

Inf2C - Computer Systems

Lectures 3-4

Assembly Language

Boris Grot

School of Informatics
University of Edinburgh



Announcements: Labs

- Purpose:
 - prep for courseworks by getting familiar with the tools
 - reinforce class concepts
- Regular schedule (starting next week):
 - Mon: 2-3pm, 5-6pm in FH 3.D02
 - Wed: 3-4pm in FH 3.D01
 - Thr: 11am-12pm in FH 1.B31
- Extras (whenever coursework is out)
 - Tue: 11am-12pm in FH 3.D01
 - Wed: 2-3pm in FH 3.D01
 - Fri: 1-2pm in FH 3.D01
- Drop-in format
 - You may attend any & all lab sessions!
 - Demonstrator (Kuba or Arpit) on-hand to answer questions



Announcements: Other

- Tutorial 1 running next week
 - Attempt the problems ahead of time
 - Come in with specific questions
 - You'll get out what you put in
- Online discussion forum: ASK
 - URL to be confirmed

Previous lecture

- Representing numbers
 - Binary encoding
 - Negative numbers
 - Floating point numbers
 - Characters & strings
- Other things
 - Binary addition
 - Shifting
 - Hex



Lectures 3-4: MIPS instructions

- Motivation: Learn how a processor's 'native' language looks like
- We will examine the MIPS ISA
 - MIPS: Microprocessor without Interlocked Pipeline Stages – a real-world ISA used by many different processors since the 80s.
 - Regular and representative → great for learning!
- ISA reference can be downloaded from:
 - http://www.cs.wisc.edu/~larus/HP_AppA.pdf



Outline

- Instruction set
- Basic arithmetic & logic instructions
- Processor registers
- Getting data from the memory
- Control-flow instructions
- Method calls

Processor instructions

- **Instruction set architecture** (ISA): the interface between the software and the hardware
 - Collection of all machine instructions recognized by a particular processor
 - Also, privilege levels, memory management, etc.
- The ISA abstracts away the hardware details from the programmer
 - Similar to how an object hides its implementation details from its users
 - Enables multiple implementations (called **microarchitectures**) of the same ISA.



Assembly language

- Instructions are represented internally as binary numbers
 - Example: 00000011110100100000001000001011
 - For a human, very hard to make out which instruction is which (sequence of 32-64 bits!)
- **Assembly language**: symbolic representation of machine instructions

MIPS assembly: a simple example

High-level language (HLL): $a[0] = b[0] + 10$

MIPS assembly language:

```
lw    r4,0(r2)    # get the value of b[0] from memory
                    # and store it in register r4
add   r5,r4,10     # compute b[0]+10 and store into r5
sw    r5,0(r1)     # store r5 into a[0]
```

Things to notice:

- Separate instructions to **access** data (in memory) & **operate** on it
 - Cannot operate on memory directly
- All instructions have similar format



MIPS arithmetic & logical operations

- Data processing instructions look like:
operation destination, 1st operand, 2nd operand
add a,b,c $a = b + c$
sub a,b,c $a = b - c$
- Bit-wise logical instructions: and, or, xor
- Shift instructions:
sll a,b,shamt $a = b \ll \text{shamt}$
srl a,b,shamt $a = b \gg \text{shamt}$ (logical)
sra a,b,shamt $a = b \gg \text{shamt}$ (arithmetic)

