Chapter 3: MIPS Instruction Set

1

Review

```
Instruction           Meaning

add $s1,$s2,$s3    $s1 = $s2 + $s3
sub $s1,$s2,$s3    $s1 = $s2 – $s3

addi $s1,$s2,4     $s1 = $s2 + 4
ori  $s1,$s2,4     $s2 = $s2 | 4

lw $s1,100($s2)    $s1 = Memory[$s2+100]
sw $s1,100($s2)    Memory[$s2+100] = $s1

bne $s4,$s5,Label  Next instr is at Label if $s4 ¹ $s5
beq $s4,$s5,Label  Next instr is at Label if $s4 = $s5
slt $t1,$s2,$s3     if $s2 < $s3, $t1 = 1 else $t1 = 0

j Label            Next instr is at Label
```

2

Pseudo-instructions

- The MIPS assembler supports several pseudo-instructions
  - Programmers can use pseudo-instructions
  - Assembler translates them into actual instructions or sequences of instructions
- Example

  move $7,$18 contents of $18 are copied to $7

  is translated into

  add $7, $18, $0 Remember: $0 always contains 0
- Other Examples:
  - blt, seq, sle, la, li (See Appendix A for complete list)

The jr (jump register) instruction

jr $s1 # jump to address in $s1

- Usage
  - Switch/Case statements
  - Returning from procedures/functions

Using jr for implementing a switch statement

```
switch (k) {
    case 0: f = i + j; break;
    case 1: f = g + h; break
    case 2: f = g – h; break
}
```

Assume

1. f, g, h, i, j are in registers $16 to $21, register $10 contains the constant 4
2. In memory, the code for case 0 is at address L0, the code for case 1 is at address L1, and the code for case 2 is at address L2
3. An array JumpTable has already been created with the first word (at address JumpTable) containing L0, the second word (address JumpTable + 4) containing L1, and the third word (JumpTable + 8) containing L2

5

```
        mult  $9, $21, $10          # $9 = k*4
        lw    $8, JumpTable[$9]
        jr    $8
L0:     add   $16, $19, $20
        j     Exit
L1:     add   $16, $17, $18
        j     Exit
L2:     sub   $16, $17, $18
Exit:
```

6

Supporting procedure calls

- jr instruction
  - Returning from a procedure
- Jump and link (jal) instruction
  - Jump to address of procedure, while storing the return address in register $31 ($ra)
    - What is the return address?
      - PC + 4
    - In MIPS, a special register called Program Counter (PC) contains the address of the instruction currently being
  - "jal addr" stores PC+4 in register $31, and then jumps to location "addr"
  - To return from the procedure, we can simply execute "jr $31"

7

Supporting procedure calls (cont'd)

- Passing arguments/parameters
  - Parameters are passed in registers $4 - $7 ($a0 - $a3)
- Returning results
  - Results are returned in register $v0
- Example

```
procA:      <code for procedure procA>
            move    $a0, $s1        # assume parameter for
                                    # proc B is in register $s1
            jal     procB
            <more code for procedure procA>
            jr      $ra

procB:      < code for procedure procB>
            move    $v0, $t1        # move result to $v0
            jr      $ra
```

8

Supporting procedure calls (cont'd)

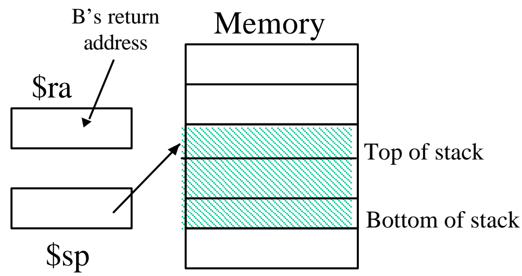
- Problems:
  - What if procB calls another procedure?
    - Contents of $ra will be overwritten!
  - What if we have more than 4 parameters?
- Solution:
  - Use stack in main memory for storing
    1. Parameters (if procedure has more than 4)
    2. Any registers that need to be preserved across procedure calls, I.e. registers that should have the same values before and after the procedure call
- The stack is an area of memory that grows and shrinks dynamically.
  - Register $29 ($sp) points to the top location of the stack, I.e. register $29 is used as the stack pointer

Example: preserving registers across procedure calls

```
A:      .........
        jal B
        ......
B:      ......
        ......
        addi    $sp, $sp,-4   } Save $ra on stack
        sw      $ra, 0($sp)
        jal     C
        lw      $ra, 0($sp)   } Restore $ra from stack
        addi    $sp, $sp,4
        ......
        ......
        jr      $ra
C:      ......
        ......
        jr      $ra
```

