Computer Architecture and Organization

INSTRUCTOR: YAN-TSUNG PENG

DEPT. OF COMPUTER SCIENCE, NCCU

Loop Statements: for

- Code

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

  sum+=i;

VAR	REG
i	$s0
sum	$s1

- Compiled MIPS code:

```
       addi $s0, $zero, 1      # $s0=1
loop: slti $t0, $s0, 11       # if($s3<11) $t0=1 else $t0=0
      beq  $t0, $zero, Exit   # if($t0==0) goto Exit
      add  $s1, $s1, $s0      # $s0=$s1+$s2
      addi $s0, $s0, 1        # $s3=$s3+1
      j    loop               # goto loop
Exit: …
```

Addressing Modes

- Register addressing (R type) - Place(get) data in(from) a register directly
  - add \$t2, \$t1, \$t0
- Immediate addressing (I type) - For instructions of arithmetic operations with a constant
  - addi \$t2, \$t1, 1 or slti \$t0, \$t1, 1 # Instructions with a constant field
  - The immediate value takes 16 bits: $-2^{15} \sim 2^{15}-1$
  - The immediate value needs sign-extension for addition
- Base addressing (I type) - Place(get) data in(from) the address computed by the base plus the relative address (RA)
  - lw \$t0, 0(\$s1)
  - Offset is also an immediate value that takes 16 bits: $-2^{15} \sim 2^{15}-1$

lw \$t0, 0 (\$s1)

RA　Base

- PC-relative addressing (I type) - For conditional branch instructions
  - beq \$0,\$3,Label #Label is also an immediate value
  - \$pc and label are added up to calculate the branch address (\$pc+4)
  - The branch address is computed by the PC plus the relative address multiplied by 4 and is then placed back to the PC
- Indirect addressing (R type)
  - jr \$s0 #The address is in a register
- Pseudo Direct addressing (J type) - For instructions with an unconditional branch
  - j Label #Label takes 26 bits, which can address 64M-word offset with \$pc as the base
  - The address is calculated by the relative address multiplied by 4, and is then placed back to the PC

Reference: https://www.cise.ufl.edu/class/cda5155fa16/protected/MIPS_addressing.pdf

Addressing Mode

I (Immediate op)

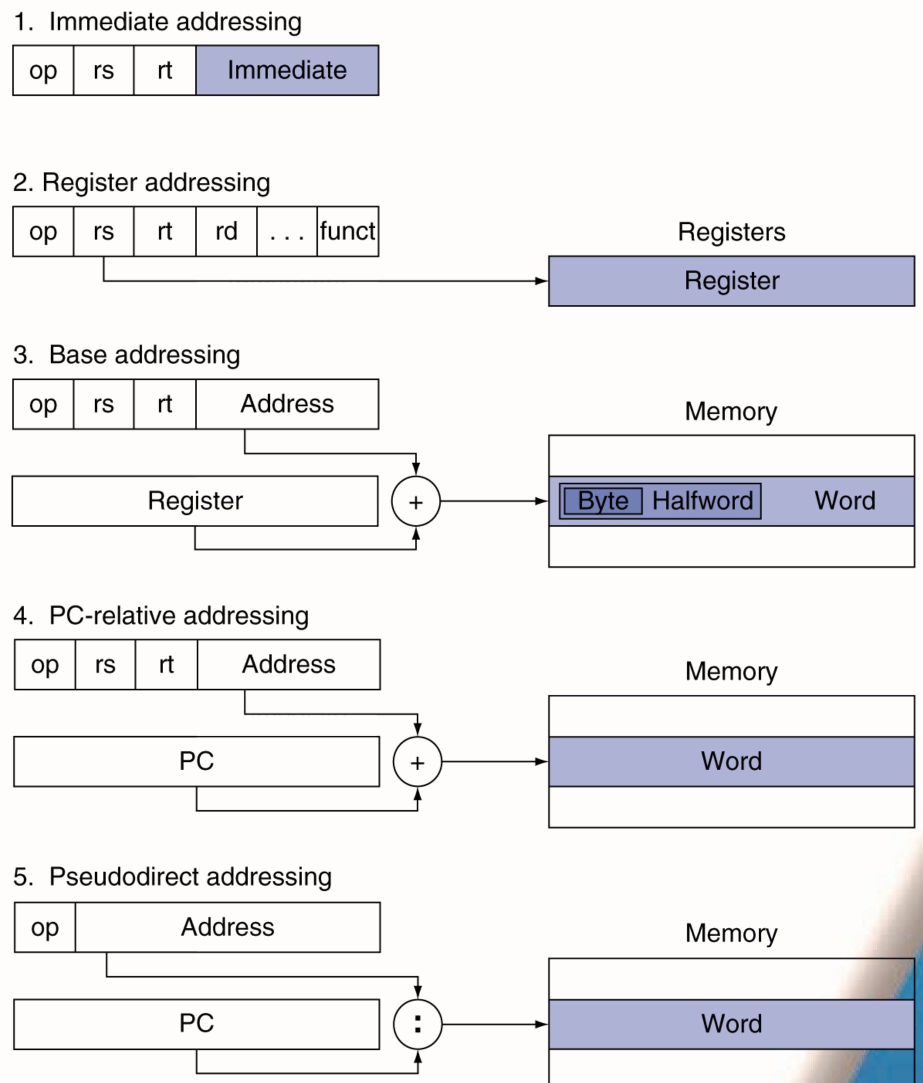
R

I (Array)

Base register

I (Branch)

J (Jump)



0000 : Address : 00

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing
  - Target address = (PC+4) + offset $\times$ 4

Conditional Branch

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

```
beq rs, rt, L1
```

PC-relative addressing

- Immediate has only 16 bits, but PC has 32 bits
  - Immediate needs to be sign-extended to fill $pc
  - Memory is word aligned, 16 bits can represents $-2^{15}+1 \sim +2^{15}$ **words** from $pc
  - If branch not taken: $pc = $pc + 4
  - If branch taken: $pc = ($pc+4) + (offset<<2)

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

- (pseudo) Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address <<2)

