Code generation for SPIM

Topics:

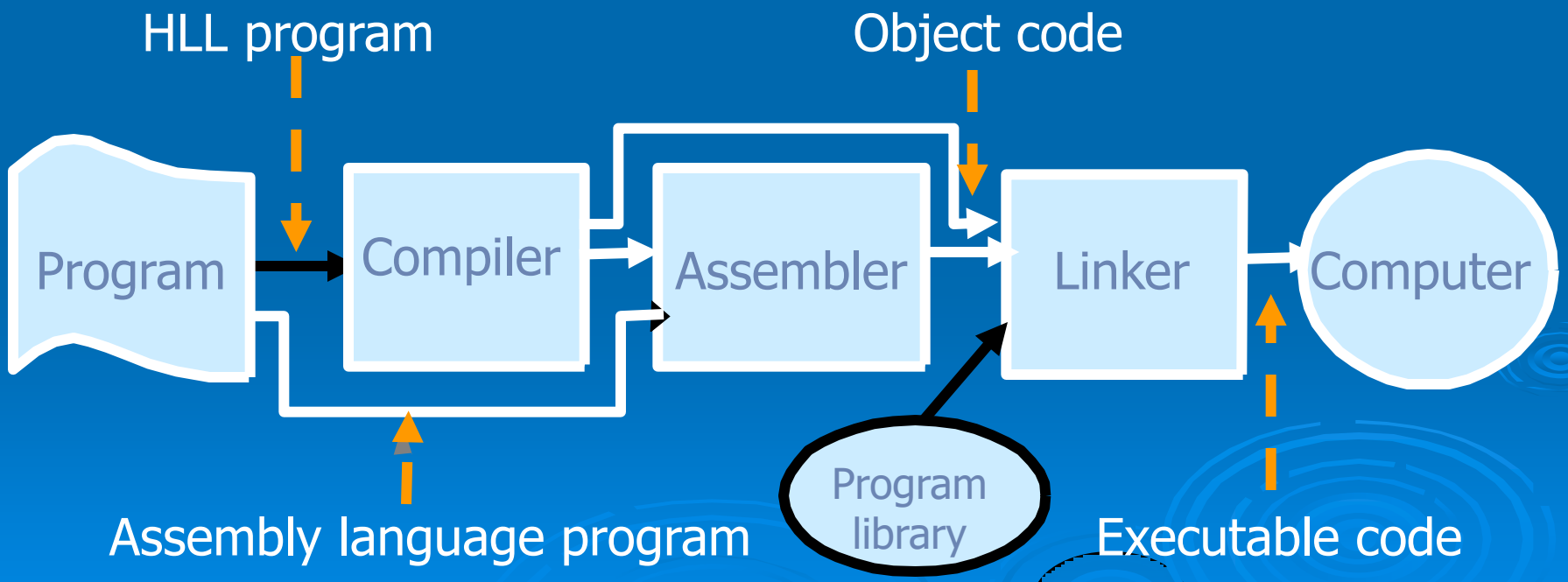
- Assembly language, assemblers
- MIPS R2000 Assembly language
  - Instruction set
  - MIPS design goals
  - Memory & registers
  - Instruction formats
  - Some MIPS instructions
- Advanced topics
  - Macros
  - Procedure calls
  - I/O

Introduction

- Instruction set:
  - The complete set of instructions (vocabulary) used by a machine
- Instruction Set Architecture (ISA):
  - An abstract interface between the hardware and the lowest-level software of a machine
  - Includes:
    - Necessary information to write correct machine-language programs
    - Specification of instructions, registers, memory size, ...etc.
- We will concentrate on MIPS- ISA
  - Used by NEC, Nintendo, Silicon Graphics, Sony, . . .

High-Level Language (HLL) Translation

- Compilers generate either machine language or assembly language object files



Assembly Language

- Symbolic representation of the machine language of a specific processor
- Advantages
  - High execution speed
  - Smaller code size
- Disadvantages:
  - Machine specific
  - Long programs
  - Less programmer productivity
  - Difficult to read, understand, & debug
  - Lacks structure

Assemblers

- Converts assembly language into machine code
- Input:
  - Assembly language program
- Output:
  - Object file containing
    - Non-executable machine instructions
    - Data
    - Bookkeeping info
- Two phases:
  - Get locations of labels and build the symbol table
  - Translate statements into equivalent binary code
- Symbol Table
  - Used to help resolve forward & external referencing to create the object file

Translation of a C-Program Into Assembly

- Example: C-Program

```
#include <stdio.h>
int main (int argc, char *argv[])
{   int i;
    int         sum = 0;
    for (i=0; i<= 100; i=i+1)
      sum = sum +i*i;
    printf("The sum from 0 .. 100 is %d\n", sum);
}
```

Equivalent Assembly Program (No Labels)

```
addiu       $29,$29, -32                    #add immediate unsigned
sw    $31,20($29)                  # $29 = $sp stack pointer
sw    $4,32($29)           # $31 = $ra return address
sw    $5,36($29)           #
sw    $0,24($29)           # Store relevant values
sw    $0,28($29)           # onto stack
lw          $14,28($29)                     #
lw          $24,24($29)                     #
multu       $14,$14                         # multiply unsigned
addiu       $8,$14,1
slti  $1,$8,101            # set $1 if < immediate
sw    $8,28($29)
mflo  $15                        # move from lo of register
addu $25,$24,$15
bne         $1,$0,-9
sw    $25,24($29)
lui         $4,4096                         # load upper immediate
lw          $5,24($29)
jal         1048812                         # jump & link
addiu       $4,$4,1072
lw          $31,20($29)
addiu       $29,$29,32
jr          $31                             # jump register
move        $2, $0
```

Equivalent Assembly Program (Labeled)

```
        .text
        .align      2
        .globl      main

main:
        subu        $sp,$sp, 32  # increment stack by a stack frame
        sw          $ra,20($sp)  #Save return address
        sd          $a0,32($sp)  # pseudo-instruction (Save double-word)
        sw          $0,24($sp)
        sw          $0,28(sp)
loop:
        lw          $t6,28($sp)
        mul         $t7,$t6,$t6
        lw          $t8,24($sp)
        addu        $t9,$%8,$t7
        sw          $t9,24($sp)
        addu        $t0,$t6,1
        sw          $t0,28($sp)
        ble         $t0,100,loop
        la          $a0,str                 # Pseudo-instruction (Load address)
        lw          $a1,24($sp)
        jal         printf                  # Jump & link
        move        $v0,$0
        lw          $ra,20($sp)
        addu        $sp,$sp,32
        jr          $ra

        .data
        .align0                             # Turn off automatic alignment
str:
        .asciiz "The sum from 0..100 is %d\n"
```

Instruction Design

- Instruction length:
  - Variable length instructions:
    - Assembler needs to keep track of all instruction sizes to determine the position of the next instruction
  - Fixed length instructions:
    - Require less housekeeping
- Number of operands
  - Depend on type of instruction

SPIM

- Software simulator for running MIPS R-Series processors' programs
- Why use a simulator?
  - MIPS workstations
    - Not always available
    - Difficult to understand & program
  - Simulator
    - Better programming environment
    - Provide more features
    - Easily modified

