- Lecture - 10 | 26/09/2019 -

Last lecture -

- Architecture class (Instruction execution model)
- Hardware - software interaction
- Machine Instruction format
- MIPS instruction format
- Addressing mode.

Today :-

- Addressing mode (MIPS)
- Register set
- procedure call
- x86-64 instruction set architecture
  - Instruction format
  - Addressing mode
  - Register set

Addressing Mode

- Immediate addressing mode → Operand
  - The data is directly available from instruction register.

Example :-

```
addi $sp $sp 8
```

sp ← [sp] + 8 (8 → immediate data)

[sp] → Register data

Supported in I-type & J-type format

- Register Addressing mode
  - The operand is available in register

Example :-

```
add to a0 a1
```

to ← [a0] + [a1]

[a0], [a1] → operand in register a0, a1

Supported in R-type format

(2)

- Base or Displacement - addressing mode :-

→ An operand is available in a memory

Example :-

→ The memory location is calculated as

Base + index

Example :-

array[10] = k + array[10]

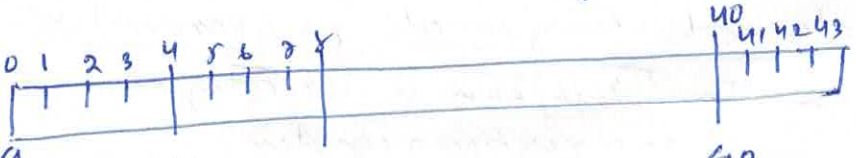
Let t_1 be the base of array

s_2 be storing k

```
lw  to, 40(t1)   //  to ← [[t1] + 40]
add to, s2, to   //  to ← [s2] + [to]
sw  to, 40(t1)   //  [t1] + 40 ← [to]
```

Why 40 ?

- byte addressable
- a word is 4 byte



a[0] a[1] a[2] a[10]

a+(0×4) a+1×4 a+2×4 a+10×4

- PC - Relative addressing mode :-

An operand is specified in relative memory.

Example :-

beq s0, s1 L1

→ register

→ offset

→ register

if [s0] is equal to [s1]

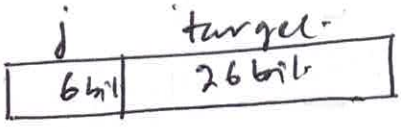
PC ← [PC] + 4 + L1

- Pseudo-direct Addressing Mode :-

An operand is memory address itself which is generated from the given target address.

Example :-

j target-



jump to the location address

address ← pc[31, 30, 29, 28] . 2shift-2[target]

Explanation :-

→ Left shift by 2 to ensure a good gap between the current-instruction and the target-

→ Concatenate with MSB of pc to ensure the region.

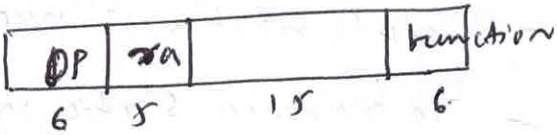
pc ← address

Register Direct- Addressing :-

An operand is memory address that- is specified in register.

Example :-

jr ra



pc ← [ra]

(4)

- MIPS Registers - (Architectural Register visible to programmer)

Name	Number	usage
zero	0	Constant value 0
at	1	Used by assembler
v0 - v1	2 - 3	values for return values
a0 - a3	4 - 7	function argument - (parameter passing)
t0 - t7	8 - 15	Temporaries
s0 - s7	16 - 23	Saved (During function call)
t8 - t9	24 - 25	Temporaries
k0 - k1	26 - 27	operating system Kernel
gp	28	global pointer
sp	29	stack pointer
fp	30	frame pointer
ra	31	return address.

- Function call & Stack :-

- Modularization of program
- To reduce static instruction count - (program size)

Example (function call in high level language) :- C/C++

```
int swap(int,int);
int main(int argc, char* argv) {
    int a,b,c;

    a=0;
    b=0;
    c=0;
    swap(
    c = swap(a,b);
    printf("%d", c);
}
int swap(int a1, int b1) {
    a1 = a1+b1;
    b1 = a1-b1;
    a1 = a1-b1;
    return a1;
}
```

place of return → passing parameters

Recieving parameters, local variables

For sequential machine!

performing computation locally

returning the computed result.

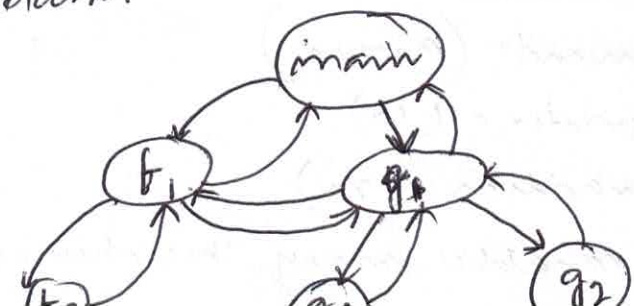
How do we ensure function call/return in Architecture?