- 26-bit address can store 2^{26} words, which takes **28** bits. Note that the PC is **32-bit** so the PC will keep the first 4 bits the same.
- Accessing across 256 MB data

Unconditional Branch

- For conditional branches, we can only branch somewhere close to \$pc
- If we need to jump to wherever in memory we want, we would choose general jump instructions (`j` and `jal`)

wherever in memory

J-format instruction

```
j target
```

2	target
6	26

- However, target address in the instructions can only have 26 bits.
  - Only jump to word aligned addresses
    - Add two zero bits to the end of target address (26+2 bits)
    - Concatenate the 4 most significant bits from the PC with the target address to make it 32 bits
    - Cannot jump across an address boundary of 256 MB (28 bits)
      - Linker and loader can help

Target Addressing Example

Loop: sll $t1, $s3, 2	Loop: 80000	0	0	19	9	2	0
add $t1, $t1, $s6	80004	0	9	22	9	0	32
lw $t0, 0($t1)	80008	35	9	8	0		
bne $t0, $s5, Exit	80012	5	8	21	2		
addi $s3, $s3, 1	$\boxed{80016}$	8	19	19	1		
j Loop	80020	2	20000				
Exit: …	Exit: 80024						

PC relative:
(PC+4) + 2*4
=80016+8
=80024

pseudo direct, PC top 4 bits are zero

20000*4=80000

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
beq $s0,$s1, L1
```

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Address range is limited by (PC+4)+$(2^{15}-1 \sim -2^{15}) \times 4$

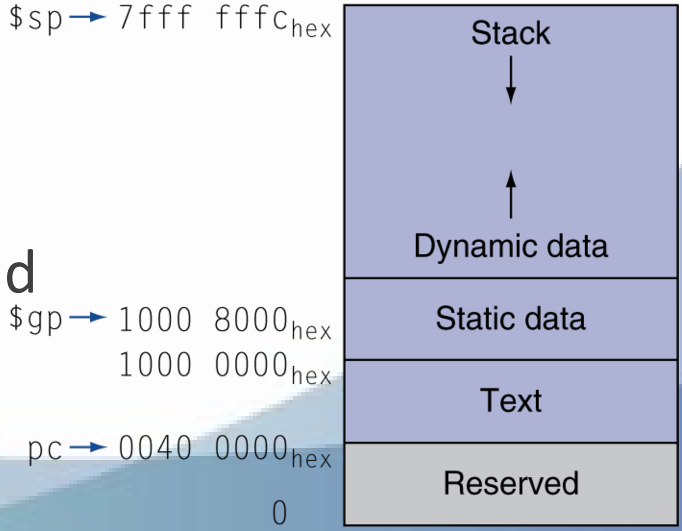
```
bne $s0,$s1, L2
j L1
L2:     …
```

op	address