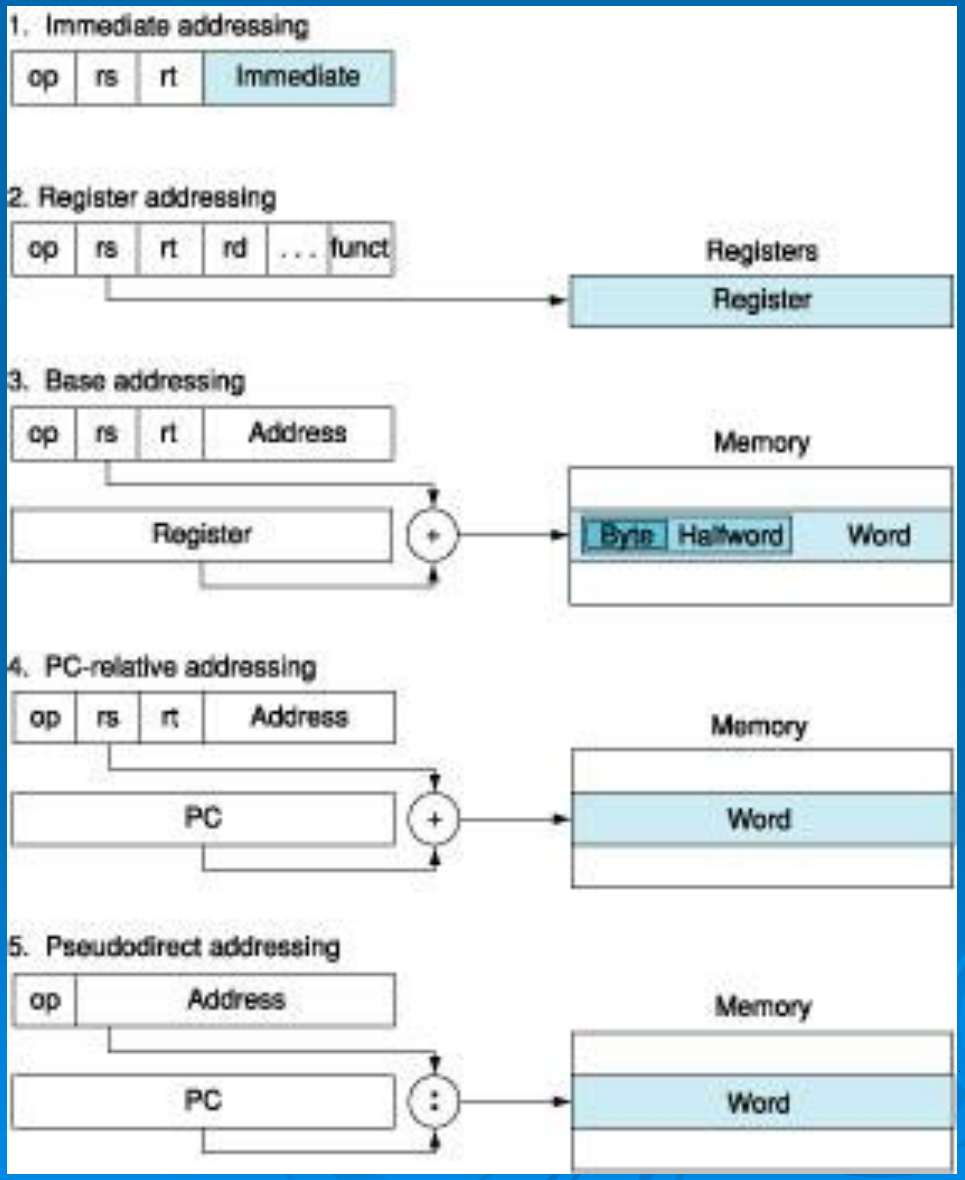
MIPS Processors

- Addressing modes:
  - Describe the manner in which addresses for memory accesses are constructed
  - MIPS is a “Load-Store” architecture
    - Only load/store instructions access memory
  - Data should be aligned (usually multiple of 4 bytes)
  - More details in MIPS Quick Reference
  - https://imagination-technologies-cloudfront-assets.s3.amazonaws.com/documentation/MD00565-2B-MIPS32-QRC-01.01.pdf

Addressing Modes

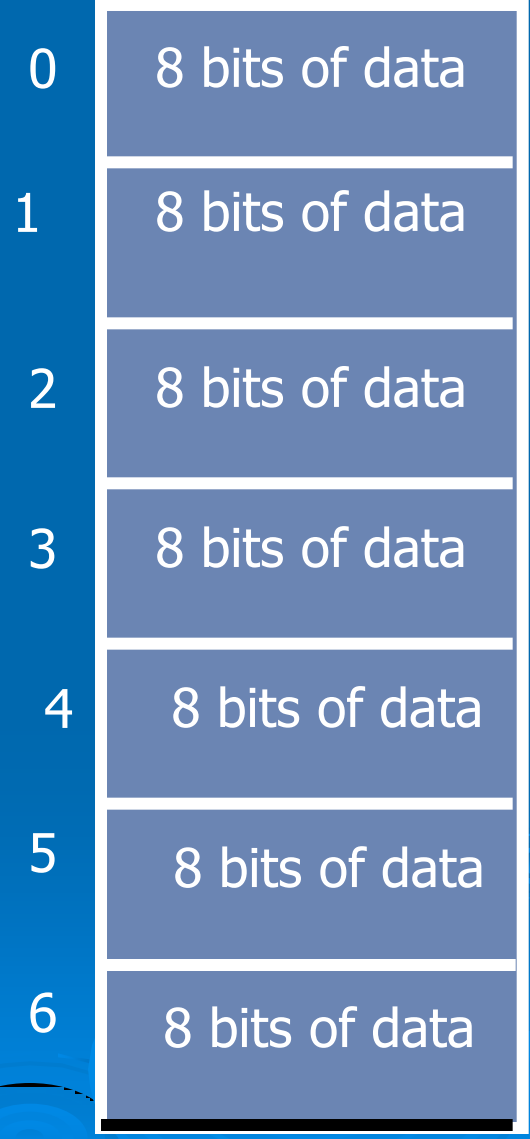
- Register addressing
  - Operand is a register
  - Value is the contents of the register
- Base or displacement addressing
  - Operand is at the memory location whose address is the sum of a register and a constant in the instruction
- Immediate addressing
  - Operand is a constant within the instruction itself
  - Immediate
- PC-relative addressing
  - Address is the sum of the PC and a constant in the instruction
- Pseudo-direct addressing
  - Jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

Addressing Modes



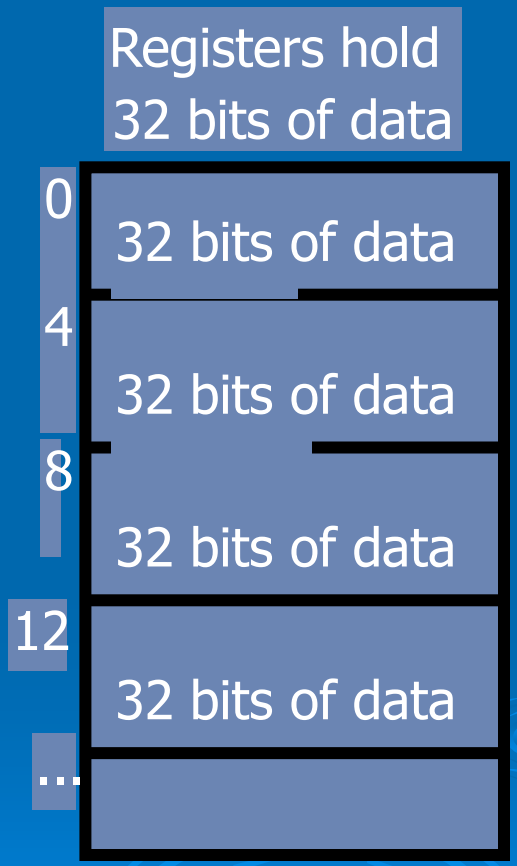
Memory Organization

- Memory is viewed as a large, single-dimensional array
- To access a word, memory address is supplied by instruction
- Memory address is an index to the array, starting at 0



Byte & Word Addressing

- Index points to a byte of memory
- Most data items use "words"
- Words are aligned at word boundaries
  - For MIPS, a word is 32 or 64 bits
  - They are usually called MIPS =32 & MIPS =64 respectively
  - We will only consider MIPS32
  - MIPS32 (4 bytes) can access
    - 2^{32} bytes with byte addresses from 0 to 2^{32}-1, or
    - 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4

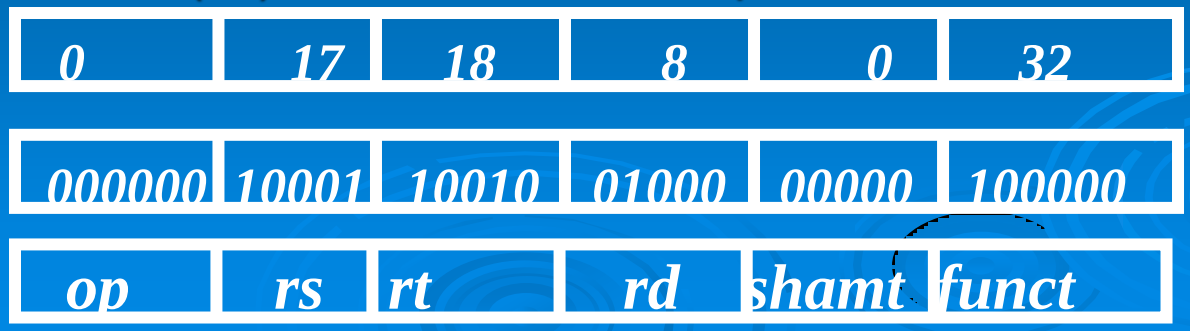


MIPS Instruction Formats

- MIPS has 3 instruction formats
  - R-type (Register) format
  - J-type (Jump) format
  - I-type (Immediate) format
- All MIPS instruction formats are 32 bits long
  - Example: ***add $t0, $s1, $s2***
- Registers can be written in their symbolic or numeric forms
  - ***$t0=8, $s1=17, $s2=18***

R-Format (Register) Instructions

- ***op:*** Operation code (6-bits)
- ***rs;*** 1st source register (5-bits)
- ***rt:*** 2nd source register (5-bits)
- ***rd:*** Destination register (5-bits)
- ***shamt:*** Shift amount (5-bits)
- ***funct:*** Function code (6-bits)
  - The first (***op***) & last fields (***funct***), combined, indicate the type of instruction
  - Second (***rs***) & third (***rt***) fields are the source operands
  - Fourth field (***rd***) is the destination operand