- Manage using stack!

Requirement -

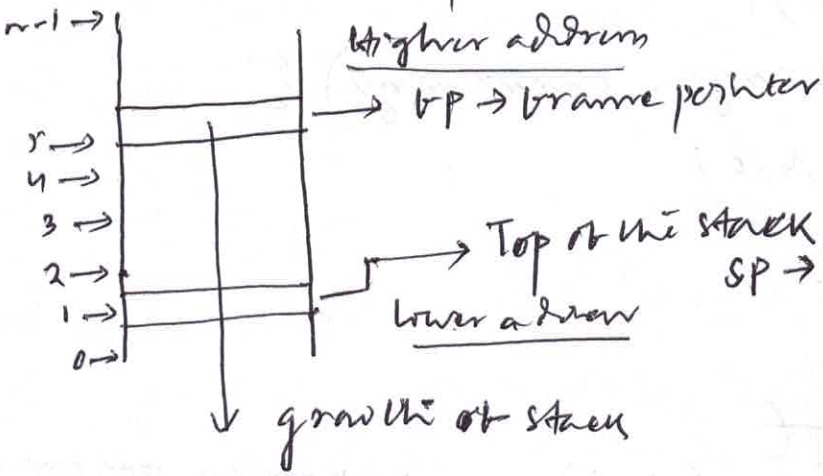
→ The order in which procedures are called, must in the same order they should be returned.

→ All of their local variables must be preserved during call and return.

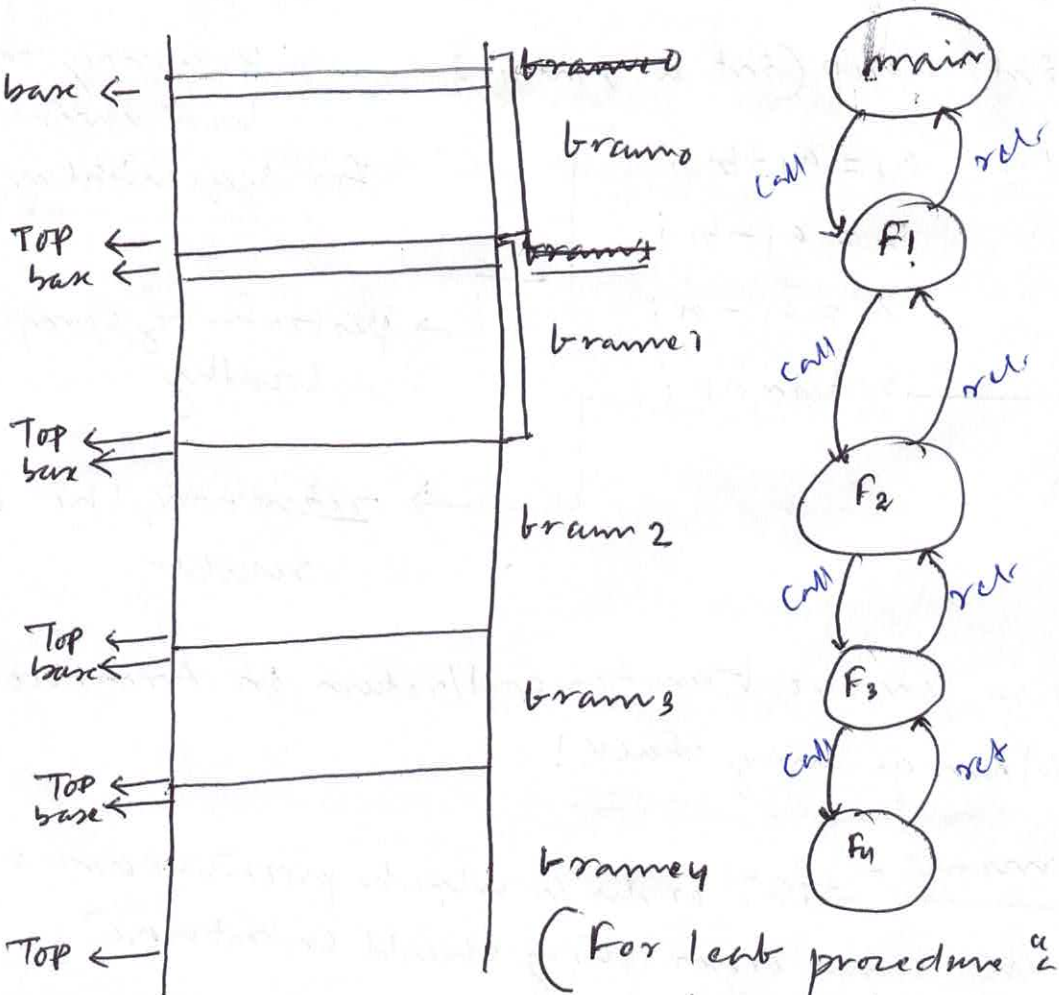


→ leaf procedures

Stack - a portion of memory / address space



Scenario of deeper function call :-



(For leaf procedure "actually" a frame is not needed)

The registers / variable to be saved :-

- → arguments (a0 - a8)
- → stack pointer (sp)
- → return address (ra)
- → Local variable, array, structure etc.

