Jumping and Procedures

Procedures are a powerful stucturing mechanism in programming. Computers provide various support facilities, but success in using procedures rests mostly in following the conventions of register usage.

© 1995 LS All Rights Reserved

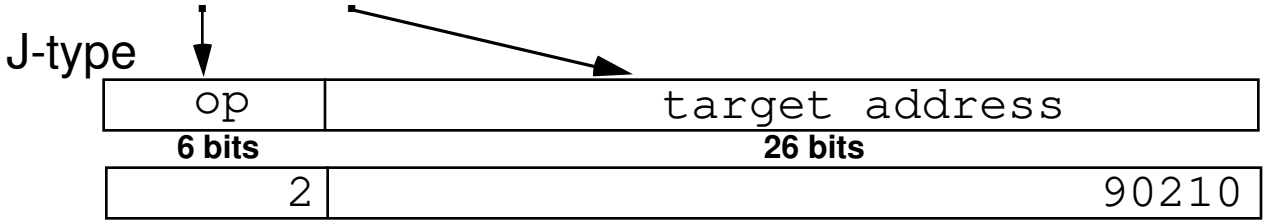
Jumps

There are several forms of unconditional jump instructions.
Jumping unconditionally to an address in a register is `jr` (R-type)

```
jr $10    # jump to  address in $10
```

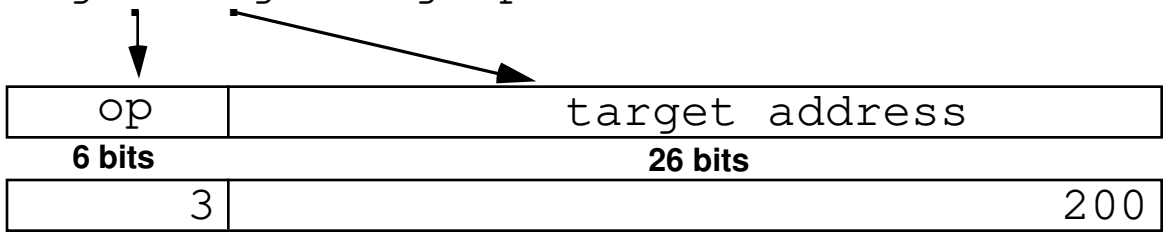
Jumping unconditionally to a target address is, `j` (J-type)

```
j  Start  # jump to Start(s'pose addr=90210)
```



Jump-and-link: Go to a procedure, saving return address, jal (J-type)

```
jal Fuji  # jump & save return addr in $31
```



© 1995 LS All Rights Reserved

Calling a Procedure

The basic mechanism of going to a procedure and returning is embodied in the `jal` instruction, but there is more . . .

$31 must be vacated if a called procedure is to call another

Different registers might contain information when the procedure is called from different locations?

Parameters must be passed to the procedure?

Values must be returned?

. . . conventions solve these matters in a standard way.

A stack (in memory) solves these problems . . .

- Save (push) registers onto the stack for use later
- Pass information through specific registers
- Restore state by loading (popping) values off stack

© 1995 LS All Rights Reserved

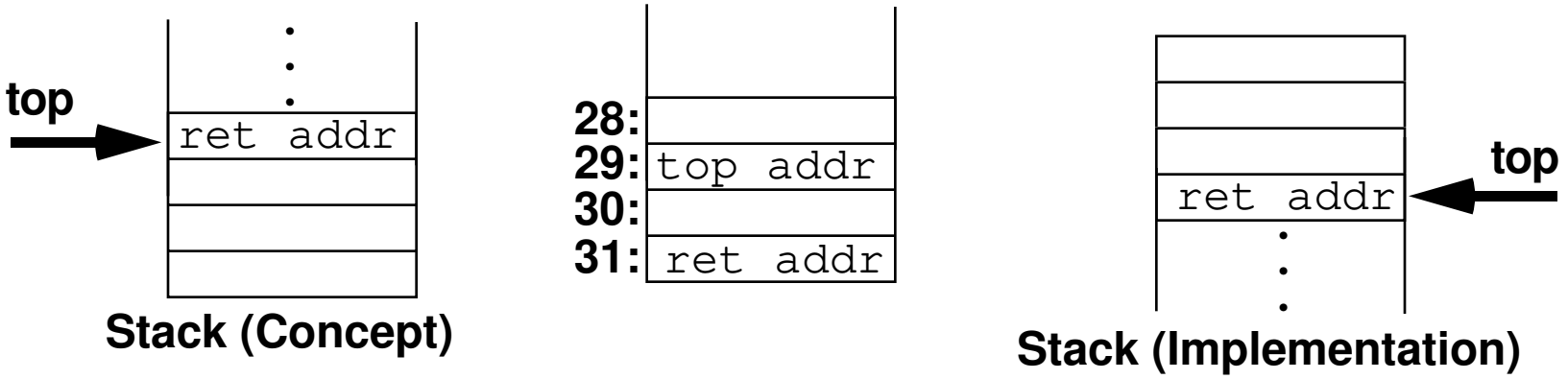
The Stack

By convention register $29 contains a pointer to the top of the stack

The stack grows "down", i.e. allocate space by subtracting from $29.