Registers Names & Numbers

Name	Register#	Usage	Preserved on call?
$zero	0	the constant value 0	n.a.
$v0-$v1	2-3	values for results & expr. evaluation	no
$a0-$a3	4-7	arguments	yes
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved	yes
$t8-$t9	24-25	more temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

SPIM

- Software simulator for running MIPS R-Series processors' programs
- SPIM Simulator simulates most of the functions of three MIPS processors
- More about SPIM will be discussed in the labs
- Why use a simulator?
  - MIPS workstations
    - Not always available
    - Difficult to understand & program
  - Simulator
    - Better programming environment
    - Provide more features
    - Easily modified

MIPS Design

- Goals
  - Maximize performance
  - Minimize cost
  - Reduce design time
- How can we reach these goals?
  - Principles
    1. Simplicity favors regularity
    2. Smaller is faster
    3. Good design demands good compromises
    4. Make common case fast

Simplicity Favors Regularity

- Three operands keeps the instruction logically simple
- Examples:

C Code	Assembly Equivalent
A = b + c	add a, b, c
b = x * y	mul b, x, y
a = b + 42	addi a, b, 42

Simplicity Favors Regularity

- A more complex example

C Code	Assembly equivalent	
f = (g+h) – (i+j)	***add***	***$t0, g, h***
	add	***$t1, i, j***
	sub	***f, $t0, $t1***

- Notes
  - Consider the operator precedence
    - ( ) before -
  - This is a pseudo code
    - Cannot use the symbols g and h
    - Values should exist in some registers, then use register names or numbers

Simplicity Favors Regularity

- More Examples:

  Using variable names

  - C code:

    A = B + C + D + E

  - MIPS pseudocode:

```
add  A, B, C        # add B + C,put result into A
add  A, A, D        # put B + C +  D into A
add  A, A, E        # put B + C +  D + E into A
```

- Syntax:

```
add rd, rs, rt   # destination, suorce1, source2
```

- Exercise
  - Assume that A, B, C, D, & E are stored in registers $s0, … $s4, rewrite the code using registers' names

Simplicity Favors Regularity

- More Examples:
  - Using symbolic register names
    - C code:

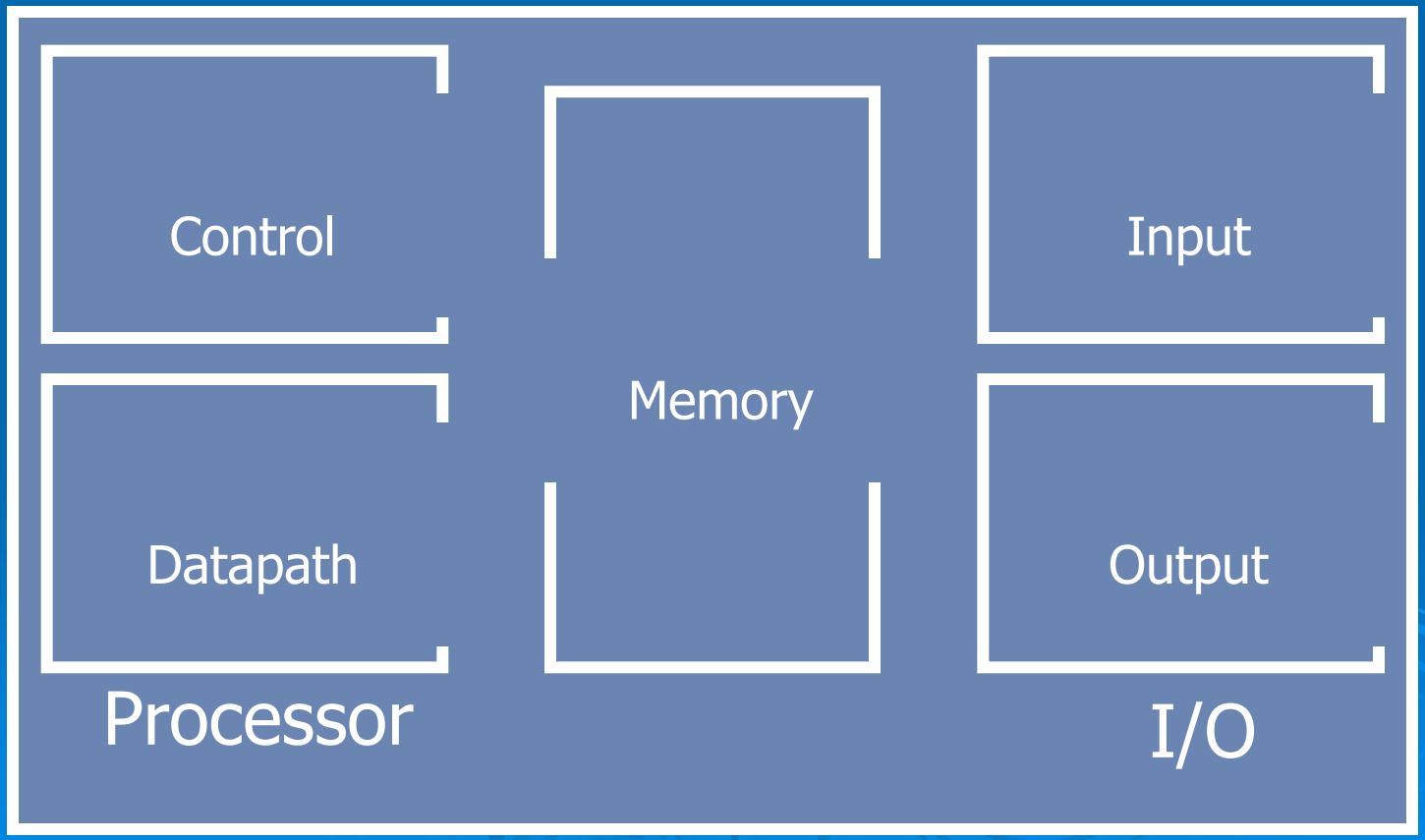
```
A = B + C + D;
E = F - A;
```

    - MIPS code:

```
add  $t0, $s1, $s2
add  $s0, $t0, $s3
sub  $s4, $s5, $s0
```

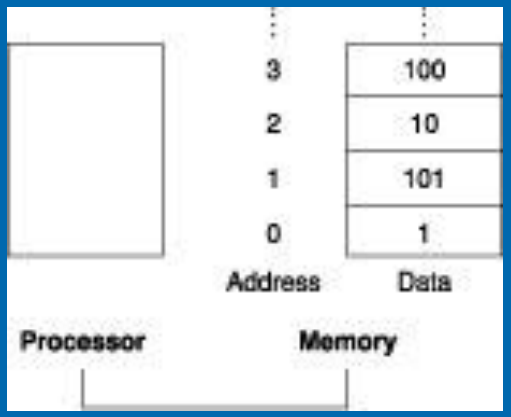
Why are register faster?

- Where are the registers?



Memory Access

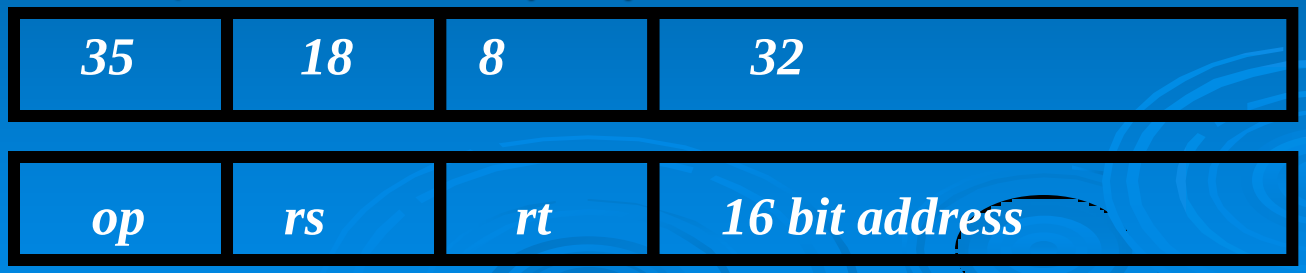
- Data transfer instructions are used to transfer data between registers and memory
- They must supply a memory address
- Example
  - C Code
    - g = h + A[8]
  - Assumptions
    - Register $s3 contains the base address of array A
    - 8 is the offset of the 8th element of the array
  - MIPS equivalent




```
lw      $t0, 8($s3)     # Temporary register $t0 gets A[8]
add     $s1, $s2, $t0   # g = h + A[8]
```

Instruction Format

- Compromise between providing for larger addresses & constants in instruction and keeping all instructions the same length
- Addresses needs more than 5-bits
  - Introduce a new type of instruction format for data transfer instructions (I-format)
  - We have two options:
    - Change instruction length for different types of instructions, or
    - Keep instruction length & change field format
    - Example: ***lw $t0, 32($s2)***