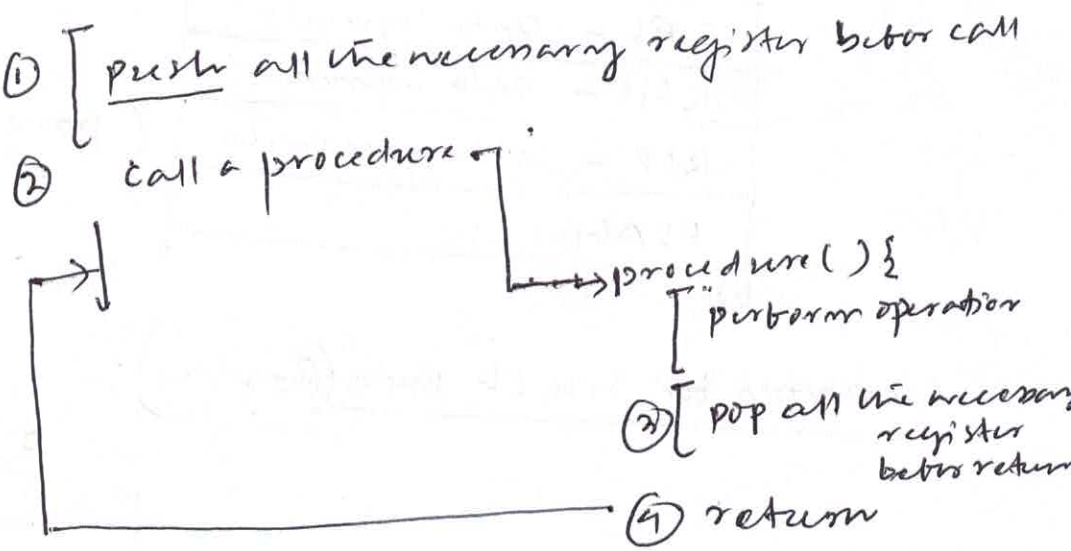
Specific registers & instructions for procedure call (MIPS) :-

Register
- a_0 — a_3 :- registers to pass parameter
- ra :- return address register (What if you want to pass more than 4 parameters)
- s_0 — s_2 :- saved register (to reuse by callee function)
- fp :- frame pointer (Keep track of base address of a procedure ~~stack~~ frame or activation record)
- sp :- stack pointer (to perform stack operations such as push & pop)

Instructions :-

```
lw   a0   0(sp)   // load word equivalent to POP
sw   a0   0(sp)   // store word equivalent to PUSH
jal  callee   → to call a procedure callee
jr   ra       → to return back to the caller function
```

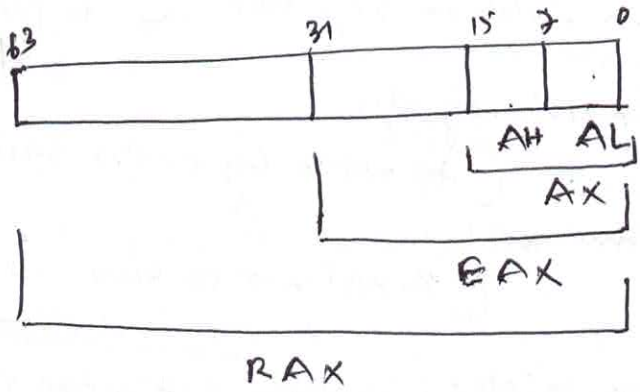
General Rule of push and pop operations :-



(5)

CISC architecture (x86 set - intel) AMD64

Register set - x86_64



RBX ~~RBX~~	→ General purpose
RCX	→ General purpose
RDX	→ General purpose
RSP — Stack pointer	
RBP — Base pointer	
RDI — Destination indexing	
RSI — Source indexing	
RCS — Code segment	
~~SS~~RSS - Stack segment	
RDS — Data segment 1	
RES — Data segment 2	
RFS — Data segment 3	
RGS — Data segment 4	
RIP — Instruction pointer	(program counter)
FLAGS	

63 0

Convention for Size of ~~Data~~ (Register) :-