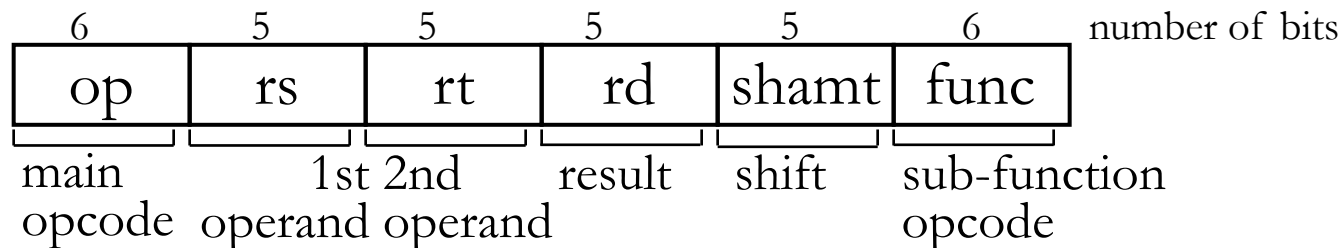
Registers

- ISA places restrictions on instruction operands
 - How many operands and where they come from
- MIPS processors operate on registers only
 - **Registers** are storage locations inside the processor that hold program variables
- Registers are sized to contain machine's **word**
 - 32 or 64 bits common today
- Processors have relatively few registers
 - MIPS has 32
 - x86 has 8-16



MIPS instruction example

- Assembly: **add \$s1, \$s2, \$s3** # \$s1 = \$s2 + \$s3
- Binary (R-format – used for arithmetic instructions):



0	18	19	17	0	32 ₁₀	add \$s1, \$s2, \$s3
0	18	9	8	0	34 ₁₀	sub \$t0, \$s2, \$t1

Each and every assembly instructions translates to exactly 1 machine instruction (no choice or ambiguity)

More on MIPS registers

- Most registers are available for any use
 - with a few important exceptions
- Program (C/Java) variables held in regs \$s0-\$s7
- Temporary variables: \$t0-\$t9
- Registers with special roles
 - Register 0 (**\$zero**) is hardwired to 0
 - **Program Counter (PC)** holds address of next instruction to be executed (it is not a general purpose registers)
 - Other special-purpose registers exist



Immediate operands

- What if we need to operate on a constant?
 - Common in arithmetic operations, loop index updates (e.g., `i++`), or even character manipulations (e.g., changing the case of an ASCII character)
- MIPS has dedicated instructions with one constant (**immediate**) operand
 - e.g. `addi $r1, $r1, 1 # r1=r1+1`
- General form: `opi $r1, $r2, n`

Loading a constant operand

- Load a (small) constant into a register:

- Constant is 16 bits and is signed

`addi $s0,$zero,n` # $\$s0=n$ ($\$s0_{31-16}=\text{sign ext}$; $\$s0_{15-0}=n$)

- What if need a larger/smaller constant?
- Assembler **pseudo-instruction** `li reg,constant`
 - Translated into 1 instruction for immediates $< 16\text{bits}$ and into more instructions for more complicated cases (e.g. 2 instructions for a 32-bit immediate)

`lui $s1,n1` # $\$s1_{31-16}=n1$; $\$s1_{15-0}=0$
`ori $s1,$s1,n2` # $\$s1_{31-16}=n1$; $\$s1_{15-0}=n2$



Getting at the data

- Programming statement:

$$g = h + A[1]$$

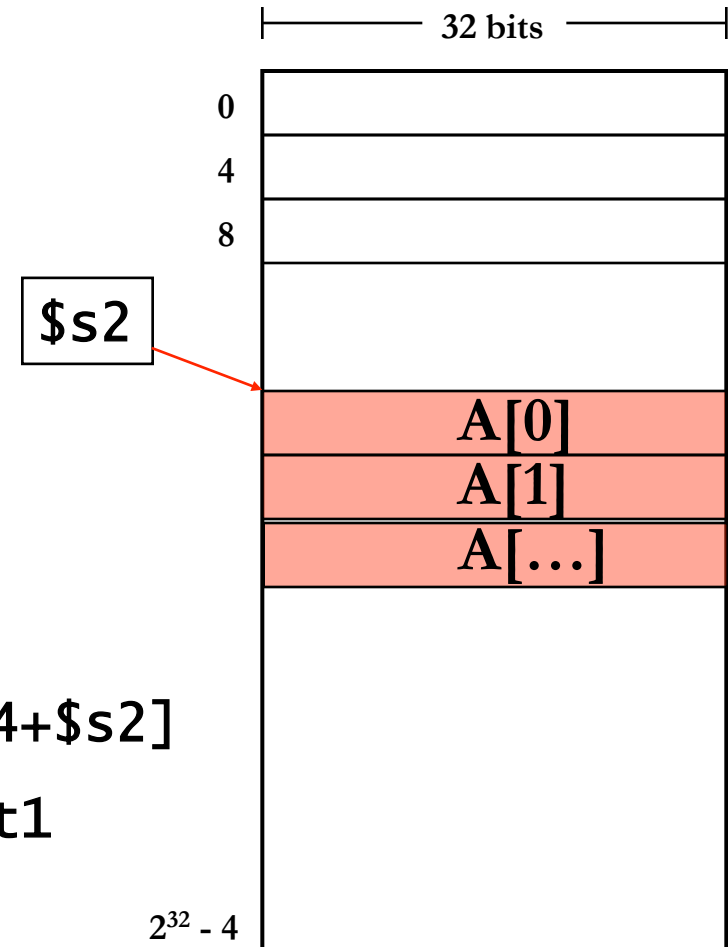
where

- h is in register \$s1
- A[0] is the first element of array A and is pointed to by \$s2

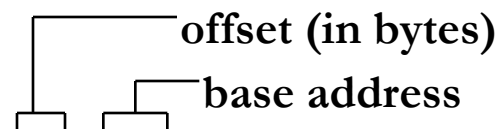
- MIPS: **offset**

```
lw  $t1, 4($s2)    # $t1=memory[4+$s2]
```

```
add $t2,$s1, $t1    # $t2 = h + $t1
```



Data-transfer instructions

- Load Word:
`lw r1,n(r2) # r1=memory[n+r2]`

- Store Word:
`sw r1,n(r2) # memory[n+r2]=r1`
- Load Byte:
`lb r1,n(r2) # r17-0= memory[n+r2]`
`r131-8= sign extension`
- Store Byte:
`sb r1,n(r2) # memory[n+r2]=r17-0`
`no sign extension`

Memory addressing

- Memory is **byte addressable**, but it is organised so that a whole word can be accessed directly
- Where can a word be stored?
 - Option 1: anywhere (**unaligned**)
 - Option 2: at an address that is a multiple of the word size (**aligned**)
 - Both options in use today: MIPS requires alignment, x86 doesn't
 - What are the trade-offs?

Memory addressing: Endianness

Given a memory address, **Endianness** tells us where to find the starting byte of a word