Memory Access

- ***lw*** instruction can load words within (+/-) 2^{15} immediately
- The meaning of the field (***$rt***) changes:
  - for ***lw***: destination register
  - for ***sw***: source register
- Each format is assigned a set of values of the op-field from which it recognizes how to treat the instruction (R- or I-format type) and how many operands are involved

Control Flow Instructions

- The ability to make decisions
- Change the control flow (i.e., "next" instruction to be executed)
- Types:
  - Conditional
  - Unconditional
- See Appendix for more comparison & branch instructions
- In high-level languages, you don’t have to write explicit labels
- Compilers create branches & labels that don’t appear in the HLL

Unconditional Branches

- Forms:
  - ***j label # jump to label***
  - ***jr rs #jump to addr stored in register***

Conditional Branches

- Forms:
  - ***beq*** (Branch on equal)
  - ***bne*** (Branch on not equal)
  - ***slt*** (Set on less than)
- Examples:

```
bne     $t0,$t1,L     # go to L if $t0≠$t1

beq     $t0,$t1,L     # go to L if $t0=$t1

slt     $t0,$t1,$t2   # $t0=1 if $t1<$t2, $t0=0 otherwise
```

More Control Flow Instructions

- Branch-if-less-than

  slt $t0,$s1,$s2 ⇔ ***if $s1 < $s2 then***
  $t0 = 1
  else
  $t0 = 0

- We can use this instruction to build

  blt $s1, $s2, Label

  - ***blt*** is a pseudo-instruction meaning “branch if less than”
- We can now build general control structures
- Note that the assembler needs a register to do this

From C to MIPS – Array Manipulation

- Arrays with Constant Index
  - C code: ***A[8] = h + A[8];***
  - Equivalent MIPS code:
    - Assumptions:
      - ***$s3*** contains starting address of the array ***A***
      - ***$s2*** contains the value of ***h***

```
lw      $t0,32($s3)     # $t0 gets A[8]
                        # offset =8 x 4 =32
add     $t0,$s2,$t0     # Add h
sw      $t0,32($s3)     # store value back in A[8]
```

From C to MIPS – Logical Operations

- Shifts
- Bitwise AND
- Bitwise OR
- Bitwise NOR

From C to MIPS – Logical Operations

- Shifts
  - Left/right (***sll, srl***)
  - Can be used to represent multiplication/division for multiples of 2
- Example

```
Sll $t2, $s0, 4    # reg $t2 = reg $s0 << 4 bits
```

From C to MIPS – Logical Operations

- Bitwise AND
  - Bit by bit operation
  - Leaves a 1 in the result only if both bits of the operands are 1
- Example:
  - Assumption

    $t2 = 0000 0000 0000 0000 0000 1101 0000 0000

    $t1 = 0000 0000 0000 0000 0011 1100 0000 0000
  - Operation

    ```
    and $t0, $t1, $t2          # reg $t0 = reg $t1 & reg $t2
    ```
  - Result

    $t0 = 0000 0000 0000 0000 0000 1100 0000 0000

From C to MIPS – Logical Operations

- Bitwise OR
  - Bit by bit operation
  - Leaves a 1 in the result only if any bit of the operands is 1
- Example:
  - Assumption

    $t2 = 0000 0000 0000 0000 0000 1101 0000 0000

    $t1 = 0000 0000 0000 0000 0011 1100 0000 0000
  - Operation

    ```
    or $t0, $t1, $t2  # reg $t0 = reg $t1 | reg $t2
    ```
  - Result

    $t0 = 0000 0000 0000 0000 0011 1101 0000 0000

From C to MIPS – Logical Operations

- Bitwise NOR
  - Bit by bit operation
  - Inverse of OR
- Example:
  - Assumption

    $t2 = 0000 0000 0000 0000 0000 1101 0000 0000

    $t1 = 0000 0000 0000 0000 0011 1100 0000 0000
  - Operation

    ```
    and $t0, $t1, $t2       # reg $t0 = ~ (reg $t1 | reg $t2)
    ```
  - Result

    $t0 = 1111 1111 1111 1111 1100 0010 1111 1111

From C to MIPS – Array Manipulation

- Exercise:
  - What should change in the previous MIPS code for EACH OF the following C-statements?

```
A[300] = h + A[300];
A[16] = h + A [8];
A[i]  = h + A [i];
```

  - Write the equivalent machine code in each case

From C to MIPS – Array Manipulation

- Arrays with Variable Index
  - C code:

    g = h + A [i];
  - Equivalent MIPS code

From C to MIPS – Array Manipulation

- Arrays with Variable Index
  - C code:

    g = h + A [i];
  - Equivalent MIPS code
    - Assumption: ***$s4*** contains ***I***

```
add     $t1, $s4, $s4
add     $t1, $t1, $t1
add     $t1, $t1, $s3
lw      $t0, 0($t1)
add     $s1, $s2, $t0
```

From C to MIPS – Array Manipulation

- Arrays with Variable Index
  - C code:

    g = h + A [i];

  - Equivalent MIPS code
    - Assumption: ***$s4*** contains ***i***

```
    # Multiply index by 4 due to byte addressing
    # Store the value in $t1
add         $t1, $s4, $s4       # $t1 = 2 *i
add         $t1, $t1, $t1       # $t1 = 4 *i
    # Base is stored in $s3
    # Get address of A[i]
add         $t1, $t1, $s3       #$t1=Address(A[i])
    #Load A[i] into temporary register
lw    $t0, 0($t1)           # $t0 = A[i]
    # Add A[i] to h
add         $s1, $s2, $t0       # $s1 = h + A[i]
    # $s1 corresponds to g
```

From C to MIPS - If-Statement

- C-Code:

```
if (i==j)
        h = i + j;
```

- Equivalent MIPS Code:

```
        bne   $s0, $s1, Label
        add   $s3, $s0, $s1
Label:  ....
```

- Assumptions:
  - *$s0 = i*
  - *$s1 = j*
  - *$s3 = h*

From C to MIPS- If-else statement

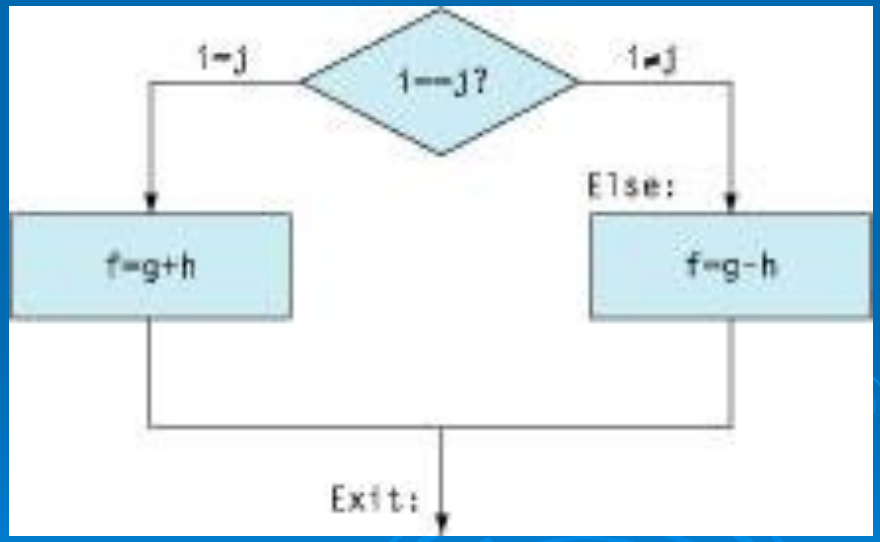
- C statement

```
if (i != j)
        f = g + h;
else
        f = g - h;
```

- Equivalent MIPS code:

```
        beq     $s0, $s1, Else
        add     $s2, $s3, $s4
        j       Exit
Else:   sub     $s2, $s3, $s4
Exit:   ...
```

- Assumptions:
  - *$s0 = i*
  - *$s1 = j*
  - *$s2 = f*
  - *$s3 = g*
  - *$s4 = h*



From C to MIPS – For Loops

- C-Code

  for (; i != h; i = i+j)

  g = g + a[i];

- Equivalent MIPS code:

- Assumptions:

  $s1 = g ***$s2 = h***

  $s3 = i ***$s4 = j***

  $s5 = base address of array A

From C to MIPS – For Loops

- C-Code

```
for (; i != h; i = i+j)
    g = g + a[i];
```

- Assumptions:
  - *$s1 = g $s2 = h*
  - *$s3 = i $s4 = j*
  - *$s5 = base address of array A*

- Equivalent MIPS code:

```
Loop:   add   $t1,$s3,$s3    # $t1 = 2 * i
        add   $t1,$t1,$t1    # $t1 = 4 * i
        add   $t1,$t1,$s5    # $t1 = addr(A[i])
        lw    $t0,0($t1)     # $t0 = A[i]
        add   $s1,$s1,$t0    #g = g +A[i]
        add   $s3,$s3,$s4    #i = i + j
        bne   $s3,$s2,Loop   #go to Loop if i≠h
```

Note: Check if this is not a do-while loop!!!

