CS 465 Lecture 4

MIPS ISA III: Procedures

*Slides adapted from Computer Organization and Design by Patterson and Henessey

Road Map – MIPS ISA

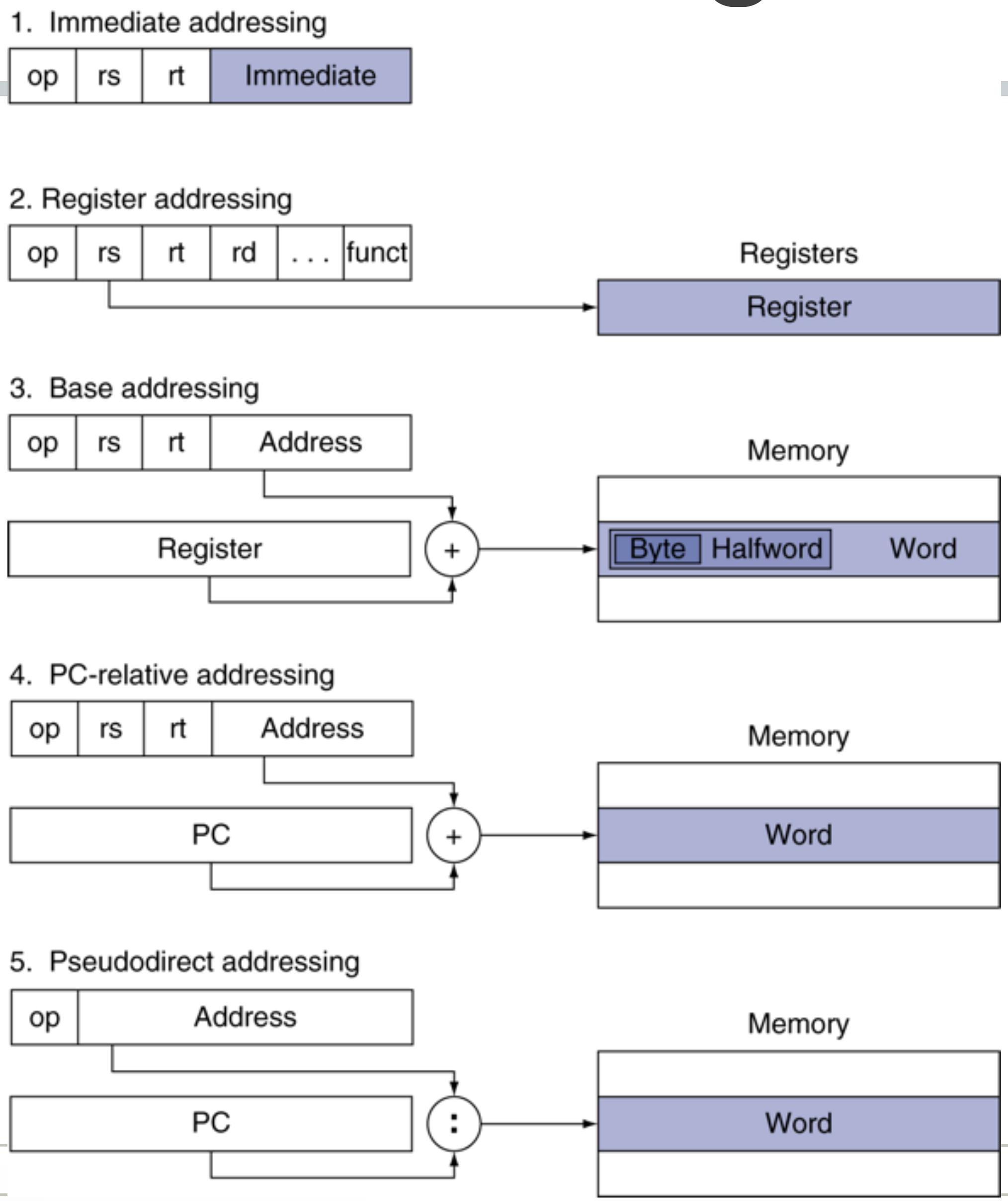
- MIPS basic instructions
- MIPS instruction format
  - I/J/R – encoding/decoding
  - Large immediates
  - PC-relative branch addressing
  - Jump instruction: pseudo-direct addressing
- Procedure calls ←
- Misc.

Review: MIPS Instruction Format

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

- Fixed instruction length
- Instruction format as regular as possible

MIPS Addressing Modes



Warmup

- PC address is at 0x0FFF 1000
- Can we use a single branch (beq or bne) to get to
  - a) 0x 0FFE 1004
  - b) 0x 1000 1004
- How about a single j instruction ?