6 bits	26 bits

Address range is limited by $0 \sim (2^{26}-1) \times 4$

Memory Layout

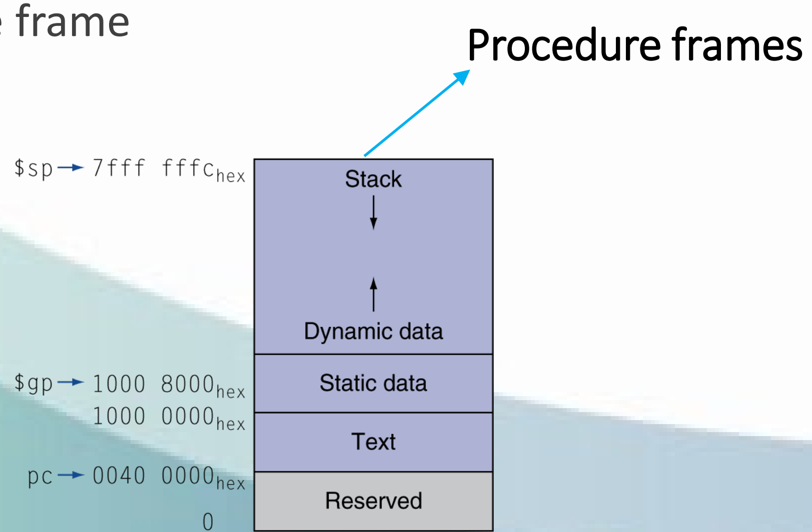
- Text segment: MIPS machine code
- Static data: global variables (C variables declared outside all procedures are considered static, as are any variables declared using the keyword *static*.)
  - e.g., static variables in C, constant arrays and strings
  - $gp, a reserved register pointed into the segment initialized for allowing ±offsets (1000 0000 – 1000 FFFF, 16-bit offsets) to access static data
- Dynamic data: heap
  - e.g., malloc in C (malloc, free)
- Stack: automatic storage
  - starts in the high end of memory and grows down.

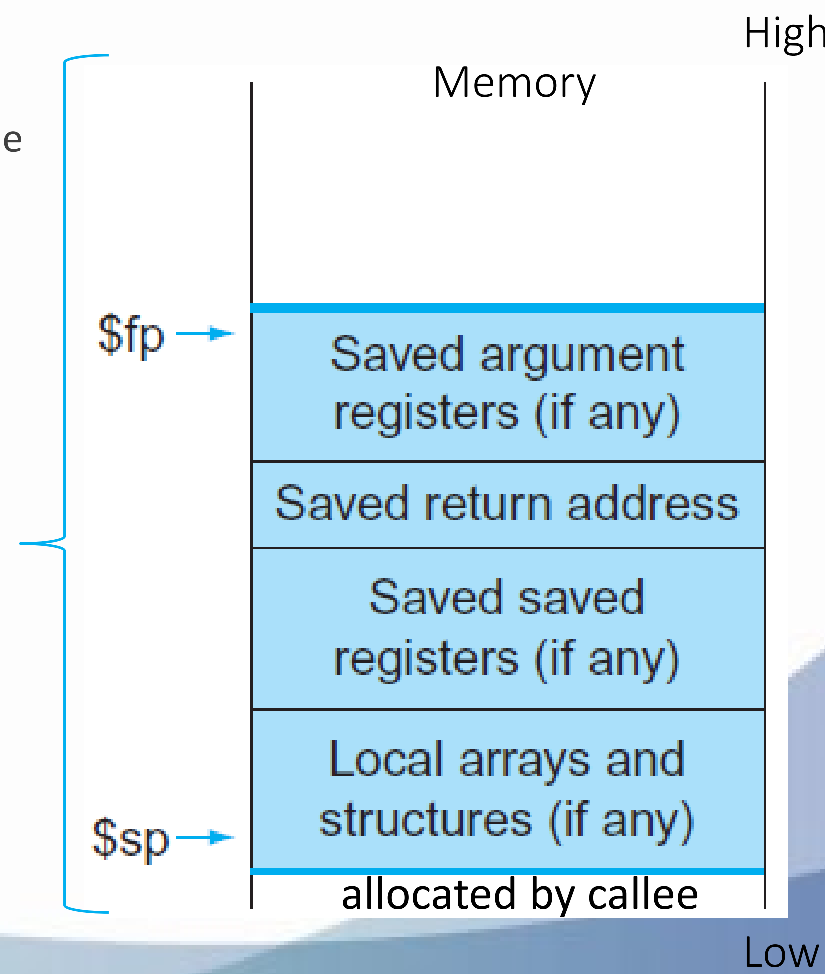


p.104 Computer Organization And Design 5th, David A. Patterson and John L. Hennessy

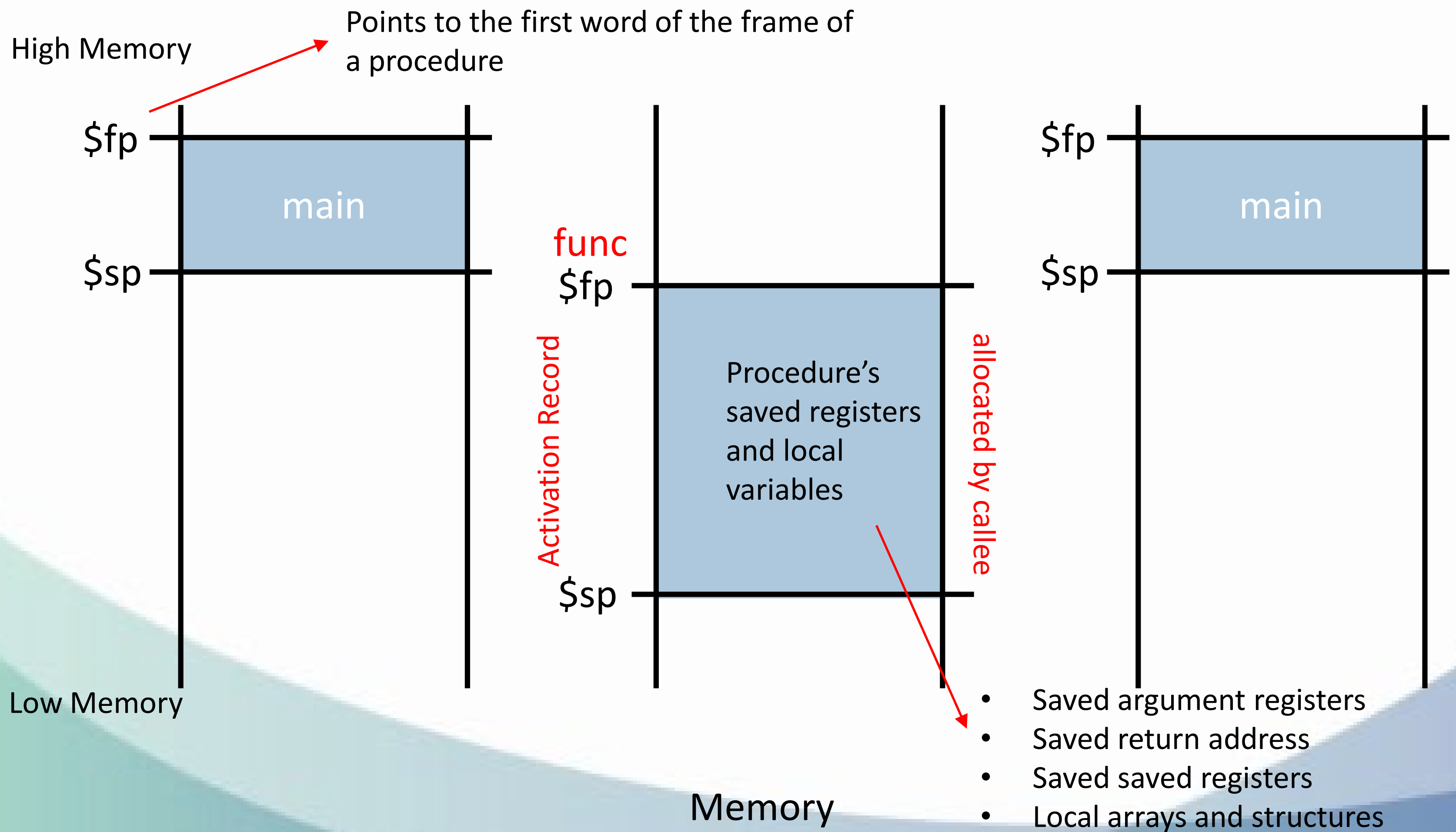
Procedure Frames

- A frame (activation record) for each function execution Used by compilers
  - Store variables that are too large to fit into the registers, Ex: local arrays, local structs
  - Store arguments
  - Store return address
  - Store saved registers
- Frame pointer (fp) points to the first word of the frame





Allocating Space on the Stack



- Saved argument registers
- Saved return address
- Saved saved registers
- Local arrays and structures

Procedure Calls

Caller:

```
void main(){
        ...
        val = func(x, y)
}
```

Passing parameters to a func through arguments
leave the caller and transfer the control to func

Callee:

```
int func(int x, int y)
{
        ...
        sum=x+y;
        return sum;
}
```

Allocate storage for func
Do the func thing
Return the result to the caller

Procedure Call Instructions

- **In the caller - Call:** Procedure call: jump and link

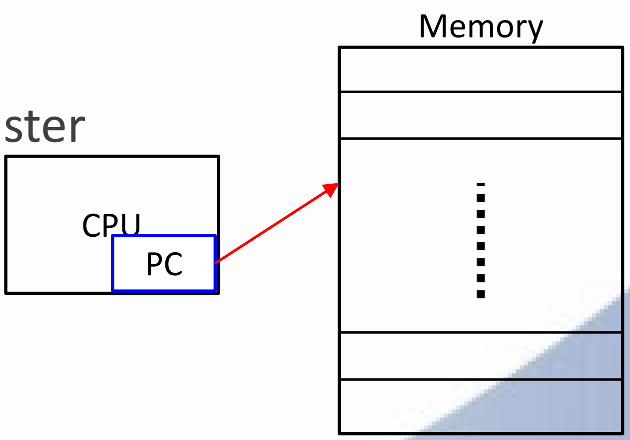
```
jal ProcedureLabel
```

  - It jumps to an address and simultaneously save the address of the next instruction in $ra
  - Link: Address of the next instruction put in $ra ($PC+4)
  - Jump: Jumps to procedure's address

- **In the callee - Return:** Procedure return: jump register

```
jr $ra
```

  - Copy $ra to PC ($PC = $ra)
  - Can also be used for computed jumps (see P.95)
    - e.g., for case/switch statements



Procedure Calling

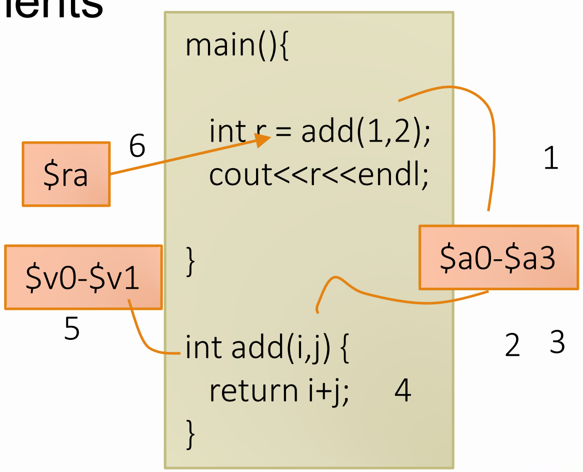
- Steps

Caller

1. Passing parameters to a func through arguments ($a0-$a3)
2. Transfer the control to the procedure
   - jal

Callee

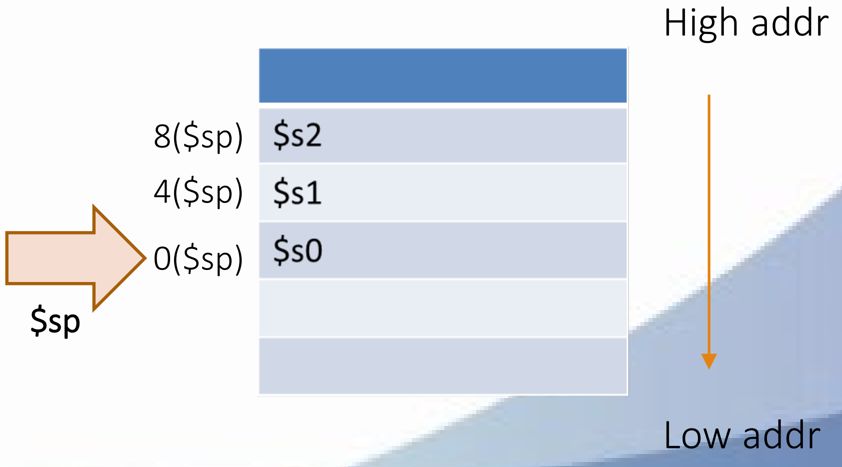
3. Allocate storage for the procedure
   - Stack frame (store saved registers in Stack)
4. Do the procedure thing
5. Place the result in register(s) ($v0-$v1) and restore saved registers
6. Return to place of call
   - j $ra



Procedure Call: Using More Registers

- Condition: \$a0-\$a3 (passed arguments) and \$v0-\$v1 (returned arguments) are not enough
- **Spilling** registers to the stack
  - push: placing the data onto the stack
  - pop: removing data from the stack
- There are two types of procedure calls
  - leaf procedures
    - Procedures that do not call others
  - Non-leaf procedures
    - Procedures that call others
    - Recursive or nested calls

Stacks grow from higher to lower addresses



MIPS Registers Conventions

Number	Name	Purpose
$0	$0	Always 0
$1	$at	The *Assembler Temporary* used by the assembler in expanding pseudo-ops. reserved for assembler
$2-$3	$v0-$v1	These registers contain the *Returned Value* of a subroutine; if the value is 1 word only $v0 is significant.
$4-$7	$a0-$a3	The *Argument* registers, these registers contain the first 4 argument values for a subroutine call.
$8-$15,$24,$25	$t0-$t9	The *Temporary Registers*. saved by Caller if used
$16-$23	$s0-$s7	The *Saved Registers*. saved by Callee if used
$26-$27	$k0-$k1	The *Kernel Reserved registers*. DO NOT USE. for OS kernel
$28	$gp	The *Globals Pointer* used for addressing static global variables. For now, ignore this.
$29	$sp	The *Stack Pointer*.
$30	$fp (or $s8)	The *Frame Pointer*, if needed (this was discussed briefly in lecture). Programs that do not use an explicit frame pointer (e.g., everything assigned in ECE314) can use register $30 as another saved register. Not recommended however.
$31	$ra	The *Return Address* in a subroutine call.

More Explanation

- $a0-$a2: arguments
  - like argc, argv, envp in C
- Register $sp (29) points to the last location in use on the stack.
- Register $fp (30) is the frame pointer.
  - points to the start of the **stack frame (**the first word of the frame of a procedure)
  - it does not change during a subroutine call
- Register $ra (31) is written with the return address for a call by the jal instruction.
- Register $gp (28) is a global pointer that points to a segment that holds static data, such as constants and global variables.

Conventions are Important

- Caller and Callee should follow the procedure conventions to have mutual benefits
  - Your code is portable and can be called by others

- Caller's Rights, Callee's Rights
  - Right to use:

<table>
<tr><th>Caller</th><th>Callee</th></tr>
<tr><td rowspan="4">$s0 - $s7</td><td>$v0, $v1</td></tr>
<tr><td>$ra</td></tr>
<tr><td>$a0, $a1, $a2, $a3</td></tr>
<tr><td>$t0 - $t9</td></tr>
</table>

Example

C code:

```
int leaf_example (int g, int h, int i, int j)
{
  int f;
  f = (g + h) - (i + j);
  return f;
}
```

Variable	Register
g	$a0
h	$a1
i	$a2
j	$a3
f	$s0
g+h	$t0
i+j	$t1
Return var	$v0

Return address	$ra
Arguments	$a0, $a1, $a2, $a3
Return value	$v0, $v1
Local variables	$s0, $s1, …
temp variables	$t0, $t1, …

Caller’s Code

- ```
  . . .
  sum = leaf_example(a,b,c,d);
  . . .
  ```
- MIPS code

```
add  $a0, $zero, $s0
add  $a1, $zero, $s1
add  $a2, $zero, $s2
add  $a3, $zero, $s3
```
set augment registers

```
jal  leaf_example
```
jump to the procedure

```
add  $s4, $0, $v0
```
get the return value
(stored in $v0 done by callee)
```

Callee

```
int leaf_example (int g,int h,int i, int j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

Variable	Register
g	$a0
h	$a1
i	$a2
j	$a3
f	$s0
g+h	$t0
i+j	$t1
Return var	$v0

Necessary??

```
leaf_example:
  addi  $sp, $sp, -12      Reserve space for 3 words
  sw    $s0, 0($sp)        Backup registers
  sw    $t0, 4($sp)
  sw    $t1, 8($sp)
  add   $t0, $a0, $a1      t0←g+h
  add   $t1, $a2, $a3      t1←i+j
  sub   $s0, $t0, $t1      s0←t0-t1
  add   $v0, $s0, $zero    Assign f to return value (v0←s0+0)
  lw    $s0, 0($sp)        Pop stack elements
  lw    $t0, 4($sp)
  lw    $t1, 8($sp)
  addi  $sp, $sp, 12       Adjust stack pointer
  jr    $ra                Return to the caller
```

Non-leaf Procedure

- Nested procedure calls
  - Any arguments and temporaries needed after the call
  - Restore all from the stack after the call
  - Callee could be the next-round caller

C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

- n in $a0
- Return result in $v0
- Return add in $ra

$a0,$v0, and $ra will be overwritten many times

Ex: Recursive Procedure

```
int fact (int n)
 {
   if (n < 1) return 1;
   else return n * fact(n - 1);
 }
```

- MIPS code:

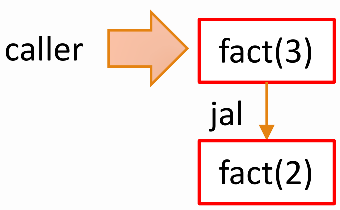
Initial action

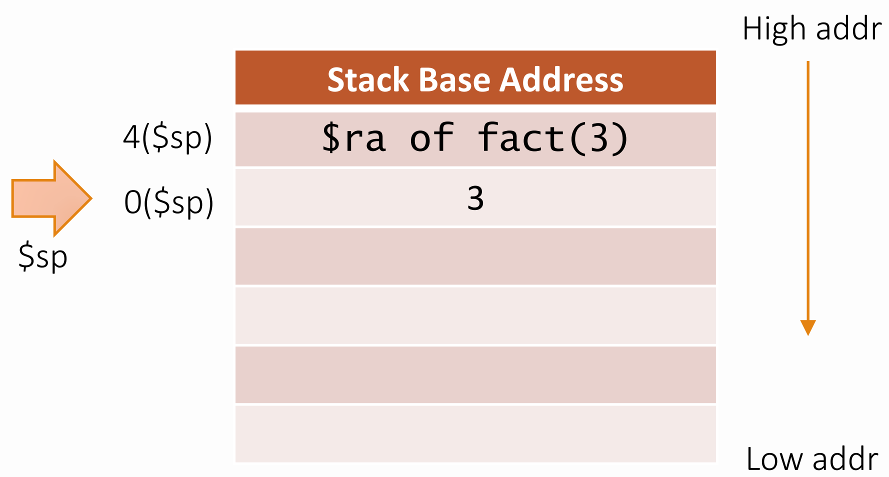
Branch op

Return

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument (n)
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, ELSE  # jump to ELSE if n ≥ 1
    addi $v0, $zero, 1     # if not, result is 1
    addi $sp, $sp, 8       #   clean stack frame
    jr   $ra               #   return
ELSE: addi $a0, $a0, -1    # else decrement n by 1
    jal  fact              # recursive call(save PC+4 in $ra)
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       # restore return address
    addi $sp, $sp, 8       # clean stack frame
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

```c
int fact (int n)
 {
   if (n < 1) return 1;
   else return n * fact(n - 1);
 }
```

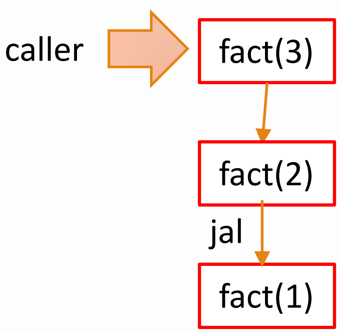


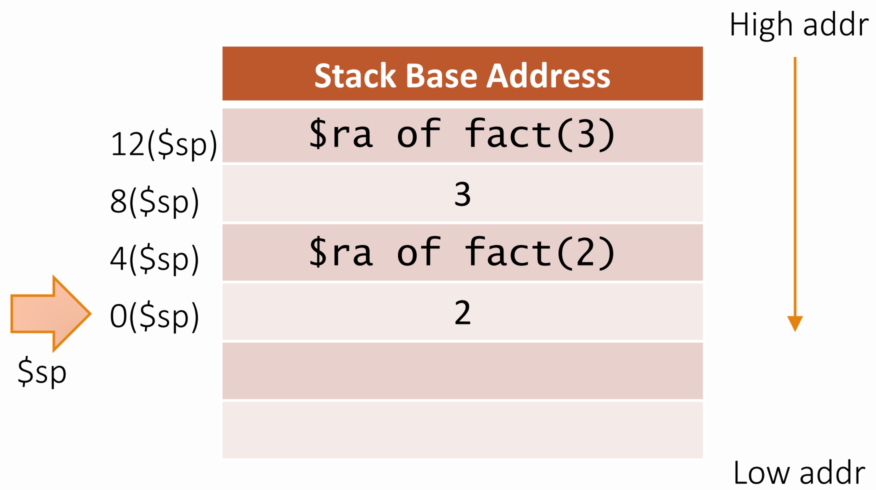


call