From C to MIPS – While Loop

- C-Code

  while (a[i] == k)

  i = i + j;

- MIPS Equivalent

- Assumptions:

  $s3 = i
  $s4 = j
  $s5 = k
  $s6 = Base address of A

From C to MIPS – While Loop

- C-Code

```
while ( a[i] == k )
    i = i + j;
```

- MIPS Equivalent

```
Loop:
    sll     $t1, $s3, 2     # $t1 = 4 * i
    add     $t1, $t1, $s6   # $t1=addr(A[i])
    lw      $t0, 0($t1)     # $t0 = A[i]
    bne     $t0, $s5, Exit  # goto Exit if A[i]≠k
    add     $s3, $s3, $s4   # i=i+j
    j       Loop            # Loop back
Exit:       ...             # Next statement
```

- Assumptions:
  - *$s3 = i*
  - *$s4 = j*
  - *$s5 = k*
  - *$s6 = Base address of A*

From C to MIPS - Less Than Test

- C-Code:

  if (a < b) goto Less;

- Equivalent MIPS code :

```
slt  $t0, $s0, $s1        # $t0=1 if $s0 < $s1
                          # ($s0=a, $s1=b)
bne  $t0,$zero,Less       #goto Less if $t0≠0
```

From C to MIPS - Switch Statement

- The jump register ***(jr)*** instruction is used
- Unconditional jump to the address given in the register
- Possibilities
  - Convert it into a group of nested ***if-then-else*** statements
  - Use a table of addresses ***(jump address table)*** for the instruction sequences and use an index to jump to the appropriate entry

From C to MIPS - Switch Statement

- C-Code:

```
switch(k)
{       case 0:  f = I + j; break;      /* k=0 */
        case 1:  f = g + h; break;      /* k=1 */
        case 2:  f = g - h; break;      /* k=2 */
        case 3:  f = I - j; break;      /* k=3 */
}
```

- Steps:

1. Check that k in within limits, otherwise exit
2. From k, find out where to jump to (using index table)
3. After statement execution, jump to Exit label (break)

From C to MIPS - Switch Statement

- Assumptions:

$s0 = f, $s1 = g, $s2 = h, $s3 = I, $s4 = j, $s5= k, $t2= 4

```
      slt   $t3, $s5, $zero      # test if k<0 ($s5=k)
      bne   $t3, $zero, Exit     # go to Exit if k <0
      slt   $t3, $s5, $t2        # Test if k<4, $t2=4
      beq   $t3, $zero, Exit     # go to Exit if k>=4
      add   $t1, $s5, $s5        # $t1 = 2*k
      add   $t1, $t1, $t1        # $t1=4*k=jump address
      add   $t1, $t1, $t4        #$t1=addr(JumpTable[k])
      lw    $t0, 0($t1)          # $t0=JumpTable[K]
      jr    $t0
L0:   add   $s0, $s3, $s4        # k=0 => f=i+j
      j     Exit
L1:   add   $s0, $s1, $s2        # k=1 => f=g+h
      j     Exit
L2:   sub   $s0, $s1, $s2        # k=2 => f g-h
      j     Exit
L3:   sub   $s0, $s3, $s4        # k=3 => f=i-j
Exit: ...
```

Input/Output

- We are not going to discuss MIPS I/O instructions, except what is necessary to display messages on the console window
- See examples

Procedure Calls

- Execution of a procedure follows 6 steps
  1. Place parameters in a place where the procedure can access them
  2. Transfer control to the procedure
  3. Acquire storage resources to the procedure
  4. Perform desired task
  5. Place result in a place accessible by the calling program
  6. Return control to the point of origin

Procedure Calls

- MIPS register convention for procedures
  - $a0-$a3: 4 arguments registers to pass parameneters
  - $v0-$v1: 2 value registers to return values
  - $ra: return address register to return to point of origin

Procedure Calls

- MIPS instructions used with procedures
  - jal: Jump & Link
    - Jump to an address & save address of the following instruction in $ra register
  - jr $ra: Jump to return address
    - Jump to the address stored in $ra

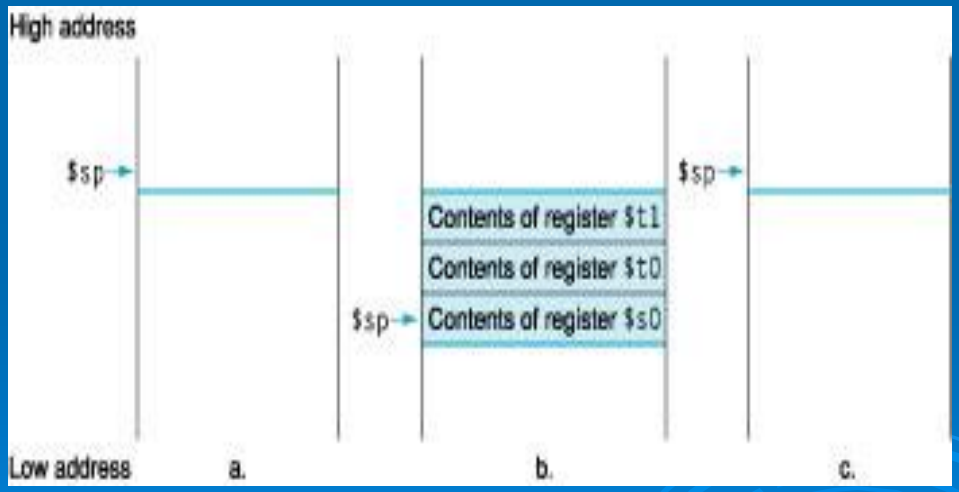
Procedure Calls

- What if more than 4 arguments need to be transferred?
  - Put it onto the stack
- Stack:
  - Needs a pointer ($sp) to the most-recently allocated address, to show where the next procedure should be allocated
  - $sp grows from higher to lower address
    - Push: subtract from $sp
    - Pop: Add to $sp

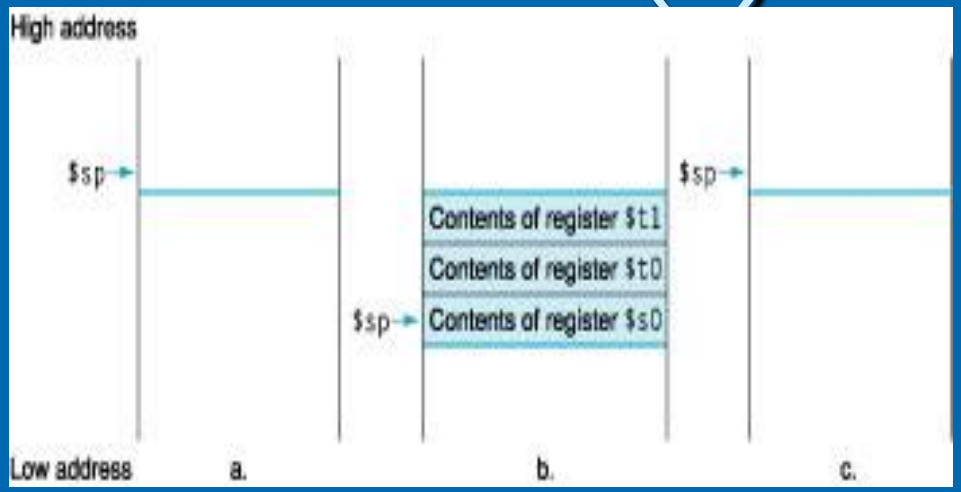
Example: Leaf Procedure

- Leaf procedure doesn't call other procedures
- C Code

```
Int leaf_example (int g, int h, int I, int j)
{
    int f;
    f = (g + h) – ( I + j)
    return f;
}
```



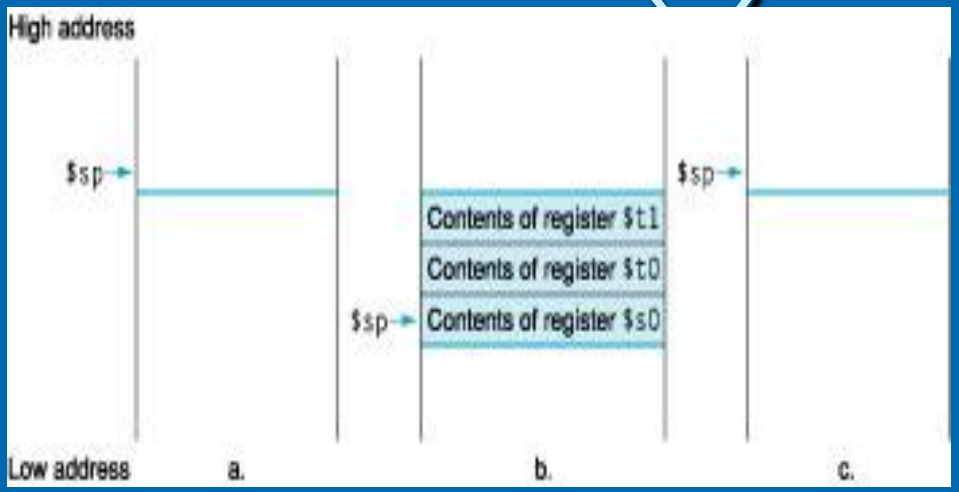
Example: Leaf Procedure(1)



