Road Map

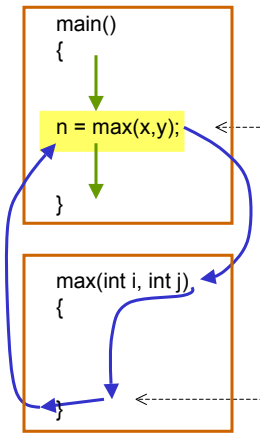
- C functions
  - Computation flow
  - Implementation using MIPS instructions
- Useful new instructions
- Addressing modes
- Stack data structure

EE 361
University of Hawaii

Implementation of C functions and MIPS machine language

[MIPSb Notes]

C Functions



Caller function/subroutine

Subroutine call

Callee function/subroutine

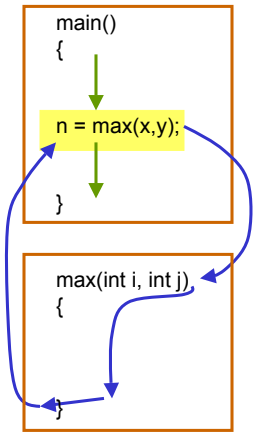
Return from subroutine

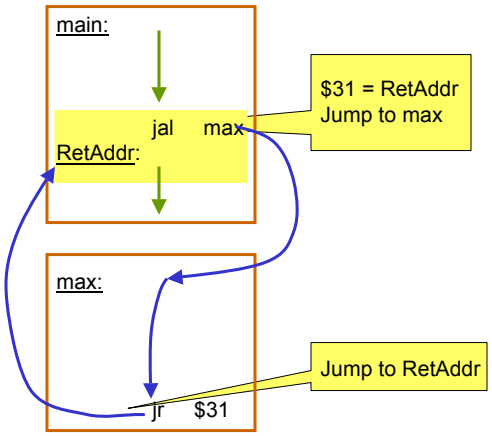
What do we need?

1. Jump to max
2. Store *return address*

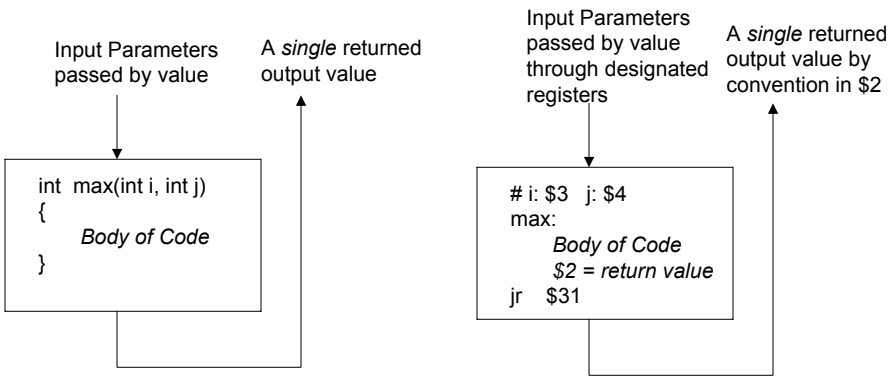
3. Jump to *return address*

Implementation





Implementation



Example

```
main()
{

n = max(x,y);

}
```

```
main:

#  Calling procedure, where x: $20.  y: $21.  n: $22
move    $3,$20    # Load input parameters
move    $4,$21
jal     max       # Call subroutine
move    $22,$2    # Load return value into n
```

```
max(int i, int j)
{ if (i < j) return j;
  else       return i;
}
```

```
# Input parameters    i: $3.   j: $4
max:
        slt     $1,$3,$4
        beq     $0,$1,Skip
        move    $2,$4
        jr      $31
Skip:   move    $2,$3
        jr      $31
```

jump-and-link (jal)

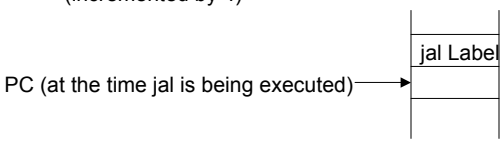
jal	Label	$31 = return address jump to Label	J-Type