```
addi $sp, $sp, -8      # adjust stack for 2 items
sw   $ra, 4($sp)       # save return address
sw   $a0, 0($sp)       # save argument (n)
slti $t0, $a0, 1       # test for n < 1
beq  $t0, $zero, ELSE # jump to ELSE if n ≥ 1
…
ELSE: addi $a0, $a0, -1 # else decrement n by 1
jal  fact              # recursive call
```

```
int fact (int n)
 {
   if (n < 1) return 1;
   else return n * fact(n - 1);
 }
```

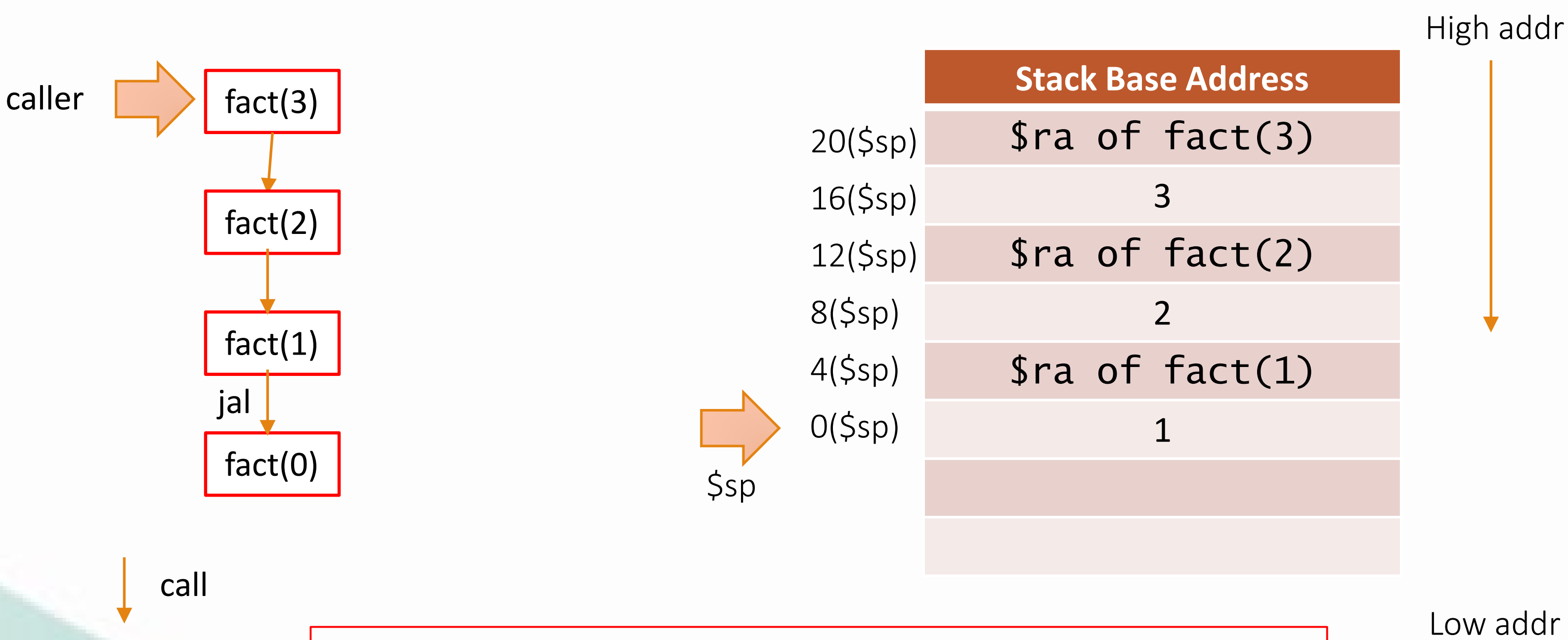




call

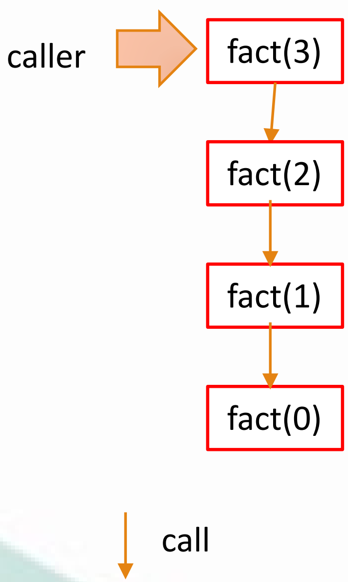
```
addi $sp, $sp, -8     # adjust stack for 2 items
sw   $ra, 4($sp)      # save return address
sw   $a0, 0($sp)      # save argument (n)
slti $t0, $a0, 1      # test for n < 1
beq  $t0, $zero, ELSE # jump to ELSE if n ≥ 1
…
ELSE: addi $a0, $a0, -1 # else decrement n by 1
jal  fact             # recursive call
```

```
int fact (int n)
 {
   if (n < 1) return 1;
   else return n * fact(n - 1);
 }
```




```
addi $sp, $sp, -8     # adjust stack for 2 items
sw   $ra, 4($sp)      # save return address
sw   $a0, 0($sp)      # save argument (n)
slti $t0, $a0, 1      # test for n < 1
beq  $t0, $zero, ELSE # jump to ELSE if n ≥ 1
…
ELSE: addi $a0, $a0, -1 # else decrement n by 1
jal  fact             # recursive call
```