To "push" the return register $31 onto the stack . . .

```
addi   $29, $29, -4      #allocate a word on top of stack
sw     $31, 0($29)       #store return address in position
```



© 1995 LS All Rights Reserved

Register Use Conventions

Registers have agreed upon names and uses:

Name	#	Use	Name	#	Use
zero	0	constant 0	s0	16	Saved Temp†
at	1	Assembler temp	s1	17	Saved Temp†
v0	2	Exp. eval, fcn ret	s2	18	Saved Temp†
v1	3	Exp. eval, fcn ret	s3	19	Saved Temp†
a0	4	Argument 1	s4	20	Saved Temp†
a1	5	Argument 2	s5	21	Saved Temp†
a2	6	Argument 3	s6	22	Saved Temp†
a3	7	Argument 4	s7	23	Saved Temp†
t0	8	Temp, not preserved*	t8	24	Temp, not preserved*
t1	9	Temp, not preserved*	t9	25	Temp, not preserved*
t2	10	Temp, not preserved*	k0	26	OS reserved
t3	11	Temp, not preserved*	k1	27	OS reserved
t4	12	Temp, not preserved*	gp	28	Global pointer
t5	13	Temp, not preserved*	sp	29	Stack pointer
t6	14	Temp, not preserved*	fp	30	Frame pointer
t7	15	Temp, not preserved*	ra	31	Return address

* Caller saved. † Callee saved

© 1995 LS All Rights Reserved

Find this list on p. A-23

Example Procedure

Write a procedure to compute:

```
temp = arg1;
arg1 = arg2;
arg2 = temp;
```

Assume:
Temp registers are used, implies no callee saved registers to store.
No embedded calls, implies the return register need not be saved.

```
exch:  lw  $8, 0($4)     # get first argument from mem
       lw  $9, 0($5)     # get second argument from mem
       sw  $9, 0($4)     # put second arg in first's position
       sw  $8, 0($5)     # put first arg in second's position
       jr  $31           # return to caller
```


© 1995 LS All Rights Reserved

Exchange Sort

Compute
for i= 1 to n-1 do
for j= i+1 to n do
if A[i] > A[j] then exch(Ai, Aj)

Registers
$16 : A
$17 : length
$18 : index i x 4
$19 : index j x 4
$20 : addr of A[i]
$21 : addr of A[j]
$22 : val A[i]
$23 : val A[j]

```
        add  $16, $0, $4   # move A from a0
        add  $17, $5, $5   # move len from a1 and double
        add  $17, $17, $17 # double len again to get 4x
        add  $18, $0, $0   # clear index i
outr:   addi $19, $18, 4   # set j beyond i
        add  $20, $16, $18 # compute address of A[i]
        lw   $22, 0($20)   # get value of A[i]
innr:   add  $21, $16, $19 # compute address of A[j]
        lw   $23, 0($21)   # get value of A[j]
        slt  $1, $23, $22  # set if A[j] > A[i]
        beq  $1, $0, OK    # branch if no change
        sw   $22, 0($21)   # first half of exchange
        sw   $23, 0($20)   # second half of exchange
OK:     addi $19, $19, 4   # increment j
        slt  $1, $19, $17  # test if length reached
        bne  $1, $0, innr  # nope, keep going on inner
        addi $18, $18, 4   # increment i
        addi $19, $19, -4  # reduce j to n-1
        slt  $1, $18, $19  # test of length reached
        bne  $1, $0, outr  # nope, keep going on outer
```


© 1995 LS All Rights Reserved

Calling Protocol

```
sort: addi$29, $29,-32      # bump stack pointer
      sw  $16, 0($29)       # save register
      sw  $17, 4($29)       # save register
      sw  $18, 8($29)       # save register
      sw  $19,12($29)       # save register
      sw  $20,16($29)       # save register
      sw  $21,20($29)       # save register
      sw  $22,24($29)       # save register
      sw  $23,28($29)       # save register

          <procedure body>

      lw  $16, 0($29)       # restore register
      lw  $17, 4($29)       # restore register
      lw  $18, 8($29)       # restore register
      lw  $19,12($29)       # restore register
      lw  $20,16($29)       # restore register
      lw  $21,20($29)       # restore register
      lw  $22,24($29)       # restore register
      lw  $23,28($29)       # restore register
      addi$29, $29, 32      # clear stack
      jr  $31
```

Registers
$16 : A
$17 : length
$18 : index i x 4
$19 : index j x 4
$20 : addr of A[i]
$21 : addr of A[j]
$22 : val A[i]
$23 : val A[j]

© 1995 LS All Rights Reserved