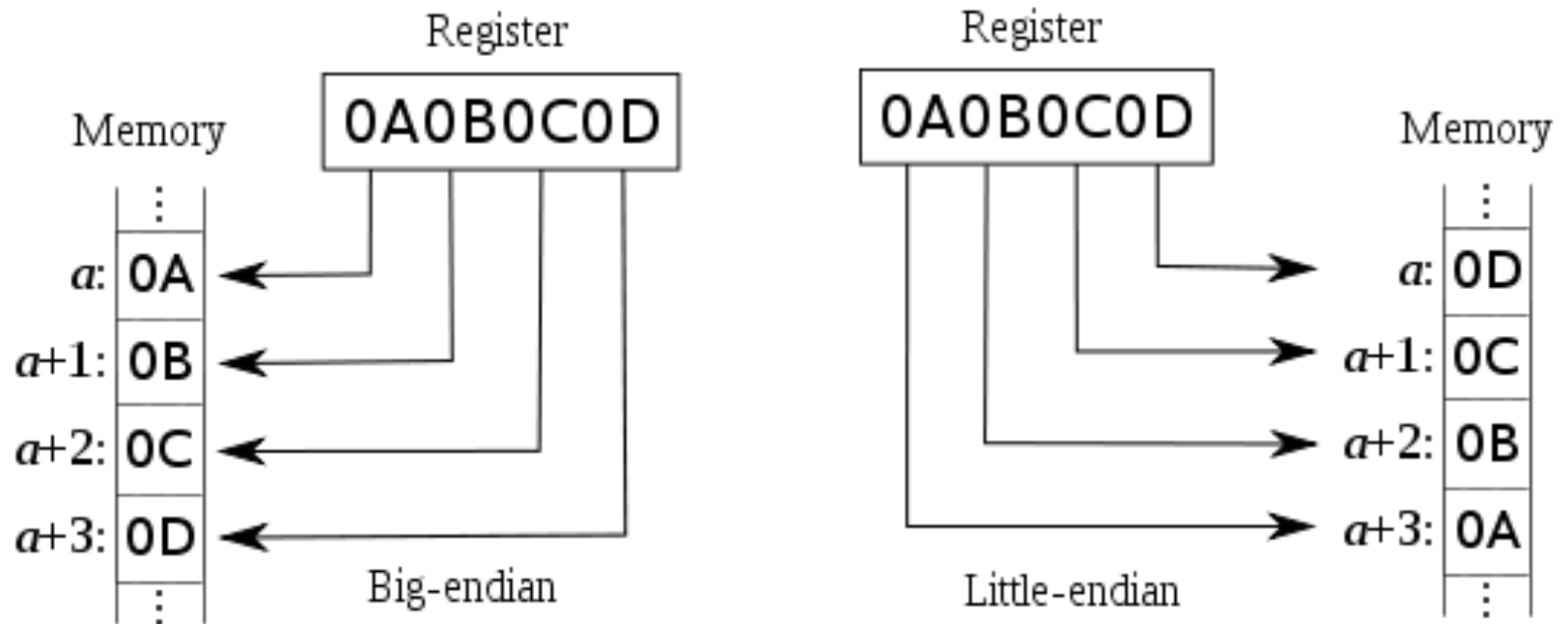
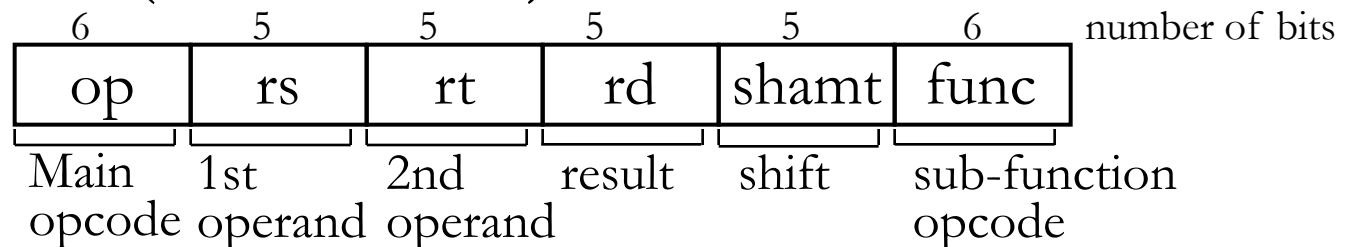


Image source: <http://en.wikipedia.org/wiki/Endianness>

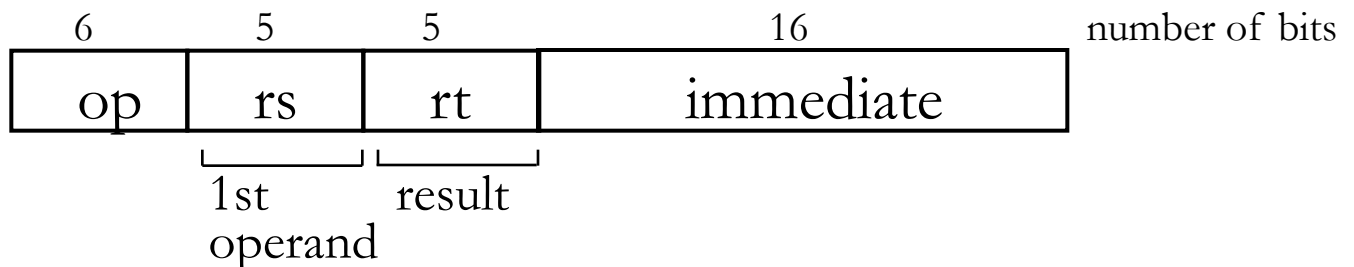
Instruction formats

- Instruction representation composed of **bit-fields**
- Similar instructions have the same format
- MIPS instruction formats:

– R-format (**add, sub, ...**)



– I-format (**addi, lw, sw, ...**)




MIPS instructions – part 2

- Last time:
 - Data processing instructions: add, sub, and, ...
 - Registers only and immediate types
 - Data transfer instructions: lw, sw, lb, sb
 - Instruction encoding
- Today:
 - Control transfer instructions

A simple program to swap array elements

```
1  int main(void) {
2      int size = 6;
3      int v[] = {1, 10, 2, 20, 3, 30};  // array w/ 6 elements
4
5      for (int i=0; i<size; i+=2)
6          swap(v, i);  // pass array (by reference) and index i
7  }
8
9  void swap(int v[], int idx) {
10     int temp;
11     temp = v[idx];
12     v[idx] = v[idx+1];
13     v[idx+1] = temp;
14 }
```



Swap () in MIPS

swap:

```
1      # inputs: $a0 - array base, $a1 - index
2
3      # Compute the address into the array
4      sll $t1, $a1, 2      # reg $t1 = idx * 4
5      add $t1, $a0, $t1    # reg $t1 = v + (idx*4)
6                          # $t1 holds the addr of v[idx]
7
8      # Load the two values to be swapped
9      lw  $t0, 0($t1)      # reg $t0 = v[idx]
10     lw  $t2, 4($t1)      # reg $t2 = v[idx+1]
11
12     # Store the swapped values back to memory
13     sw  $t2, 0($t1)      # v[idx] = $t2
14     sw  $t0, 4($t1)      # v[idx+1] = $t0
```



Control transfers: If structures

Java/C: `if (i!=j)`
 stmt1 → “if case”
`else`
 stmt2 → “else case”
 stmt3 → “follow through”

MIPS: “branch if equal” `beq $s1,$s2,label`

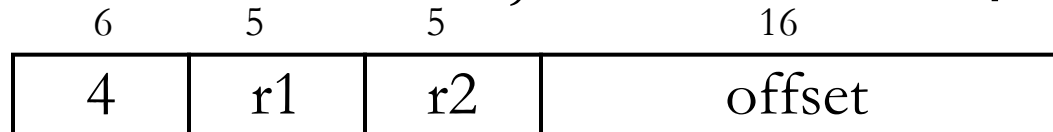
- compare value in \$s1 with value in \$s2
- if equal, branch to instruction marked label

```
        beq $s1,$s2,label2
        stmt1
        j label3    # skip stmt2
label2:  stmt2
label3:  stmt3
```



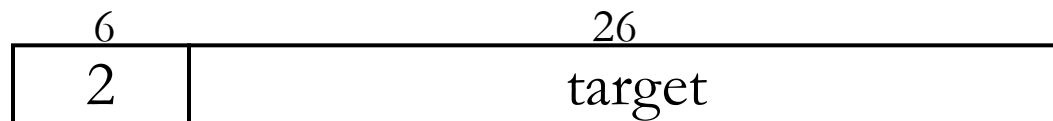
Control transfer instructions

- Conditional branches, I-format: **beq r1,r2,label**



- In assembly code label is usually a string
- In machine code, label is obtained from immediate operand as: $\text{branch target} = \text{PC} + 4 * \text{offset}$

- Similarly: **bne r1,r2,label # if r1!=r2 go to label**
- Unconditional jump, J-format: **j label**



Loops in assembly language

- Java/C: `while (count!=0) stmt`
- MIPS: `loop:`
 - `beq $s1,$zero,end # $s1 holds count`
 - `stmt`
 - `j loop # branch back to loop``end: ...`

Loops in assembly language

- Java/C: `while (flag1 && flag2) stmt`
- MIPS:

```
loop:      beq $s1,$zero,end    # $s1 holds flag1
           beq $s2,$zero,end    # $s2 holds flag2
           stmt
           j loop    # branch back to loop
end:      ...
```