```
int fact (int n)
 {
   if (n < 1) return 1;
   else return n * fact(n - 1);
 }
```



High addr

	Stack Base Address
28($sp)	$ra of fact(3)
24($sp)	3
20($sp)	$ra of fact(2)
16($sp)	2
12($sp)	$ra of fact(1)
8($sp)	1
4($sp)	$ra of fact(0)
0($sp)	0

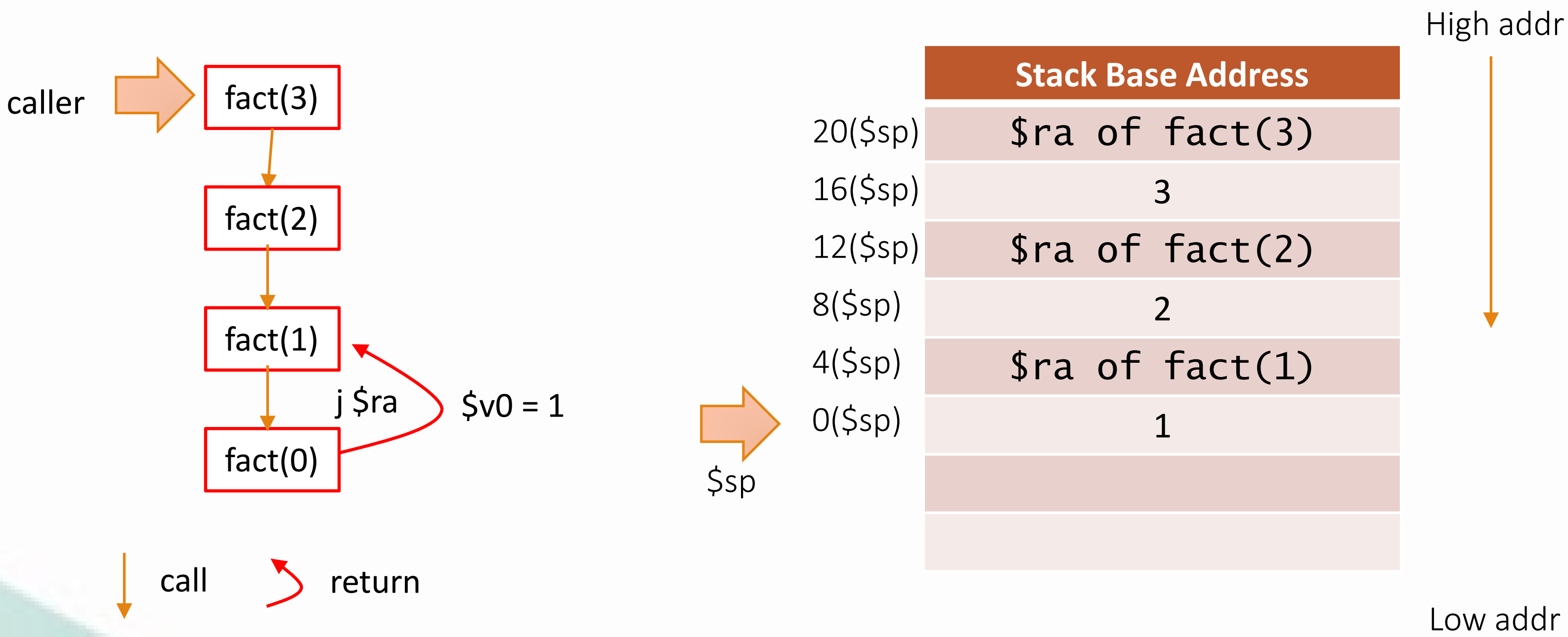
$sp

Low addr

Not yet return

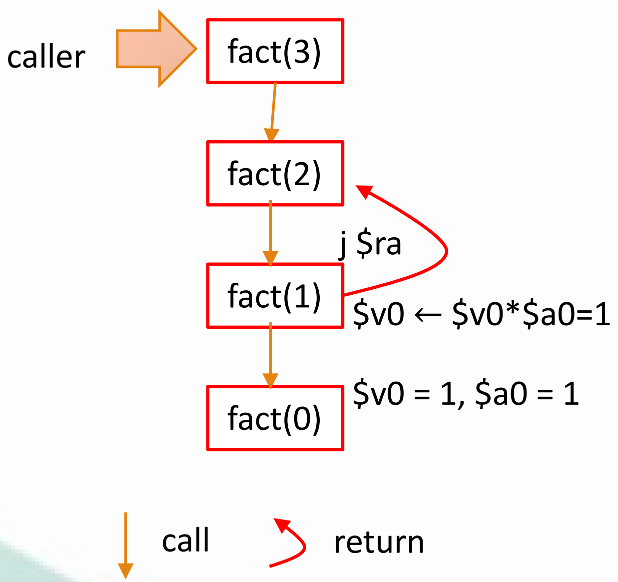
```
addi $sp, $sp, -8     # adjust stack for 2 items
sw   $ra, 4($sp)      # save return address
sw   $a0, 0($sp)      # save argument (n)
slti $t0, $a0, 1      # test for n < 1
beq  $t0, $zero, ELSE # jump to ELSE if n ≥ 1
addi $v0, $zero, 1    # if not, result is 1
```

```
int fact (int n)
 {
   if (n < 1) return 1;
   else return n * fact(n - 1);
 }
```




```
addi  $sp, $sp, 8        #   clean stack frame
jr    $ra                #   return
```

```
int fact (int n)
  {
    if (n < 1) return 1;
    else return n * fact(n - 1);
  }
```



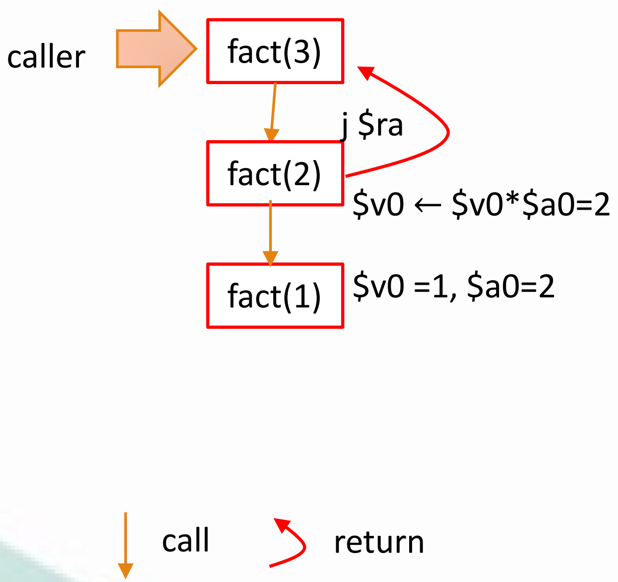
High addr

	Stack Base Address
12($sp)	$ra of fact(3)
8($sp)	3
4($sp)	$ra of fact(2)
$sp → 0($sp)	2
	~~$ra of fact(1)~~
	~~1~~

Low addr