Procedure Calling

```
int main() {
  int i,j,k,m;
  ...
  i = mult(j,k);  ...
  m = mult(i,i);  ...

}
```

What information must compiler or programmer keep track of?

- Control flow
  - Caller → callee → caller: need to know where to jump/return
- Data flow
  - Caller → callee: arguments
  - Callee → caller: return value
- Shared resources
  - Memory, register

Procedure Calling

- Steps required
  1. Prepare and pass arguments to callee
  2. Transfer control to callee procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Pass result to caller
  6. Return to place of call

Register Usage

- Registers play a major role in keeping track of information for function calls
- Register conventions in MIPS
  - Return address: $ra (reg 31)
  - Arguments: $a0, $a1, $a2, $a3 (reg's 4 – 7)
  - Return value: $v0, $v1(reg's 2 and 3)
  - Local variables: $s0, $s1, … , $s7
  - Temporaries: $t0, $t1, … , $t7
- The stack is also used; more later
  - $gp: global pointer for static data (reg 28)
  - $sp: stack pointer (reg 29)

Procedure Calling

- Steps required

1. Prepare and pass arguments to callee ✓
   - $a0, $a1, $a2, $a3
2. Transfer control to procedure ←
3. Acquire storage for callee procedure
4. Perform procedure’s operations
5. Pass result to caller ✓
   - $v0, $v1
6. Return to place of call

Program Counter

- One register keeps address of instruction being executed: Program Counter (PC)
    - Basically a pointer to memory
    - IP/EIP/RIP in x86 assembly
    - Need to be updated to address of next instruction
        - Sequential
        - Branches
        - Calls and returns

Procedure Control Flow

- Procedure call: jump and link

```
jal ProcedureLabel
```

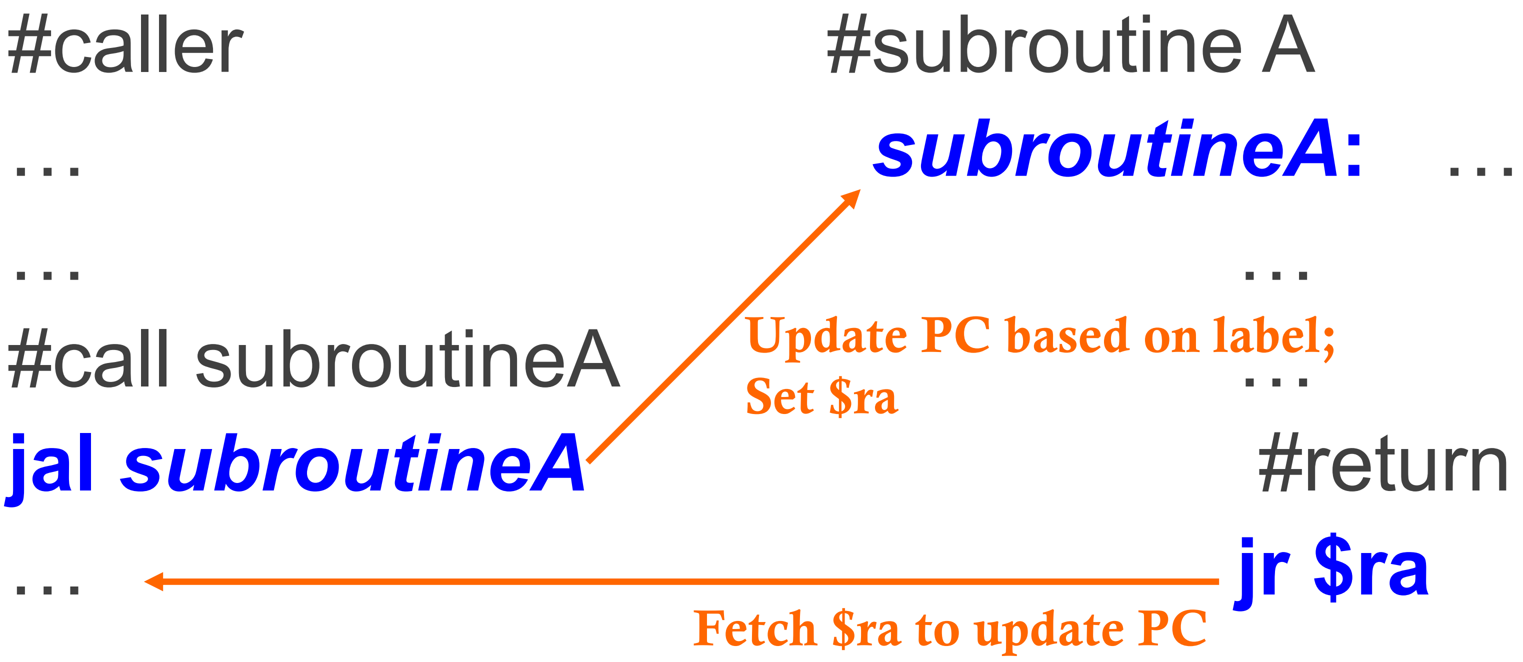
  - Jumps to target address (ProcedureLabel)
  - Address of following instruction put in $ra