Comparisons

- “Set if less than” (R-format): `slt r1, r2, r3`
 - set r1 to 1 if $r2 < r3$, otherwise set r1 to 0
- Java/C: `while (i > j) stmt`
- MIPS example:
 - assume that `$s1` contains `i` and `$s2` contains `j`

loop:

```
slt $t0, $s2, $s1    # $t0 = (j < i)
beq $t0, $zero, end  # branch if i <= j
stmt
j loop # jump back to loop
```

end: ...

MIPS Instruction Format Summary

- R-type (register to register)
 - three register operands
 - most arithmetic, logical and shift instructions
- I-type (register with immediate)
 - instructions which use two registers and a constant
 - arithmetic/logical with immediate operand
 - load and store
 - branch instructions with relative branch distance
- J-type (jump)
 - jump instructions with a 26 bit address

Practice problem:

What C code does the following piece of MIPS assembly code correspond to?

```
    slt $t0, $s1, $s2
    beq $t0, $zero, 11
    and $s3, $s1, $s2
    j 12
11: or $s3, $s2, $s1
12:
```

- (a) if ($s1 < s2$) $s3 = s2 \mid s1$; else $s3 = s1 \& s2$;
- (b) if ($s1 \leq s2$) $s3 = s2 \mid s1$; else $s3 = s1 \& s2$;
- (c) if ($s1 < s2$) $s1 = s3 \mid s2$; else $s2 = s3 \& s1$;
- (d) if ($s1 \leq s2$) $s2 = s3 \& s1$; else $s1 = s3 \mid s2$;
- (e) if ($s1 < s2$) $s3 = s1 \& s2$; else $s3 = s2 \mid s1$;

Common MIPS Arithmetic & Logical Operators

■ Integer Arithmetic

+	add
-	sub
*	mul
/	div
%	rem

■ Bit-wise logic

	or
&	and
^	xor
~	not

■ Shifts

>>	(signed)	shift-right-arithmetic
>>	(unsigned)	shift-right-logical
<<		shift-left-logical

■ Boolean

	(src1 != 0 or src2 != 0)
&&	(src1 != 0 and src2 != 0)

Common MIPS Relational Operators

■ Relational

< slt, sltu
<= sle, sleu
> sgt, sgtu
>= sge, sgeu
== seq
!= sne

C operator	Comparison	Reverse	Branch
==	seq	0	bne
!=	seq	0	beq
<	slt, sltu	0	bne
>=	slt, sltu	0	beq
>	slt, sltu	1	bne
<=	slt, sltu	1	beq

Method calls

- Method calls are essential even for a small program
- Most processors provide support for method calls
- Java/C:

```
...  
foo();  
...  
foo();  
...
```

call to foo at line L1

call to foo at line L2

```
void foo() {  
...  
return;  
}
```

where do we return to?

MIPS support for method calls

- Jumping into the method: `jal label`
 - “jump and link”:
 - set `$ra` to `PC+4`
 - set PC to label
 - Another J-format instruction
- Returning: `jr r1`
 - “jump register”: set PC to value in register `r1`