- MIPS Equivalent(1)

```
Leaf_example:
    addi    $sp, $sp, -12     # adjust stack to make room for 3 items
    sw      $t1, 8($sp)       #save $t1 on stack
    sw      $t0, 4($sp)       # save $t0 on stack
    sw      $s0, 0($sp)       # save $s0 on stack
    add     $t0, $a0, $a1     # $t0 contains g + h
    add     $t1, $a2, $a3     # $ t1 contains I + j
    sub     $s0, $t0, $t1     # f = $t0 - $t1 = (g+h)-(I-j)
    add     $v0, $s0, $zero   # return result to calling point = f = ($v0 =
    $s0+0)
```

Example: Leaf Procedure(2)



- MIPS Equivalent(2)

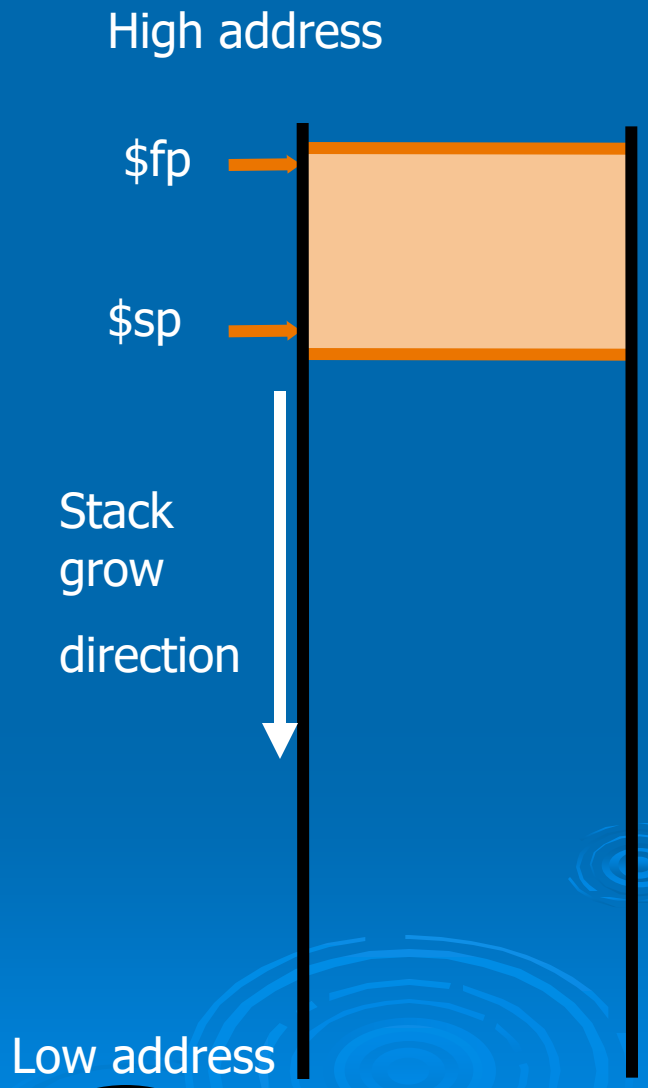
```
$s0, 0($sp)   # restore $s0 for caller
    lw        $t0, 4($sp)       # restore $t0 for caller
    lw        $t1, 8($sp)       # restore $t1 for caller
    addi      $sp, $sp, 12      # adjust stack to delete 3 items
    jr        $ra               # jump back to calling routine
```

Procedure Call Frame

- Memory block associated with the call, usually saved onto stack
- Includes
  - Argument values
  - Registers possibly modified by the procedure
  - Local variables
- Stack frame:
  - Stack block used to hold a procedure call frame
- Frame pointer (***$fp***):
  - Points to the first word in the frame
- Stack pointer (***$sp***):
  - Points to last word of the frame

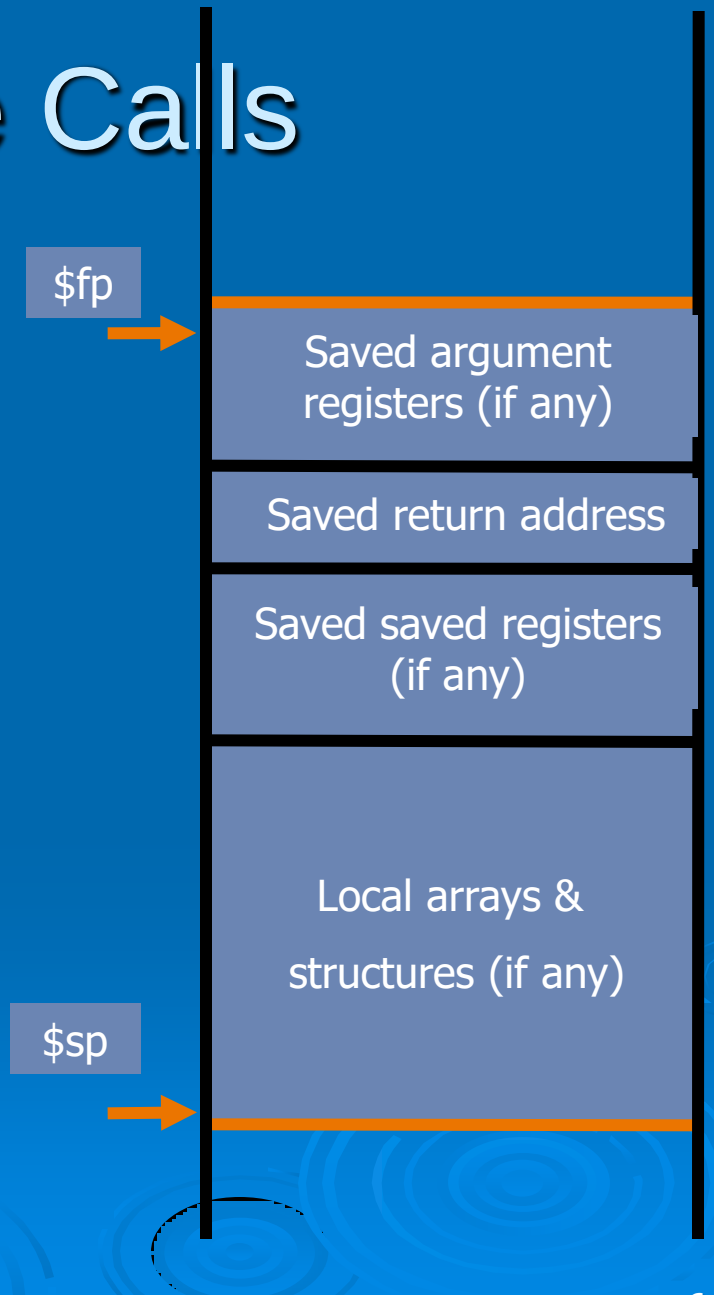
Procedure Calls

- Before the call:
  1. Pass the first 4 arguments to registers ***$a0-$a3***. The system will take care of them
  2. Remaining arguments, if any, should be pushed onto stack
  3. Save caller-saved registers onto the stack as well, since the called function might use those registers and overwrite their contents
  4. Perform ***jal*** instruction
     - Jump to callee's first instruction
     - Save return address in ***$ra***