How does CPU compute return address? Return address can be found in program counter

CPU

$0, $1, ..., $31
Program Counter (PC)

Program Counter (PC) Register: 32-bits
Holds address of next instruction to execute
It is usually incremented after every instruction
(incremented by 4)



New Instructions: Immediate Addressing

Remember: add, sub, slt

There's also:

```
addi    $2,$3,1000      #  $2 = $3 + 1000  (a constant)
subi    $2,$3,100       #  $2 = $3 - 1000 [This is a pseudo instruction like mov]
slti    $2,$3,100       #  $2 = 1 if $3 < 100
                        #       0  otherwise
```

I-type instruction

"i" means *immediate addressing*, which will be discussed a little later

Immediate Addressing

Differences with old instructions

add $2,$3,$4

R-Type

0	3	4	2	0	32

$4 is a register where operand is located

addi $2,$3,4

I-Type

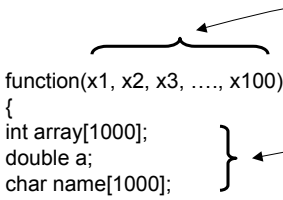
8	3	2	4

4 is the operand

Example

```
main():
main()
{
k = nonneg(x);          ret1:   move  $3,$20    #  $3 = x
                                jal   nonneg    # call nonneg
                                move  $22,$2    #  k = nonneg()

k += nonneg(y+5);               addi  $3,$21,5  # $3 = y+5
                                jal   nonneg    # call nonneg
                        ret2:   add   $22,$22,$2  # k += nonneg()
}
nonneg(int  i)          nonneg:
{                               slt   $1,$3,$0  # $1= 1 if i < 0
   if (i < 0) return 0;         beq   $1,$0,Else
   else       return i;         move  $2,$0     # return 0
                                jr    $31
}                       Else    move  $2,$3     # return i
                                jr    $31
```

Temporary Storage



```
function(x1, x2, x3, …., x100)
{
int array[1000];
double a;
char name[1000];
.
.
.

}
```

The function needs memory cells for
passing input parameters
local variables

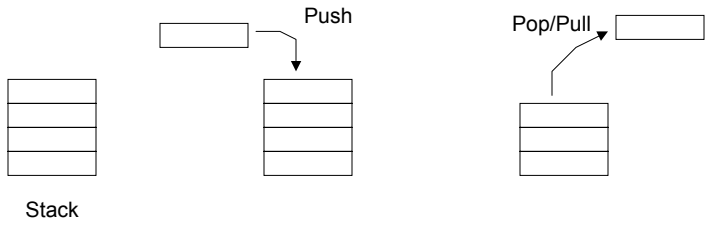
The storage is *temporary*, needed only during the "life" of the function.

Registers $1-$31 offer only a limited amount of storage.

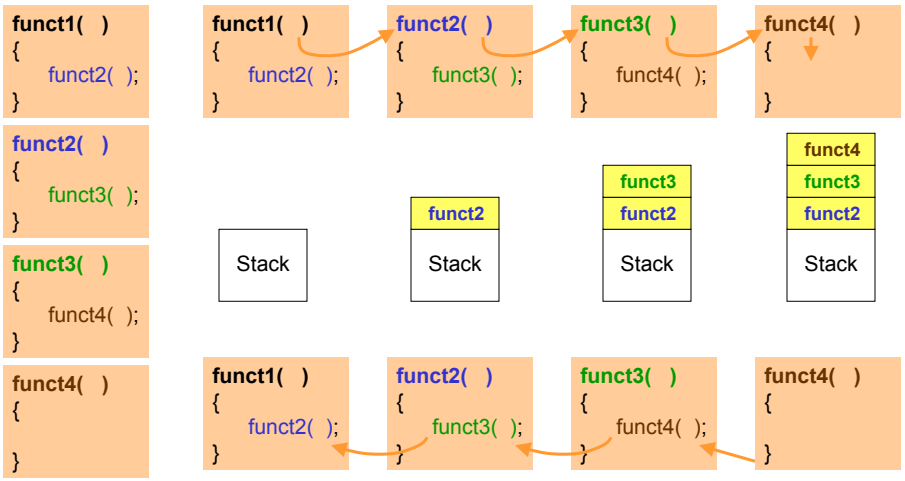
We can use main memory but can we organize it so that management of the resources is simple and efficient? Stacks

