Procedures and addressing

Andrea Janes

Content

- **Using Procedures**
- Nested Procedures
- Procedure Frame
- To Summarize Procedure Calls
- More on Addressing

Java procedures (well … methods ☺)

```
main() {
 int i,j,k,m;
 i = mult(j,k);
 m = mult(i,i);
}
/* really dumb mult function */
int mult (int mcand, int mlier){
 int product;
 product = 0;
 while (mlier > 0)  {
  product = product + mcand;
  mlier = mlier -1; }
 return product;
}
```

(1) What information must compiler / programmer keep track of?

(2) What instructions can accomplish this?

Importance of procedures

What is :

set of instructions (a subroutine) performing a definite task completely independent from the main program flow with which communicates using input values and returning outputs

Why:

- helps in structuring the program
- a procedure can be reused

Freie Universität Bozen
Libera Università di Bolzano
Free University of Bozen · Bolzano

Problem #1: Function Call Bookkeeping

- Registers play a major role in keeping track of information for function calls.
- **Register conventions:**
  - Return address `$ra`
  - Arguments `$a0, $a1, $a2, $a3`
  - Return value `$v0, $v1`
  - Local variables `$s0, $s1, … , $s7`
- The stack is also used.
- More on this later.

Steps to implement calls

Steps : a procedure execution need the program to perform the following steps

- place parameters in places where the procedure can access them
- transfer control to the procedure
- acquire the storage resources needed for the procedure
- perform the desired task
- place the result in a place where calling program can access it
- return the control to the point of origin

Register usage for the calls

Remember! Registers used:

- $a0 - $a3: four argument register used to pass parameters
- $v0 - $v1: two value register used to return values
- $ra: return address register used to come back to starting point
- $sp: stack pointer, base of an array used to save the registers needed by the call
- $fp: frame pointer, used to point to the first word of the frame (we will see ..) of a procedure

… and if not enough? We will see

A possible approach

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
int sum(int x, int y) {
    return x+y;
}
```

MIPS

address

```
1000 add   $a0,$s0,$zero   # x = a
1004 add   $a1,$s1,$zero   # y = b
1008 addi  $ra,$zero,1016 #$ra=1016
1012 j     sum             #jump to sum
1016 ...
2000 sum: add $v0,$a0,$a1
2004 jr    $ra    # new instruction
```

Make the common case fast!

- Single instruction to jump and save return address: jump and link (jal)
- Before:

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum #go to sum
```

- After:

```
1012 jal sum  # $ra=1016,go to sum
```

- Why have a jal? Make the common case fast: functions are very common.

A possible solution: jal+jar (1/2)

- Syntax for **`jal`** (jump and link) is same as for **`j`** (jump):

  `jal label`

- **`jal`** should really be called **`laj`** for "link and jump":
  - Step 1 (link): Save address of next instruction into **`$ra`** (Why next instruction? Why not current one?)
  - Step 2 (jump): Jump to the given label

A possible solution: jal+jar (2/2)

- Syntax for **`jr`** (jump register):

  ```
  jr   register
  ```

- Instead of providing a label to jump to, the **`jr`** instruction provides a register which contains an address to jump to.
- Only useful if we know exact address to jump to: rarely applicable.
- Very useful for function calls:
  - **`jal`** stores return address in register ($ra)
  - **`jr`** jumps back to that address

Summary of the new instructions

`jal address`

1. save the address of next instruction in reg. $ra,
2. unconditional jump to instruction at target and allowing a procedure return to be simply: jr $ra

Opcode (6)	Target (26)

`jr rs`

unconditional jump to instruction whose address is in register $rs

Preservation of variables

- **Preserving variables**: variables used by procedures are stored in a portion of the memory called the stack whose base address is saved in register $sp
- **The stack**: LastInFirstOut structure.

  Conventionally it grows from high address values to low, i.e.,
  - adding values (push) means pointing to a lower address
  - extracting values (pop) means pointing to an higher address

Example of use

- Example: let’s turn a simple arithmetic expression into a procedure

C Code:

$a0 $a1 $a2 $a3

$s0

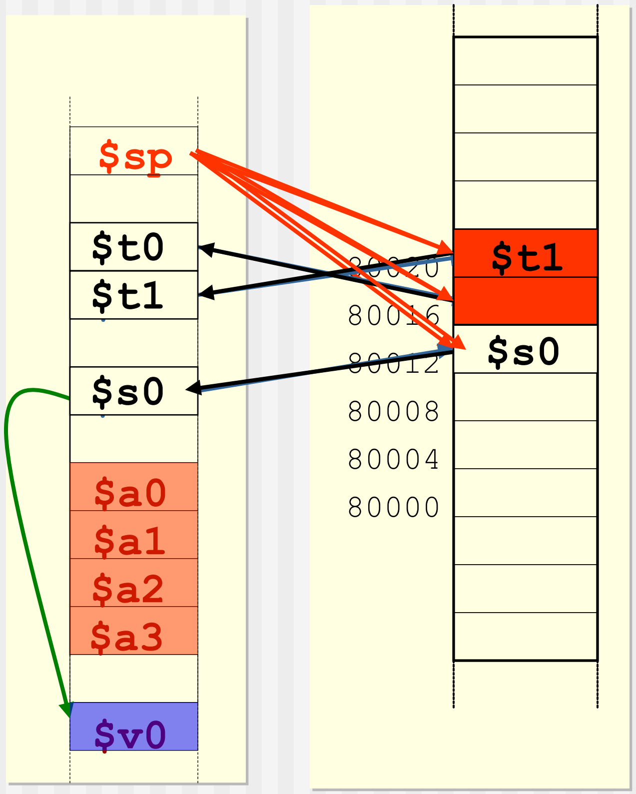
```
int leaf_example( int g, int h, int i, intj)
{ int f;
  f = (g + h) - (i + j) ;
  return f;
}
```

Translation into assembly code

- Answer:

MIPS Code:

```
leaf_example:
    sub  $sp, $sp, 12
    sw   $t1, 8($sp)
    sw   $t0, 4($sp)
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    lw   $t0, 4($sp)
    lw   $t1, 8($sp)
    add  $sp, $sp, 12
    jr   $ra
```



On the registers to preserve

- Do all registers have to be preserved?

NO! By convention …

  - $t0 - $t9: need not to be preserved in a procedure call
  - $s0 - $s9: saved registers - must be preserved in a procedure call

In the previous Example this let us omit 4 instructions. Which ones?

Content

- *Using Procedures*
- **Nested Procedures**
- Procedure Frame
- To Summarize Procedure Calls
- More on Addressing

Nested Procedures (1/2)

- Leaf Procedures

  The above example represents a “leaf” procedure, i.e., a procedure that does not call any other procedure.

- Nested procedures are procedures invoking other procedures before the return.

Nested Procedures (2/2)

- Attention

  Since procedures operates as independent entities, registers not saved ($t0 - $t9, *but also $a0 - $a3, and $ra*), may conflict.

- Solution

  Save everything into the stack.

Example of Nesting (1/2)

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

- Something called **`sumSquare`**, now **`sumSquare`** is calling **`mult`**.
- So there is a value in **`$ra`** that **`sumSquare`** wants to jump back to, but this will be overwritten by the call to **`mult`**.
- Need to save **`sumSquare`** return address before call to **`mult`**.

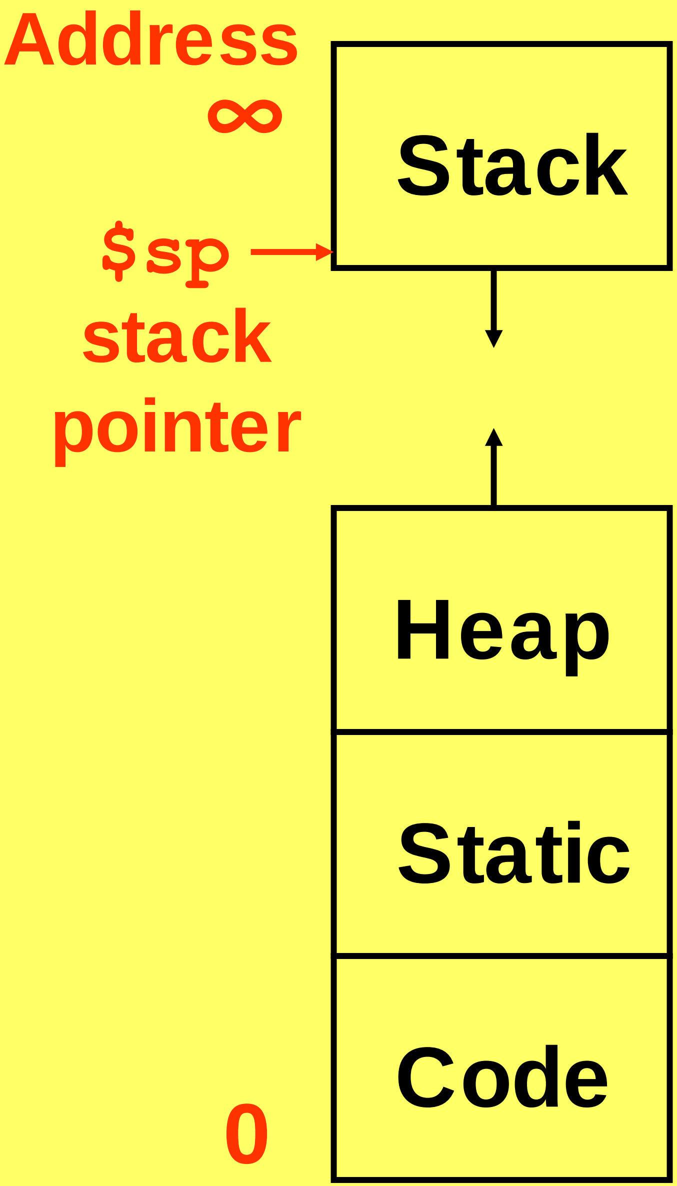
Example of Nesting (2/2)