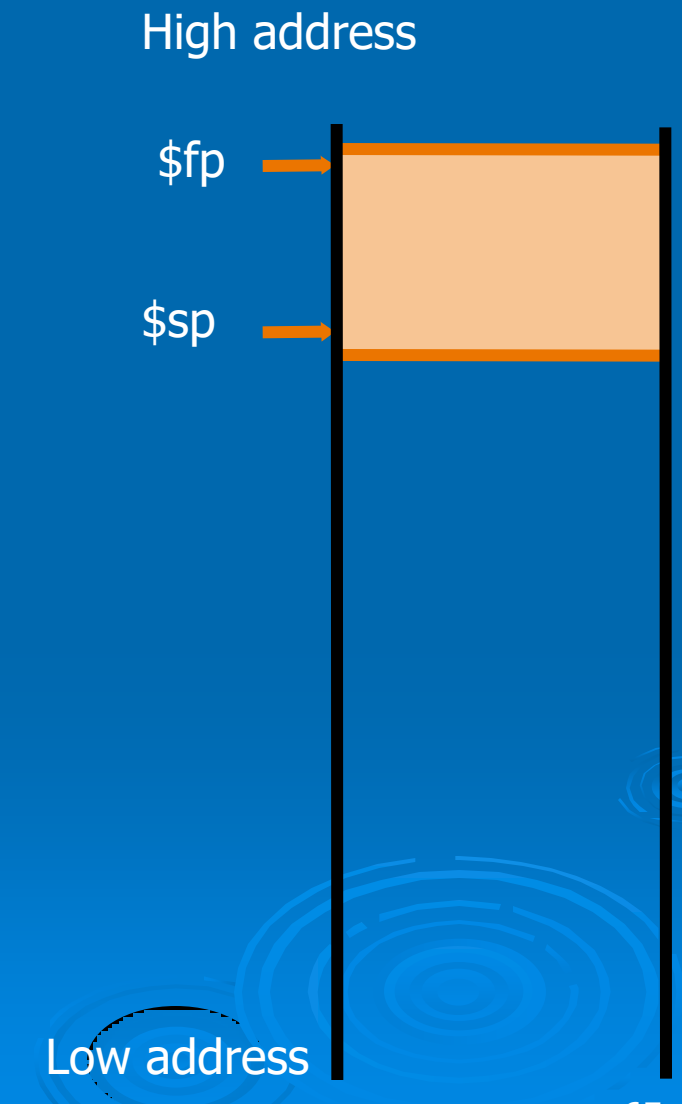
Procedure Calls

- Before execution of called procedure:
  1. Allocate memory for a stack frame
  2. Save callee-saved registers in the frame
  3. Update frame pointer

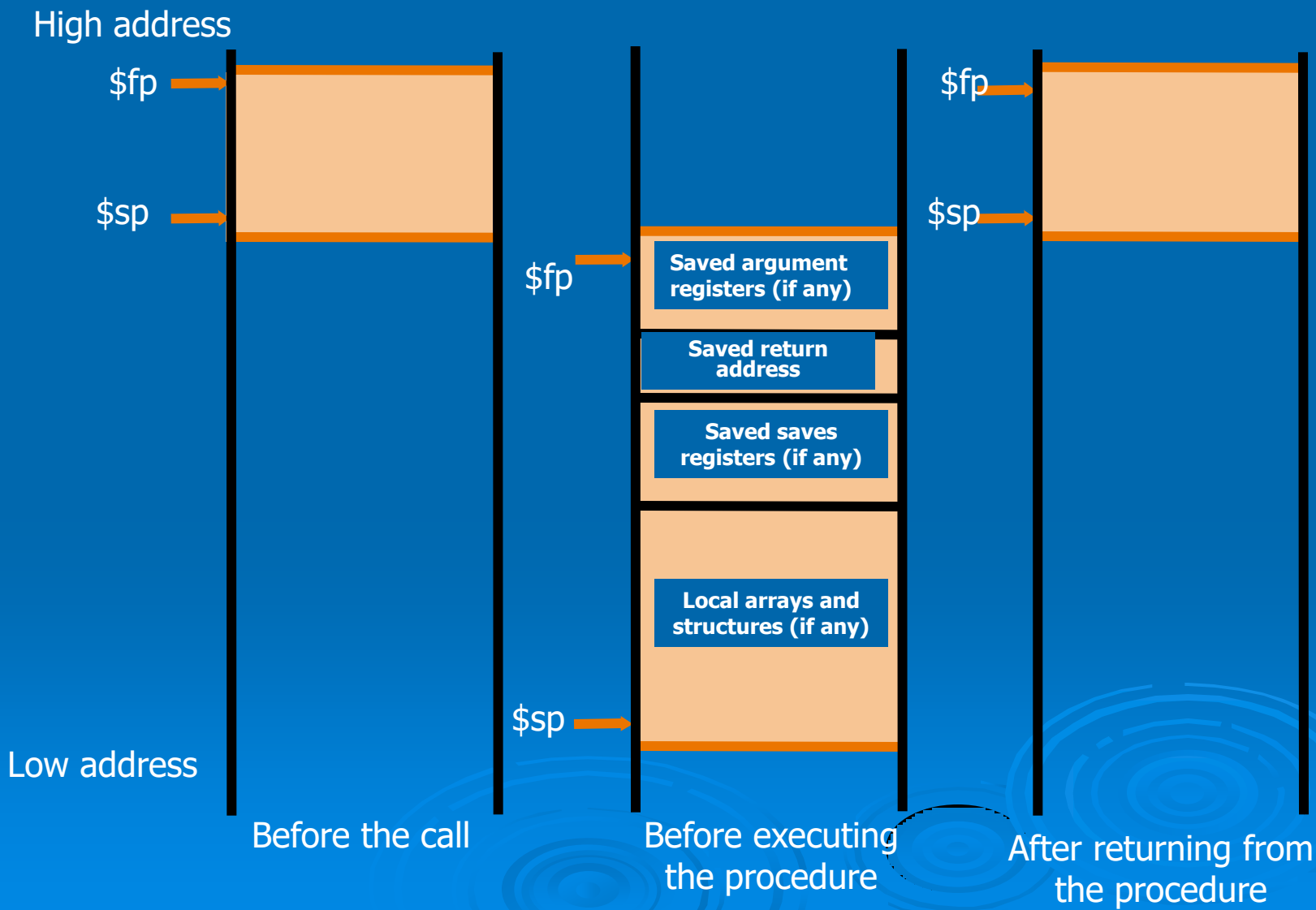


Procedure Calls

- Before returning from the procedure:
  1. Place return value, if any, in ***$v0*** register
  2. Restore callee-saved registers by retrieving their saved contents from the stack
  3. Pop stack frame to free the memory used by the procedure
  4. Jump to the return address stored in ***$ra***



Procedure Calls Review



Nested Procedures

- One procedure calls another, or calls itself (recursion)
- Example: Factorial
  - C Code

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

Nested Procedures

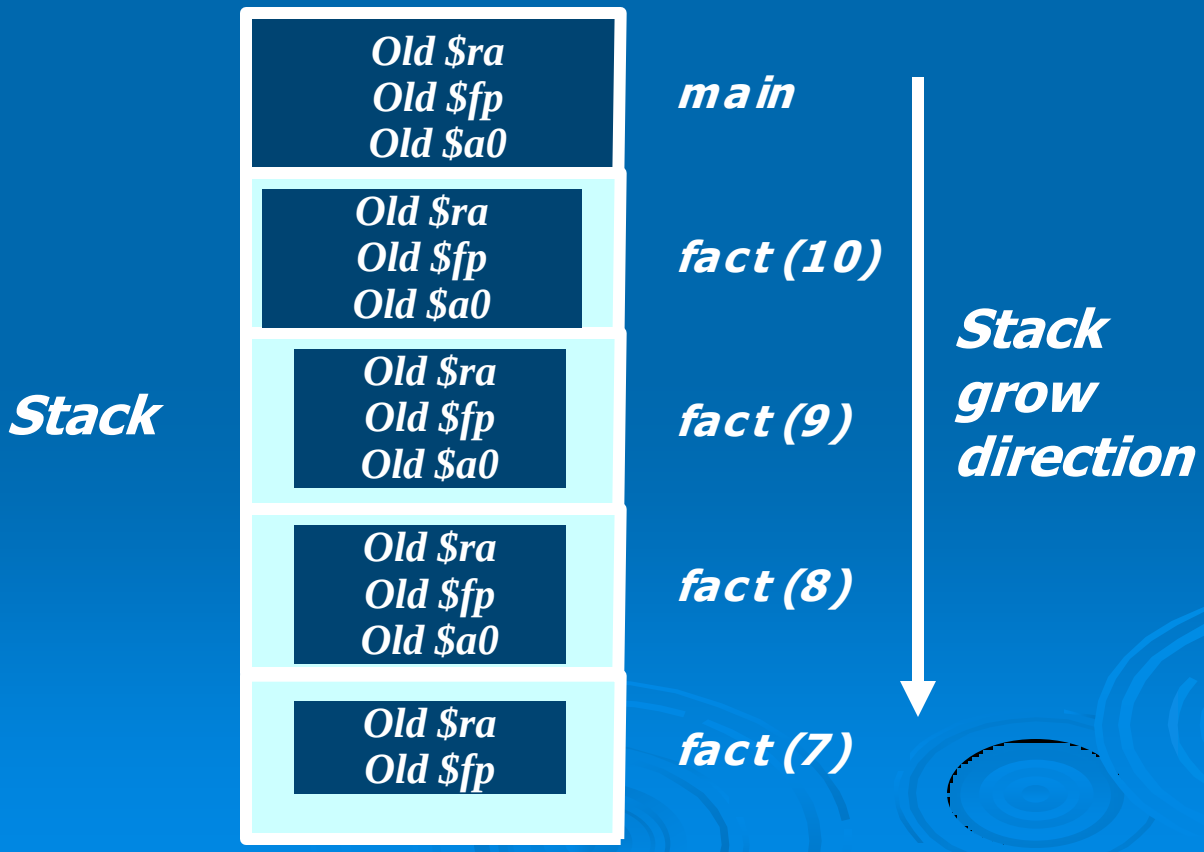
Example: Factorial

- MIPS Code

```
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, L1      # if n >= 1, goto L1
        addi    $v0, $azero, 1      # return 1
        addi    $sp, $sp, 8         # pop 2 items off stack
        jr      $ra                 # return to after jal
L1:     addi    $a0, $a0, -1        # N >=1: argument gets (n-1)
        jal     fact                # call fact with (n-1)
        lw      $a0, 0($sp)         # return from jal: restore argument n
        lw      $ra, 4 ($sp)        # restore return address
        addi    $sp, $sp, 8         # adjust stack pointer to pop 2 items
        mul     $v0, $a0, $v0       # return n * fact (n-1)
        jr      $ra                 # return to caller
```

