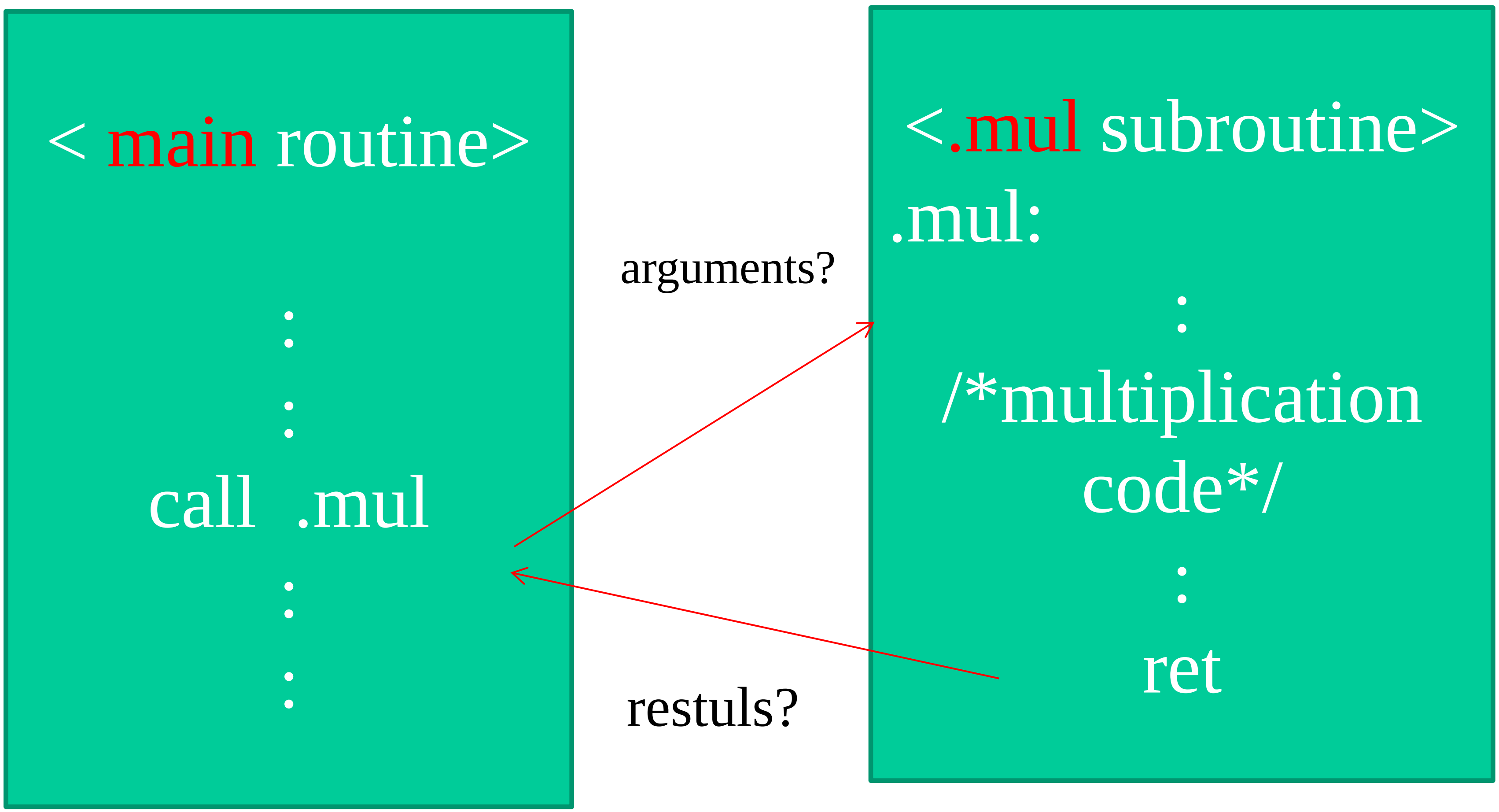
```
mov   b, %o0
mov   c, %o1
call  .mul
nop
```

✓ b / c calculation

```
mov   b, %o0
mov   c, %o1
call  .div
nop
```

Results are retuned via %o0

Use of subroutine



- Return address stored in %o7

Program example (3)