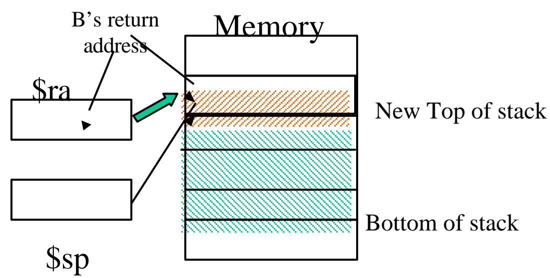
NOTE: In MIPS, the stack grows downwards

1. After A calls B



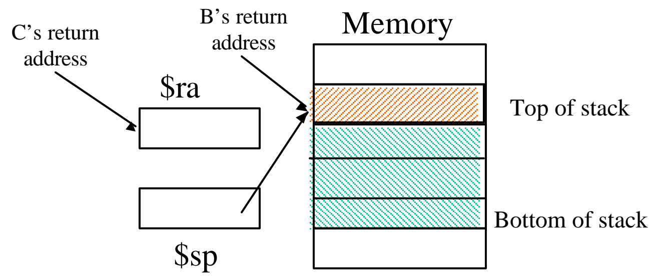
11

2. Just before B calls C



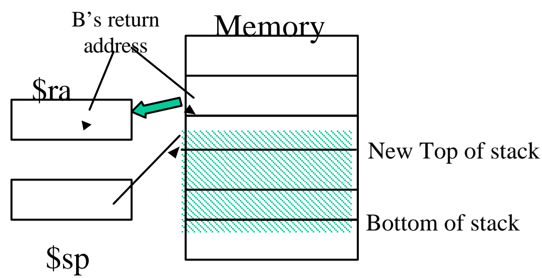
12

3. After B calls C



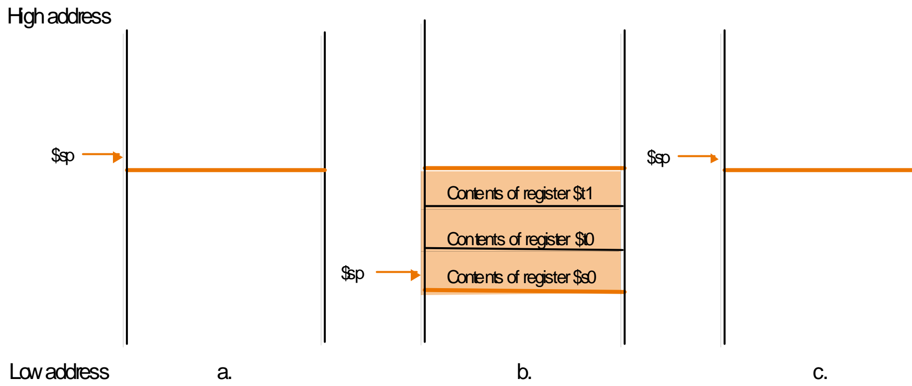
13

4. Just before B returns

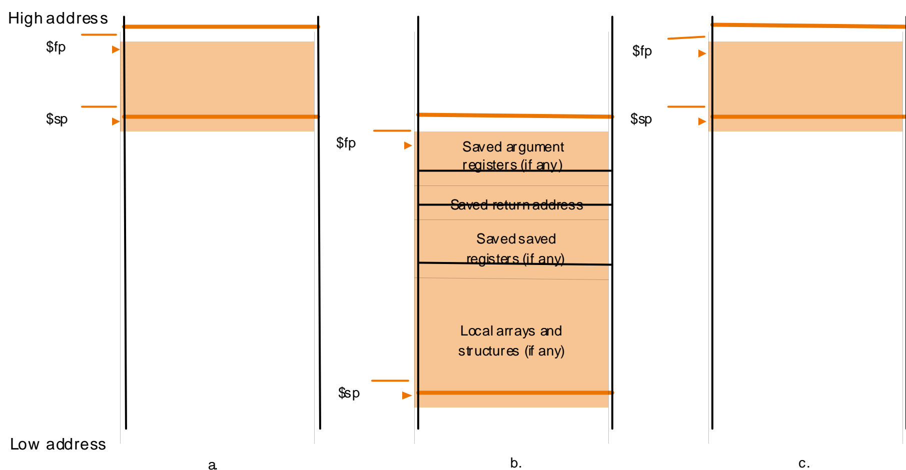


14

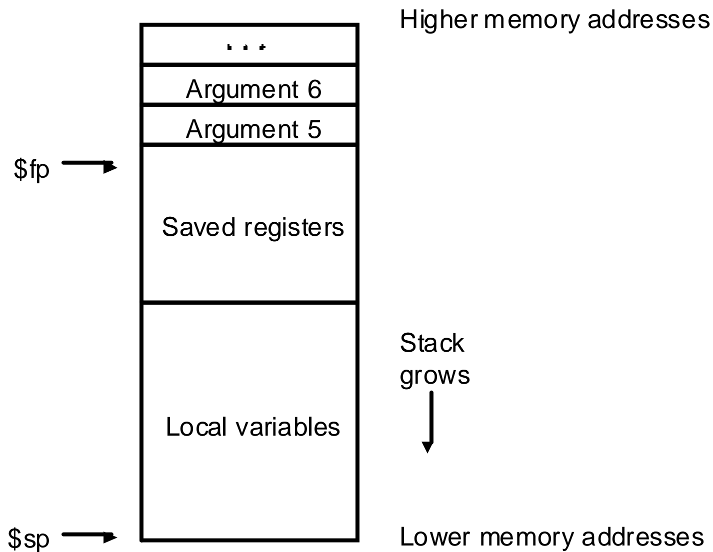
Using the stack to save registers



Procedure frames are pushed and popped off the stack

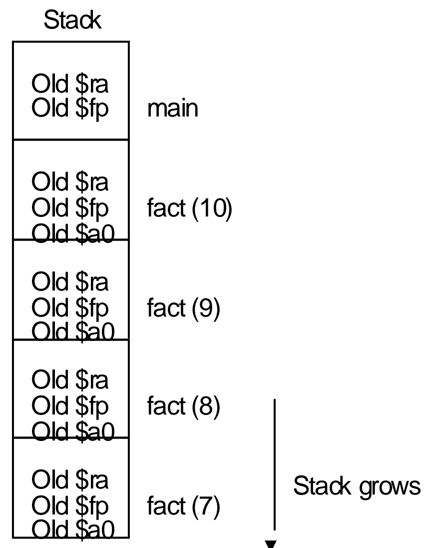


Contents of a procedure frame



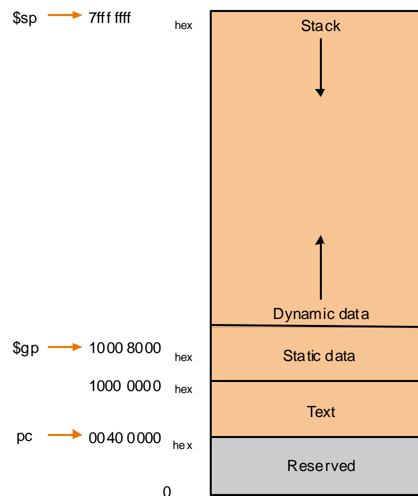
Why do running programs need a stack?

- Modern programming languages are recursive
- Example: Factorial program is recursive
  - Has a separate frame for each invocation of factorial()



Address space for a running program

- Address Space = Memory allocated to program
- 3 segments
  - Text (code)
  - Data
    - Static data
    - Dynamic data
  - Stack



Conventions for saving and restoring registers across procedure calls

- If a procedure modifies any registers that are used by calling routine, some convention is needed for saving & restoring registers
  1. Caller save: calling procedure saves and restores any registers that must be preserved across the call
  2. Callee save: called procedure saves and restores any registers that it might use
  3. MIPS convention: some registers are caller saved and some registers are callee saved

MIPS procedure call convention

- Caller:
  1. Pass arguments. First 4 are in $a0-$a3. Remaining are pushed on to stack and appear at the beginning of called procedure's stack frame
  2. Save caller-saved registers ($a0-$a3,$t0-$t9)
  3. Execute jal
- Callee: (before it starts running)
  1. Allocate memory for stack frame
  2. Save callee-saved registers in the stack frame ($s0-$s7,$fp,$ra)
  3. Establish frame pointer
- Callee: (before returning)
  1. Place return value in $v0
  2. Restore all callee-saved registers
  3. Pop the stack frame
  4. Return by jumping to $ra

21

MIPS Assembly Language

- Assembler Directives

  .allign n e.g. .allign 2

  .asciiz <str>

  .data <addr>

  .space n

  .text

  .globl

  See Appendix A for more details and examples

- System Calls for Input/Output
  1. Load system call code into register $v0 and arguments into $a0-$a3
  2. Execute syscall

  See Figure A.17

22

Assignment 2

Assignment 2: matrix multiplication

- Multiply M1 (r1 rows, c1 columns) and M2 (r2 rows, c2 columns) to obtain Mr (r1 rows, c2 columns)
  - Note c1 = r2
- Three procedures
  - Main()
  - Matrix_multiply(r1,c1,M1,r2,c2,M2,Mr)
  - Inner_product(row,num_columns,M,column,num_rows,N)
- Need to use the stack
  - for saving & restoring registers
  - Passing parameters
- Matrices stored in single dimensional array using **row-major organization**

23

Matrix Multiplication

Inner Product

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

n is the number of columns in matrix A, and the number of rows in matrix B

24