In general, may need to save some other info in addition to $ra.

When a Java program is run, there are 3 important memory areas allocated:

- **Stack**: Space to be used by procedure during execution; this is where we can save register values
- **Heap**: Variables declared dynamically
- **Static**: Variables declared once per program, cease to exist only after execution completes

Java memory allocation



Space for saved procedure information

Explicitly created space, e.g., new

Variables declared once per program

Program

A look at the code (1/2)

- Example: let's represent a procedure calculating factorial

C Code:

$a0

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

A look at the code (2/2)

- Answer: MIPS Code

```
fact:
    sub $sp, $sp, 8        # Adjust stack to host 2 args
    sw  $ra, 4($sp)        # Save the result address
    sw  $a0, 0($sp)        # Save arg n
    slti $t0, $a0, 1       # n < 1 ?
    beq $t0, $zero, L1     # … if not jump to L1
    add $v0, $zero, 1      # Return 1
    add $sp, $sp, 8        # Adjust the stack
    jr  $ra                # Return to instruction after jal
L1: sub $a0, $a0, 1        # Gets n-1 ( in case n≥1)
    jal fact               # Call fact with arg n-1 and save
                           # address of next instr. in $ra
    lw  $a0, 0($sp)        # return from jal: restore n and ..
    lw  $ra, 4($sp)        # .. restore return addr
    add $sp, $sp, 8        # adjust stack to pop 2 items
    mul $v0, $a0, $v0      # return n*fact(n-1)( NOTE mul inst.)
    jr  $ra                # rturn to the caller
```

Comment on the code

Note that

Starting from a given **`$sp`** at each cycle it is decreased (**`sub $sp, $sp, 8`**) to store the variables (registries) defined in the cycle, and increased (**`add $sp, $sp, 8`**) when variables necessary for the cycle have been loaded (**`lw $a0, 0($sp)`**; **`lw $ra, 4($sp)`**) and the cycle is executed, thus making available the words necessary for next cycle.

Proposed exercise (1/2)

- So we have a register $sp which always points to the last used space in the stack.
- As we said, to use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```
int sumSquare(int x, int y) {
 return mult(x,x)+ y;
}
```

Proposed exercise (2/2)