/* y = (x - 1) * (x - 7) / (x - 11) where x = 9 */

↑ ↑ ↑ ↑ ↑ ↑ ↑

```
    .global main
main:
    save    %sp, -96,  %sp
    mov     9,    %l0          ! x initialization
    sub     %l0,  1,    %o0    ! Store (x - 1) in %o0
    sub     %l0,  7,    %o1    ! Store (x - 7) in %o1
    call    .mul
    nop                        ! Multiplication result is in %o0
    sub     %l0,  11,   %o1    ! Store (x - 11) in %o1 (divisor)
    call    .div
    nop                        ! Division results is in %o0
    mov     %o0,  %l1          ! Store result in y
    mov     1,    %g1          ! Trap preparation to exit
    ta      0                  ! Trap
```

%l0: x
%l1: y
%o0: argument 1
%o1: argument 2

Instructions details

- SAVE instruction

① Allocate a new register set (24)

② Allocate space for a stack frame (96 bytes)

- TA instruction: to use system service

%g1	service to request
1	exit
2	fork
3	read
4	write
5	open
6	close
8	create

Generating executable files (1)

- Compiler/assembler to use: gcc
  - ✓ Compiles c code but works as a assembler when extension is s
    - .o : objective files
    - **.s** : assembly source code file
- Generating exe files
  - ✓ To **assemble**: gcc -g expr.s -o expr
  - ✓ To **execuate**: expr

❖ Generation of assembly program from C program

- ✓ gcc –S pgm.c

Debugging: gdb

- Using gdb
  - Can monitor execution status of a program
  - No effects on results of program
  - Can execute instructions one by one
  - Can use break points
  - Can track values of registers and variables

Using debugger (1)

```
ce2:/lab/hps/pyo% gdb  expr
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(gdb) r
Starting program: /lab/hps/pyo/a.out
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...
Program exited with code 0370.
(gdb) b main
Breakpoint 1 at 0x105f8
(gdb) r
Starting program: /lab/hps/pyo/a.out
(no debugging symbols found)...
Breakpoint 1, 0x105f8 in main ()
```

executable file

run

breakponit

```
(gdb) x/i $pc
0x105f8 <main+4>:       mov  9, %l0
(gdb)
0x105fc <main+8>:       sub  %l0, 1, %o0
(gdb) x/12i main
0x105f4 <main>: save  %sp, -96, %sp
0x105f8 <main+4>:       mov  9, %l0
0x105fc <main+8>:       sub  %l0, 1, %o0
0x10600 <main+12>:      sub  %l0, 7, %o1
0x10604 <main+16>:      call  0x20750 <.mul>
0x10608 <main+20>:      nop
0x1060c <main+24>:      sub  %l0, 0xb, %o1
0x10610 <main+28>:      call  0x20714 <.div>
0x10614 <main+32>:      nop
0x10618 <main+36>:      mov  %o0, %l1
0x1061c <halt>: mov  1, %g1
0x10620 <halt+4>:       ta  0
```

examine/ instruction at the address in PC

examine 12 instructions from the address “main” (gdb) b *&main+44

label “main”

Breakpoint 2 at 0x10620

(gdb) c

Continuing.

Breakpoint 2, 0x10620 in halt ()

(gdb) p $l1

$1 = -8

print register contents

(gdb) r

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /lab/hps/pyo/a.out

(no debugging symbols found)...(no debugging symbols found)...

(no debugging symbols found)...

Breakpoint 1, 0x105f8 in main ()

(gdb) x/i $pc

0x105f8 <main+4>:   mov 9, %l0

(gdb) ni

next instruction

0x105fc in main ()

```
(gdb) p $l0
$2 = 9
(gdb) display/i $pc
```

always display

```
2: x/i $pc  0x105fc <main+8>:   sub  %l0, 1, %o0
(gdb) ni
0x10600 in main ()
2: x/i $pc  0x10600 <main+12>:  sub  %l0, 7, %o1
(gdb)
0x10604 in main ()
2: x/i $pc  0x10604 <main+16>:  call  0x20750 <.mul>
(gdb)
0x10608 in main ()
2: x/i $pc  0x10608 <main+20>:  nop
(gdb)
0x1060c in main ()
2: x/i $pc  0x1060c <main+24>:  sub  %l0, 0xb, %o1
(gdb) q
The program is running.  Exit anyway? (y or n) y
ce2:/lab/hps/pyo%
```