```
lw   $a0, 0($sp)       # restore original n
lw   $ra, 4($sp)       # restore return address
addi $sp, $sp, 8       # clean stack frame
mul  $v0, $a0, $v0     # multiply to get result
jr   $ra               # and return
```

```
int fact (int n)
 {
   if (n < 1) return 1;
   else return n * fact(n - 1);
 }
```



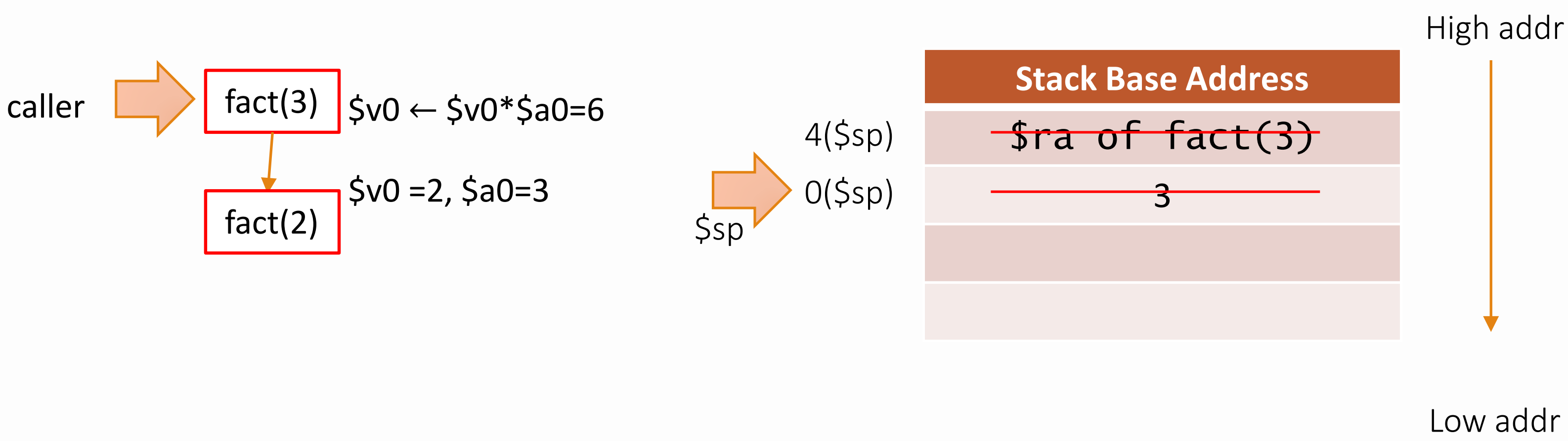
High addr

	Stack Base Address
12($sp)	$ra of fact(3)
8($sp)	3
4($sp)	~~$ra of fact(2)~~
$sp → 0($sp)	~~2~~

Low addr

```
lw   $a0, 0($sp)      # restore original n
lw   $ra, 4($sp)      # restore return address
addi $sp, $sp, 8      # clean stack frame
mul  $v0, $a0, $v0    # multiply to get result
jr   $ra              # and return
```

```c
int fact (int n)
  {
    if (n < 1) return 1;
    else return n * fact(n - 1);
  }
```




```
lw   $a0, 0($sp)      # restore original n
lw   $ra, 4($sp)      # restore return address
addi $sp, $sp, 8      # clean stack frame
mul  $v0, $a0, $v0    # multiply to get result
jr   $ra              # and return
```

Handling Byte/Halfword

- MIPS provides byte/halfword load/store
  - Ex: string processing

```
lb rt, offset(rs)     lh rt, offset(rs)      #Sign extend to 32 bits in rt
lbu rt, offset(rs)    lhu rt, offset(rs)     #Zero extend to 32 bits in rt
sb rt, offset(rs)     sh rt, offset(rs)      #Store just rightmost byte/halfword
```

Ex: String Processing

```
void strcpy (char x[], char y[])
{
  int i = 0;
  while ((x[i]=y[i])!='\0')
    i++;
}
```

Var	Reg
x	\$a0
y	\$a1
i	\$s0

- MIPS code:

```
strcpy:
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw   $s0, 0($sp)       # save $s0
    add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1     # addr of y[i] in $t1
    lbu  $t2, 0($t1)       # $t2 = y[i]
    add  $t3, $s0, $a0     # addr of x[i] in $t3
    sb   $t2, 0($t3)       # x[i] = y[i]
    beq  $t2, $zero, L2    # exit loop if y[i] == 0
    addi $s0, $s0, 1       # i = i + 1
    j    L1                # next iteration of loop
L2: lw   $s0, 0($sp)       # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr   $ra               # and return
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- If a 32-bit constant is needed, ex: $65546_{10} = 0001000A_{16}$
  - lui $s0, constant
    - Copies a 16-bit constant to left 16 bits of $s0
    - Clears right 16 bits of $s0 to 0
  - ori $s0, $s0, 10
    - Write constant on the right 16 bits of $s0

`lui $s0, 1`	0000 0000 0000 0001 0000 0000 0000 0000
`ori $s0, $s0, 10`	0000 0000 0000 0001 0000 0000 0000 1010

Pseudo-instructions

- Pseudo instructions:
  - Not real instructions
  - Re-interpreted by Assembler

```
move $t0, $t1          →   add $t0, $zero, $t1

blt $t0, $t1, L        →   slt $at, $t0, $t1
                           bne $at, $zero, L
```

  - $at (register 1): reserved to use by assembler as a temporary register

- a list of the standard MIPS pseudo-instructions
  - blt, bgt, ble, blt, bge, li, move, ...