- Procedure return: jump register

```
jr $ra
```

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Example Routine Call/Return



#caller

…

…

#call subroutineA

jal *subroutineA*

…

#subroutine A

***subroutineA*:** …

…

…

#return

jr $ra

Update PC based on label; Set $ra

Fetch $ra to update PC

Procedure Calling

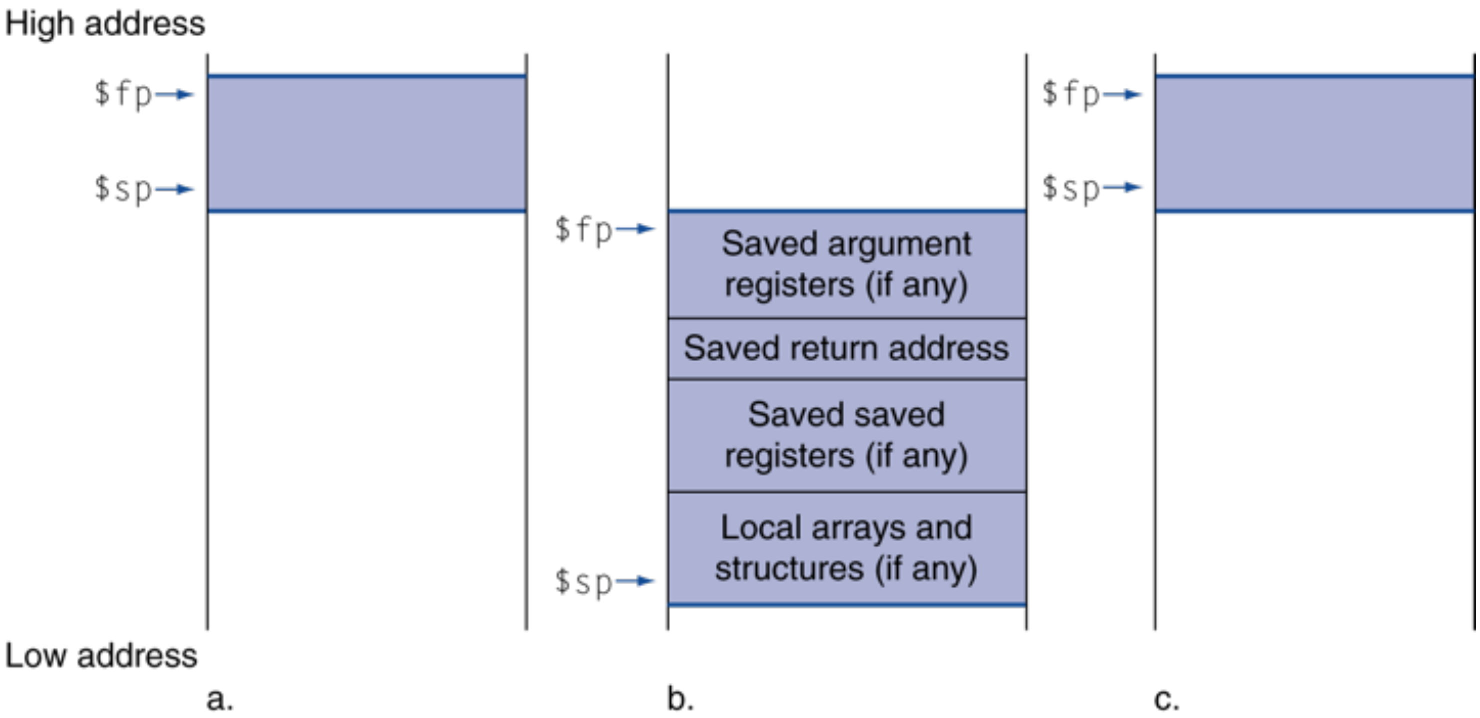
§2.8 Supporting Procedures in Computer Hardware

- Steps required
  1. Prepare and pass arguments to callee ✓
  2. Transfer control to callee procedure ✓
     - jal
  3. Acquire storage for procedure←
  4. Perform procedure's operations
  5. Pass result to caller ✓
  6. Return to place of call ✓
     - jr