Stack

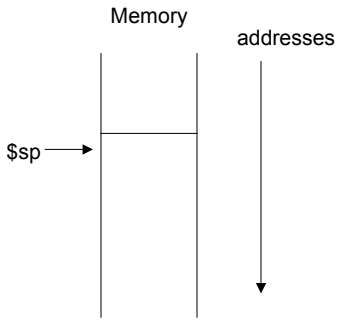
Stack: A data structure that changes in size dynamically
It has a "top"
Push things in from the "top"
Pull things out from the "top"



Motivation Example



Implementing Stacks



$sp = Stack Pointer
It can be most any register
except special registers e.g., $0 and $31.
By the MIPS convention $sp = $29

Reading from top of stack:

```
lw      $2,0($sp)
```

Writing to top of stack:

```
sw      $2,0($sp)
```

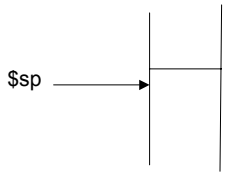
Implementing Push/Pop

Push: Example of pushing $10 on stack

```
add     $sp,$sp,-4      #  make space on top of stack for new value
sw      $10,0($sp)              #  store $10 on the stack
```

Pop: Example of popping stack and putting value in $12

```
lw      $12,0($sp)      #  load the top of stack into $12
add     $sp,$sp,+4      #  update stack pointer
```



Example

C statement

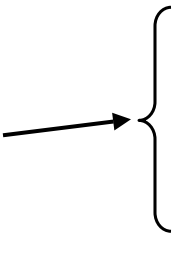
n = function(x)

Assembly language

```
# n: $10.  x: $11.
# Pass parameters through $3

add     $sp,$sp,-4      # Store old value of $3 away  just in case
sw      $3,0($sp)       #    it's being used by something else

move    $3,$11          # Move x in $3
jal     function
move    $10,$2          # n = function(x)

lw      $3,0($sp)       # Restore old value of  $3
add     $sp,$sp,4
```

Example

```
function1()
{

function2();

}

function2()
{

function3()

}

function3()
{
}
```



```
function2:
        add     $sp,$sp,-4
        sw      $31,0($sp)

        jal     function3

        lw      $31,0($sp)
        add     $sp,$sp,4
```


Good Housekeeping

If you use registers, make sure that either

it's not used by anything else

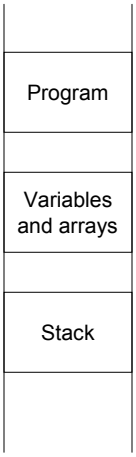
OR

if it's used, store it's contents away (say, into stack), and restore after you're done

Stack housekeeping: *keeping the stack balanced*

If you allocate (or push) space on the stack then remove all of it when you're done

Where do we store stacks? Memory Organization



Addressing Modes

Instructions have two characteristics:

operation: what it does, e.g., add, subtract, set-less-than, load word, store word

addressing modes: how to get operands.

The mode specifies where the operands are located

Some of the types:

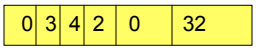
Register	Specifies register where operand is located
Immediate	Operand is in the instruction
Base or Displacement	Example: lw and sw
Program Counter (PC) Relative	Example: beq and bne
Direct	Address of operand is in the instruction

Instruction size is limited to 32 bits. Addresses stored in instructions will be less than 32 bits, often 16 bits. However, addresses for word-operands are 32 bits. How do we get a 32 bit address from a smaller number of bits? Fill in the other bits. Sign extension