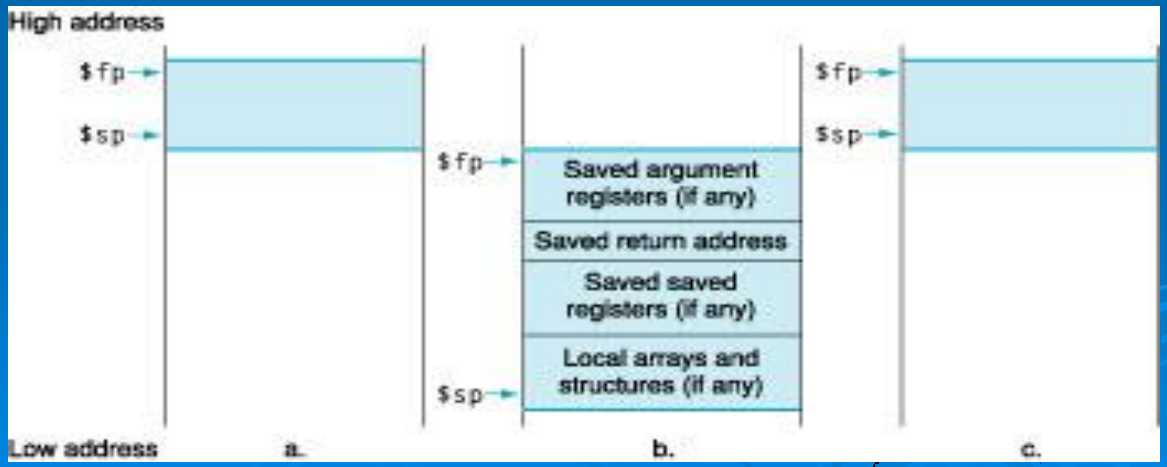
Procedure Calls Example

- Example: Factorial
  - Main calls Fact(10)
  - Stack frame during call of **fact(7)**



Allocating New Data on Stack

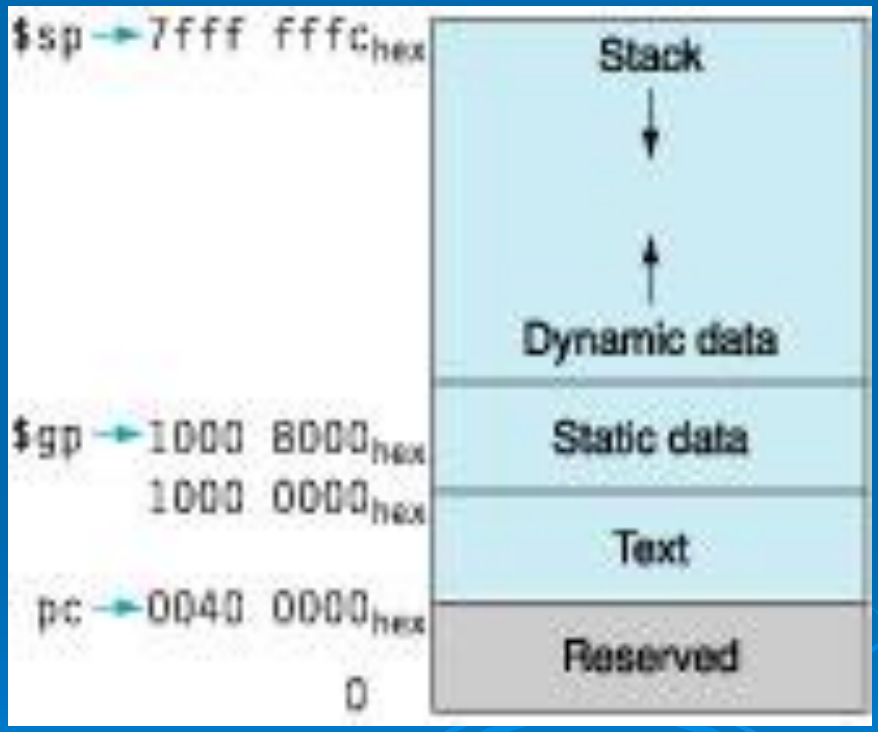
- Stack is used to store variables local to the procedure that don't fit in registers
- Some MIPS software use frame pointer $fp to point to the first word of the frame of a procedure to allow reference for local variables
- $fp offers a stable base register within a procedure for local memory reference



Stack before call Stack during call Stack after call

Allocating Space on the Heap

- $0000\ 0000_{hex}$:
  - First part of the low end is reserved by the system
- $0040\ 0000_{hex}$:
  - Followed by the text segment
- $1000\ 0000_{hex}$:
  - Static data are above the text segment used for constants & other static variables
- $1000\ 8000_{hex}$:
  - Heap hosts dynamic data structures (e.g. linked lists)
  - Stack starts in high-end of memory & grows down
  - Stack & heap grow in opposite directions

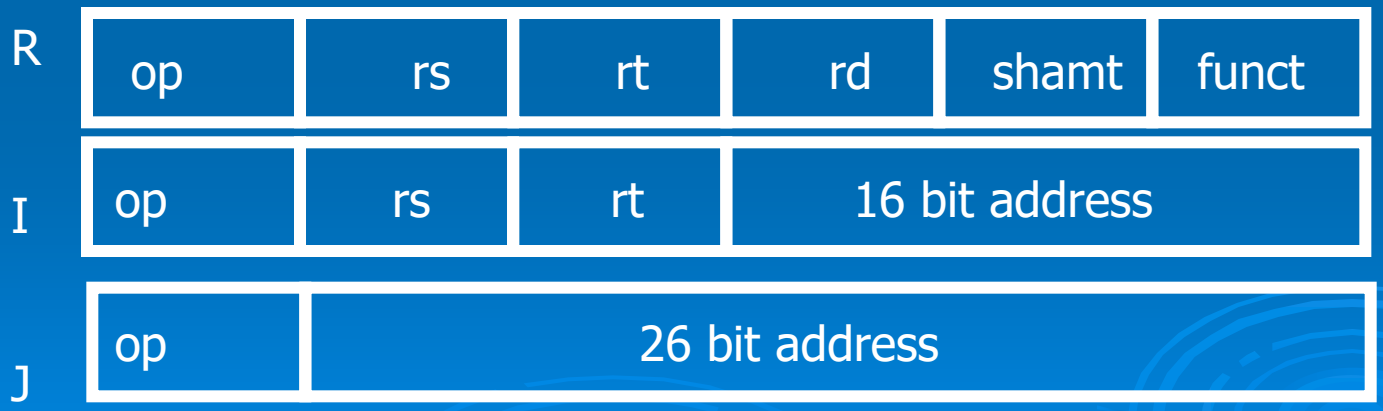


Review - Loading Programs for Execution

1. Determine size of text & data segments from executable file header
2. Create enough address space for program's text & data segments, in addition to a "stack segment"
3. Copy both instruction & data segments into address space
4. Copy arguments onto stack
5. Initialize Instruction register & stack pointer
6. Copy arguments from stack to registers
7. Call program's main routine
8. When returning from main program, terminate with exit system call

Review - MIPS instruction Formats

- Simple instructions all 32 bits wide
- Very structured
- Addresses are not 32 bits
- Only three instruction formats



Review - Branch instructions

bne $t4,$t5,Label # Next instruction is at Label if $t4 ≠ $t5

beq $t4,$t5,Label # Next instruction is at Label if $t4 = $t5

- Formats:

- We could specify a register (like ***lw*** and ***sw***) and add it to address
  - Most branches are local (***principle of locality***)
  - Use Instruction Address Register (PC = program counter)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB

Review - Addressing

1. Register addressing:
   - Operands are registers
2. Base (Displacement addressing):
   - Operand location =
     register + constant (offset) in the instruction
3. Immediate addressing:
   - Operand is a constant within the instruction
4. PC-relative addressing:
   - Address = PC (program counter)
     + constant in the instruction
5. Pseudo addressing:
   - Jump address = 26 bits of the instruction
     + upper bits of the PC

➢ A single operation can use more than one addressing mode (e.g. ***add, addi***)

Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - Simplicity favors regularity
  - Smaller is faster
  - Good design demands compromise
  - Make the common case fast
- Instruction set architecture
  - A very important abstraction

Thank you