Runtime Stack

- Memory for local data and bookkeeping
  - STACK: LIFO
  - A stack is a data structure with at least two operations:
    - Push: put a value on the top of the stack
    - Pop: remove an item from the top of the stack.
  - Create and push an activation record onto runtime stack for each procedure call
- Historically, stacks grow from higher to lower address (push)

Runtime Stack


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

- Stack grows and shrinks for procedure calls and returns
  - $sp ($29): stack top
  - $fp ($30): frame pointer

Basic Rules

- Stack (memory) usage rules:
  - Every thing you push onto the stack, you must pop from the stack
    - Stack pointer $sp: callee maintained
  - Never touch anything on the stack that does not belong to you
    - Frame / activation record of current procedure: between $fp and $sp

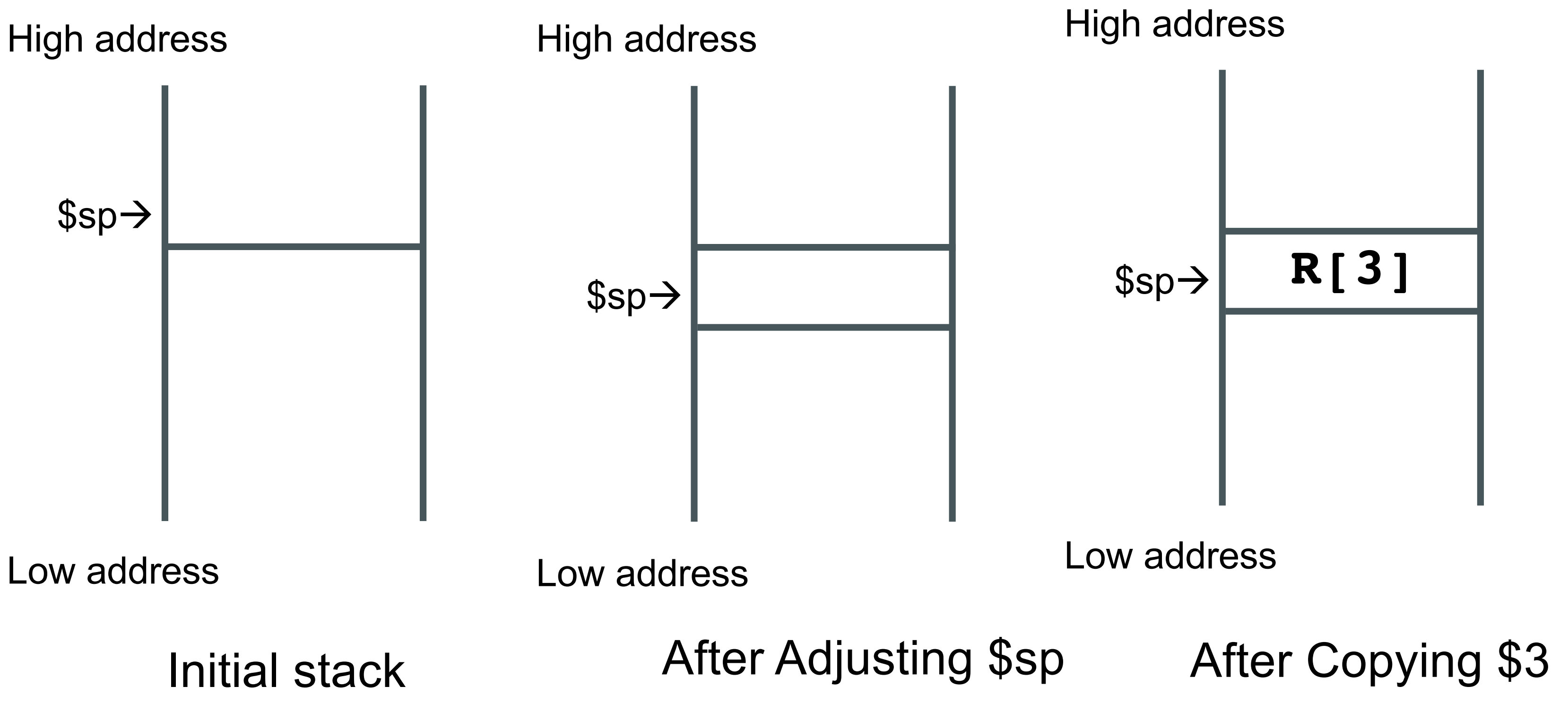
Push and Pop Operations

The MIPS has no specialized push and pop instructions (Other processors do).