Addressing Modes

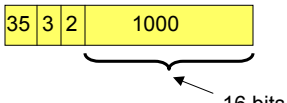
Register Addressing: register (number) is specified

add $2,$3,$4



Base or Displacement Addressing: location of operand is identified with a register and a constant (displacement)

lw $2,1000($3)



Thus, address of operand is computed by $3 + 1000

Expanded to 32 bits

Addressing Modes

Immediate Addressing: operand is in the instruction

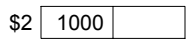
Example: addi $2,$3,1000

For practical benchmarks, over half of the arithmetic instructions have constant operands

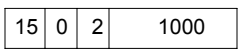
Principle: Make the common fast. So make immediate addressing fast.

Most constants are 16 bits, but sometimes you may get a BIG ONE.

lui $2,1000 load-upper-immediate loads into register $3 the value 1000, but in the upper 16 bits

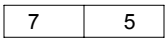


I-Type Instruction

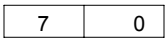


Addressing Modes

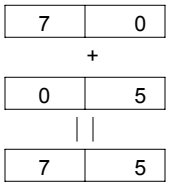
Loading a Big Constant: $3



lui $3,7



addi $3,$3,5

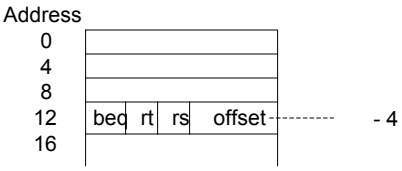


PC-Relative Addressing

PC-Relative Addressing: address = PC + offset (specified in instruction)

Example: bne $1,$2,Label What's stored in machine instruction is an offset

```
Loop:   instruction
        instruction
        instruction
        beq  $3,$4,Loop
        instruction
```



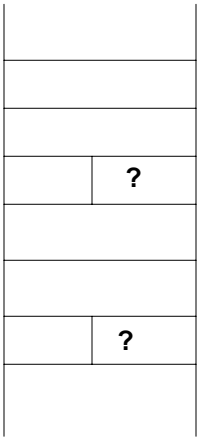
PC + address offset = Loop

address offset = Loop - PC
= 0 - 16
= - 16

BUT notice address offsets are always multiples of 4, so the **Actual Offset** in the machine code is the address offset/4

Example

```
#  Clear array A[100]
        move    $20,$0
        addi    $21,$0,100
Loop:   beq     $0,$21,Skip
        sw      $0,Astart($20)
        addi    $21,$21,-1
        beq     $0,$0,Loop
Skip:
```



Offsets

beq $2,$3,Label has a 16-bit offset

16-bit offset can be positive or negative

represented using twos complement

Twos Complement Brief Overview:

Representation of signed integers with bits
high order bit indicates sign
1 means negative
0 means positive

Examples 10001 is negative
01101 is positive

PC-Relative Addressing:

32-bit PC + 16-bit offset

Sign Extension

To make a 16-bit twos complement number into a 32-bit twos complement number just add more bits to the left.

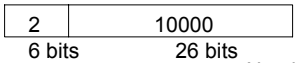
add 1s if number is negative
add 0s if number is positive

Comparison of Jumping and Conditional Branching

Jump and Jump-and-Link: J-type

Direct addressing

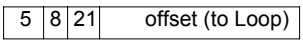
j 10000



Number is used directly as address (plus a type of extension)

Conditional Branch: I-type

bne $3,$21,Loop



Range is better for J-type because there are more bits

16 bit offset is sufficient for most instances of conditional branch

What About Far Away Branches?

```
beq     $18,$19,Loop
```

Compiler figures out that the offset is greater than 16 bits

```
        bne     $18,$19,Skip
        j       Loop
Skip:
```

Review

- Stacks
- Addressing
- Computation of addresses and offsets
  - Base or displacement addressing
  - PC-Relative addressing
  - How to compute offsets for MIPS
- MIPS instructions
  - add, sub, slt, addi, subi, slti, lui, lw, sw, beq, bne, j, jr, jal, mul. 15 instructions out of 32 in the back cover of the text. Actually, there's just a few different types of instructions left.