Using debugger (2)

```
        .global main
main:   save    %sp, -96, %sp
        mov     9, %l0
l1:     sub     %l0, 1, %o0
        sub     %l0, 7, %o1
        call    .mul
        nop
        sub     %l0, 11, %o1
        call    .div
        nop
        mov     %o0, %l1
        mov     1, %g1
        ta      0
```

```
(gdb) b l1
(gdb) r
(gdb) p $l0     →    9
(gdb) p $o0     →    ?
(gdb) ni
(gdb) p $o0     →    8
(gdb) p $o1     →    ?
(gdb) ni
(gdb) p $o1     →    2
(gdb) ni
(gdb) p $o0     →    8
(gdb) ni
(gdb) p $o0     →    16
```

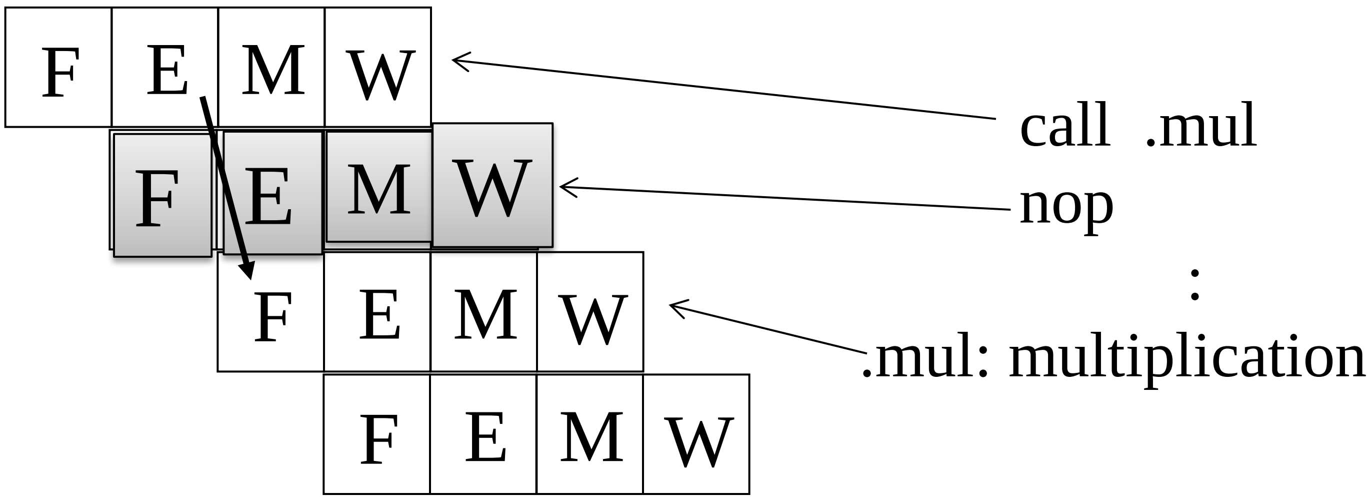
Use of delay slots

```
        .global main
main:
        save    %sp,    -96,    %sp
        mov     9,      %l0

        sub     %l0,    1,      %o0
        sub     %l0,    7,      %o1
        call    .mul
        nop
        sub     %l0,    11,     %o1
        call    .div
        nop
        mov     %o0,    %l1
        mov     1,      %g1
        ta      0
```

- Instructions right below call, branch instructions are executed

Delay slot



- Due to the pipeline structure, instructions right next to branch instructions are executed: otherwise need to clear pipeline
- nop: no operation

```
        .global main
main:
        save    %sp, -96, %sp
        mov     9, %l0
        sub     %l0, 1, %o0
        call    .mul
        sub     %l0, 7, %o1
        call    .div
        sub     %l0, 11, %o1
        mov     %o0, %l1
        mov     1, %g1
        ta      0
```

- Performance enhancement?

- Execution steps