- Instead the stack is implemented using the stack pointer register $sp ($29), lw and sw
  - Update stack pointer to grow/shrink
  - Use data moving instructions to copy values into/out of stack

Push in MIPS

```
#stack.push($3)
push:  addi $sp, $sp, -4  # decrement stack pointer by 4
       sw  $3, 0($sp)   # move the contents of $3 to stack top
```



Pop in MIPS

```
#$3 = stack.pop()
pop: lw   $3, 0($sp)       # Copy from stack to $r3
     addi $sp, $sp, 4      # Increment stack pointer by 4
```

Local Data Example

- C code:

```
int func (int n)
{
  int A[12];

  …

//assume no other
//stack usage

  return A[0];
}
```

- MIPS:

```
func:
        #make space for A
        addi $sp, $sp, -48
        …
        #A starts at stack top
        #set return value
        lw $v0, 0($sp)
        #pop local data off stack
        addi $sp, $sp, 48
        #return
        jr $ra
```

Runtime Stack

- Multiple activation records in stack
  - LIFO: same order as the lifetime of caller/callee procedures
- Each activation record
  - Local data
  - Saved status: bookkeeping/resource sharing
- How should we deal with shared registers?
  - Most conservative choice: always save and restore

Leaf Procedure Example

- C code:

```
int leaf_example (int g, int h, int i, int j)
{

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

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0
  - Result in $v0

Most conservative assumption: save and restore any register we update (except for $v registers)

Leaf Procedure Example

```
leaf_example:
    addi $sp, $sp, -12          # make space in the stack for 3 words
    sw   $t1, 8($sp)            # save/push $t1 before updating it
    sw   $t0, 4($sp)            # save/push $t0 before updating it
    sw   $s0, 0($sp)            # save/push $s0 before updating it
    add  $t0,$a0,$a1            # procedure body
    add  $t1,$a2,$a3            # procedure body
    sub  $s0,$t0,$t1            # procedure body
    add  $v0,$s0,$zero          # set return result
    lw   $s0,0($sp)             # restore $s0
    lw   $t0,4($sp)             # restore $t0
    lw   $t1,8($sp)             # restore $t1
    addi $sp,$sp,12             # adjust stack to pop 3 words
    jr   $ra                    # jump back to caller
```

- We are saving and restoring everything we change … ☹

Shared Data Registers

- Cover our tracks: convention to divide the job between caller and callee
  - Avoid unnecessary savings
- $s0-$s7 the saved registers, these registers should be unchanged after a function call
  - Callee must save and restore if use any

`"Callee Save"`

- $t0-$t9 these are temporaries, are are not necessarily preserved across function calls
  - Callee can override; caller needs to save and restore if use any across function calls

`"Caller Save"`

Leaf Procedure Example

```
leaf_example:
    addi $sp, $sp, -12          # make space in the stack for 3 words
    ~~sw   $t1, 8($sp)~~          # save/push $t1 for use afterwards
    ~~sw   $t0, 4($sp)~~          # save/push $t0 for use afterwards
    sw   $s0, 0($sp)            # save/push $s0 for use afterwards
    add  $t0,$a0,$a1            # procedure body
    add  $t1,$a2,$a3            # procedure body
    sub  $s0,$t0,$t1            # procedure body
    add  $v0,$s0,$zero          # set return result
    lw   $s0,0($sp)             # restore $s0
    ~~lw   $t0,4($sp)~~           # restore $t0
    ~~lw   $t1,8($sp)~~           # restore $t1
    addi $sp,$sp,12             # adjust stack to pop 3 words
    jr   $ra                    # jump back to caller
```

- MIPS convention: only need to save/restore $s0 as a callee

Leaf Procedure Example

- MIPS code:

MIPS convention: save and restore "saved" registers

```
leaf_example:
  addi $sp, $sp, -4
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero
  lw   $s0, 0($sp)
  addi $sp, $sp, 4
  jr   $ra
```

Save $s0 on stack
(“push stack”)

Procedure body

Result

Restore $s0
(“pop stack”)

Return

Register Conventions

- Caller preserved registers
  - Return address: $ra (reg 31)
  - Arguments: $a0, $a1, $a2, $a3 (reg's 4 – 7)
  - Return value: $v0, $v1(reg's 2 and 3)
  - Temporaries: $t0, $t1, … , $t7, $t8, $t9 (reg's 8 – 15,24,25)
- Callee preserved registers
  - Local variables: $s0, $s1, … , $s7 (reg's 16 – 23)
  - Stack/frame pointer: $sp, $fp (reg 29,30)
- Only need to save used registers
- A subroutine can be both caller and callee (non-leaf)