```
sumSquare:
        addi $sp,$sp,-8 #space on stack
        sw $ra, 4($sp)      # save ret addr
        sw $a1, 0($sp)      # save y
        add $a1,$a0,$zero # mult(x,x)
        jal mult            # call mult
        lw $a1, 0($sp)   # restore y
        add $v0,$v0,$a1 # mult()+y
        lw $ra, 4($sp)   # get ret addr
        addi $sp,$sp,8 # restore stack
        jr $ra
```

Summary of the Steps for a Call

1. Save necessary values onto stack.
2. Assign argument(s), if any.
3. `jal` call
4. Restore values from stack.

Summary of the Rules for Calls

- Called with a **`jal`** instruction, returns with a **`jr $ra`**
- Accepts up to 4 arguments in **`$a0`**, **`$a1`**, **`$a2`**, and **`$a3`**
- Return value is always in **`$v0`** (and if necessary in **`$v1`**)
- Must follow **register conventions**, even in functions that only you will call!

Content

- *Using Procedures*
- *Nested Procedures*
- **Procedure Frame**
- To Summarize Procedure Calls
- More on Addressing

Procedure Frame (1/3)

- Allocating space for new data

  The stack is used also to store variables local to the procedure that do not fit into registries (local array or structures)

- Frame pointer ($fp)

  Is a pointer to the first word of the frame of a procedure

  Unlike stack pointer ($sp), that can change during a procedure, it offers a a stable base register within a procedure for local memory references

Procedure Frame (2/3)

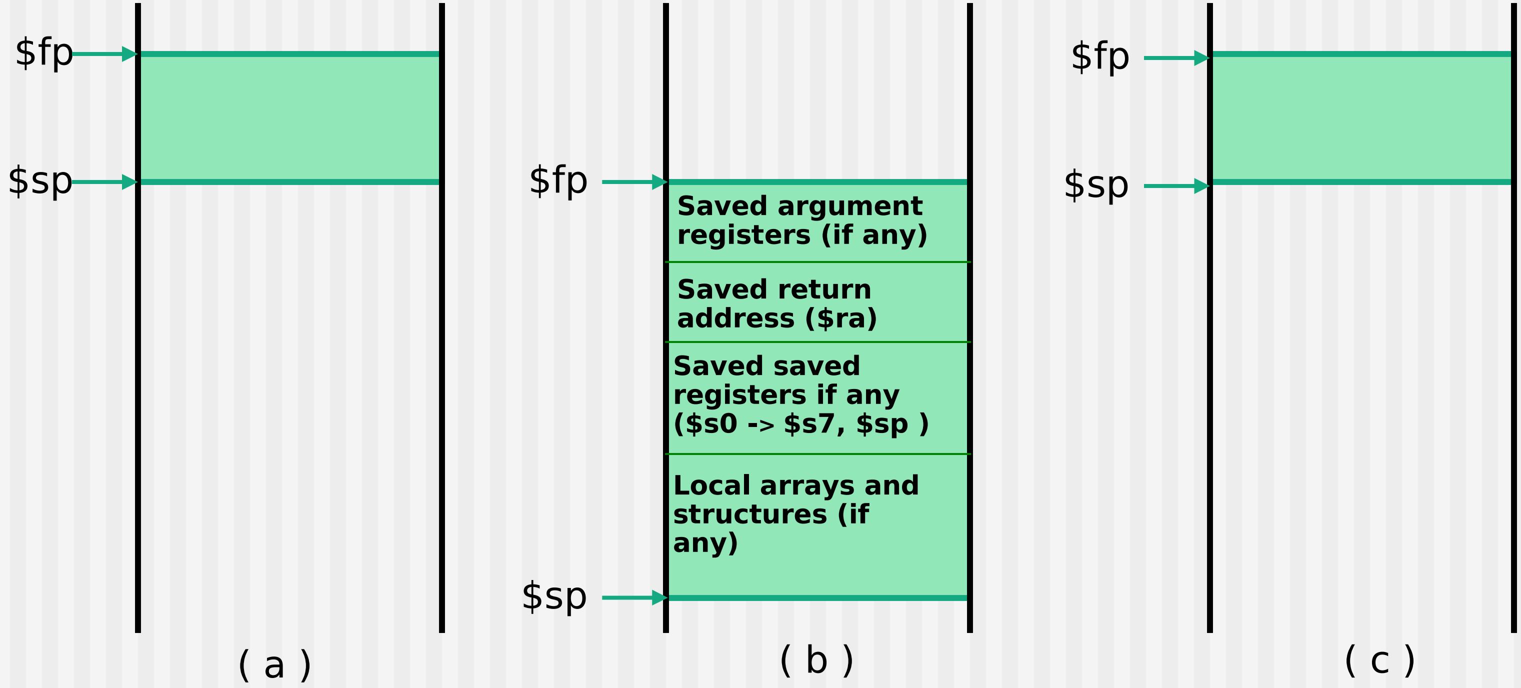
- More variables to be used (see slide 5)?

  The MIBS convention is to place extra parameters in the stack just above the frame pointer.

  The procedure then expects the first parameters to be in \$a0 -> \$a3, the others in memory addressable via the frame pointer \$fp.

Procedure Frame (3/3)

Stack allocation (a) before, (b) during, and (c) after a procedure call.



Content

- *Using Procedures*
- *Nested Procedures*
- *Procedure Frame*
- **To Summarize Procedure Calls**
- More on Addressing

Register Conventions

- We have seen a few conventions on the 32 available registers
- Such conventions have to be followed, even if you are the only programmer

Register Conventions (1/5)

- Caller: the calling function
- Callee: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may be changed.

Register Conventions (2/5)

- $0: No Change. Always 0.
- $v0-$v1: **Change**. These are expected to contain new values.
- $a0-$a3: **Change**. These are volatile argument registers.
- $t0-$t9: **Change**. That’s why they’re called temporary: any procedure may change them at any time.

Register Conventions (3/5)

- \$s0-\$s7: No Change. Very important, that's why they're called saved registers. If the callee changes these in any way, it must restore the original values before returning.
- \$sp: No Change. The stack pointer must point to the same place before and after the jal call, or else the caller won't be able to restore values from the stack.
- \$ra: **Change**. The `jal` call itself will change this register.

Register Conventions (4/5)

What do these conventions mean?

A. If function A calls function B, then function A must save any temporary registers that it may be using onto the stack before making a `jal` call.

B. Function B must save any S (saved) registers it intends to use before garbling up their values

C. Remember: Caller/callee need to save only temporary/saved registers they are using, not all registers.

Register Conventions (5/5)

- Note that, if the *callee* is going to use some s registers, it must:
  - save those s registers on the stack
  - use the registers
  - restore s registers from the stack
  - **`jr $ra`**
- With the temp registers, the callee doesn't need to save onto the stack.
- Therefore the *caller* must save those temp registers that it would like to preserve though the call.

Other Registers

- $at: may be used by the assembler at any time; unsafe to use
- $k0-$k1: may be used by the kernel at any time; unsafe to use
- $gp: don’t worry about it
- $fp: we have seen
- Note: Feel free to read up on $gp and $fp in Appendix A: you can write perfectly good MIPS code without them.

Example: Compile This (1/5)

```
main() {
 int i,j,k,m; /* i-m:$s0-$s3 */
 i = mult(j,k); ... ;
 m = mult(i,i); ...
}
int mult (int mcand, int mlier){
 int product;
 product = 0;
 while (mlier > 0)  {
  product += mcand;
  mlier -= 1; }
 return product;
}
```

Example: Compile This (2/5)

```
__start:
  add $a0,$s1,$0          # arg0 = j
  add $a1,$s2,$0          # arg1 = k
  jal mult                # call mult
  add $s0,$v0,$0          # i = mult()
  ...
  add $a0,$s0,$0          # arg0 = i
  add $a1,$s0,$0          # arg1 = i
  jal mult                # call mult
  add $s3,$v0,$0          # m = mult()
  ...
  done
```

Example: Compile This (3/5)

- Notes:
  - **`main`** function ends with **`done`**, not **`jr $ra`**, so there's no need to save **`$ra`** onto stack
  - all variables used in **`main`** function are saved registers, so there's no need to save these onto stack

Example: Compile This (4/5)

```
mult:
     add  $t0,$0,$0         # prod=0
Loop:
     slt  $t1,$0,$a1        # mlr > 0?
     beq  $t1,$0,Fin        # no=>Fin
     add  $t0,$t0,$a0       # prod+=mc
     addi $a1,$a1,-1        # mlr-=1
     j     Loop                  # goto Loop
Fin:
    add  $v0,$t0,$0         # $v0=prod
    jr   $ra                # return
```

Example: Compile This (5/5)

- No `jal` calls are made from `mult` and we don’t use any saved registers, so we don’t need to save anything onto stack
- Temp registers are used for intermediate calculations (could have used s registers, but would have to save the caller’s on the stack.)
- $a1 is modified directly (instead of copying into a temp register) since we are free to change it
- Result is put into $v0 before returning

Things to Remember (1/2)

- Functions are called with `jal`, and return with `jr $ra`.
- The stack is your friend: Use it to save anything you need. Just be sure to leave it the way you found it.

Things to Remember (2/2)

- Instructions we know so far

  Arithmetic: **`add, addi, sub, addu, addiu, subu, sll`**

  Memory: **`lw, sw`**

  Decision: **`beq, bne, slt, slti, sltu, sltiu`**

  Unconditional Branches (Jumps): **`j, jal, jr`**

- Registers we know so far
  - All of them!

To Summarize - Operands

MIPS operands		
Name	**Example**	**Comments**
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

To Summarize – Operations

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC+4+100	Equal test; PC-relative branch
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) go to PC+4+100	Not equal test; PC-relative
	set on less then	slt $t0, $s1, $s2	if ($s1 < $s2) set $t0 = 1	True if inequality holds or false
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC+4; go to 10000	store next instruction in $ra and jump

Content

- *Using Procedures*
- *Nested Procedures*
- *Procedure Frame*
- *To Summarize Procedure Calls*
- **More on Addressing**

Freie Universität Bozen
Libera Università di Bolzano
Free University of Bozen

Addressing modes