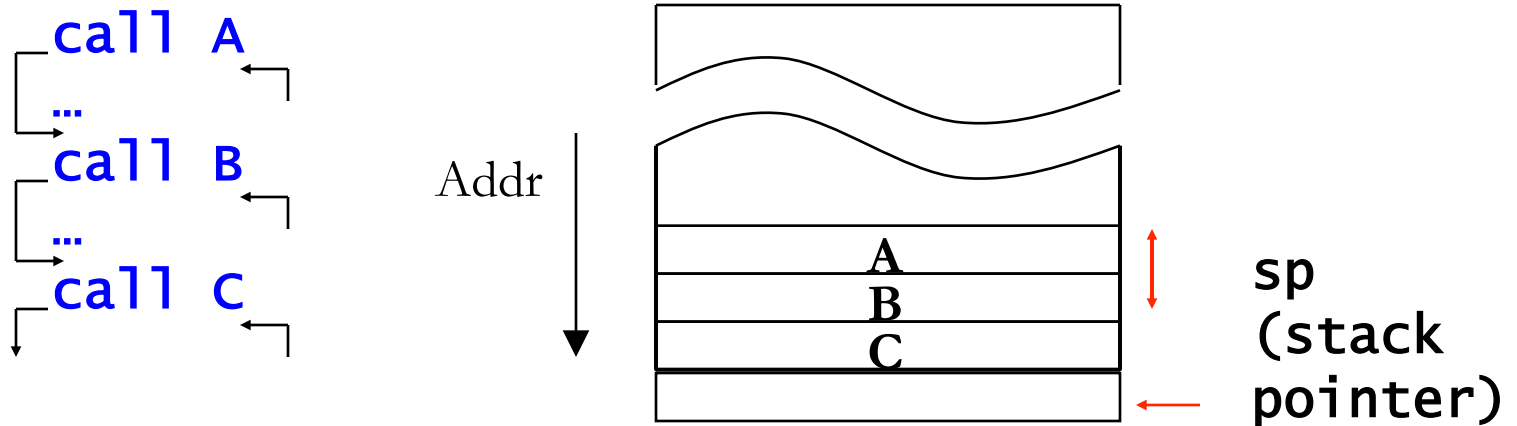
MIPS register convention on method calls

- Method parameters: \$a0 - \$a4
- Return values: \$v0, \$v1
- Regs preserved across call boundaries: \$s0 - \$s7
- Regs **not** preserved across call boundaries: \$t0 - \$t9

What about nested method calls?

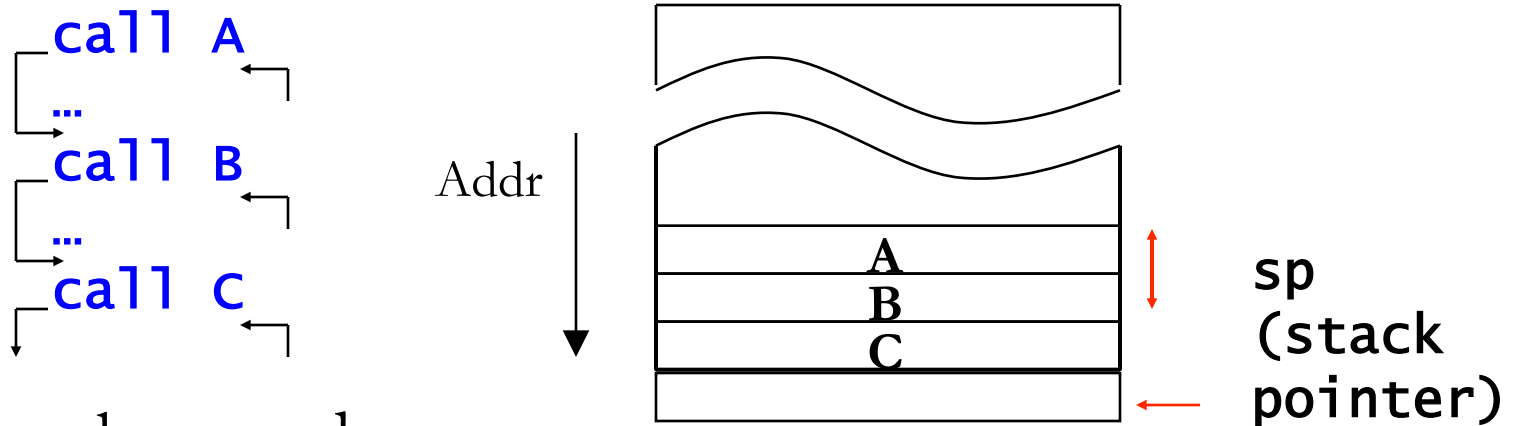
Using a stack for method calls

- Nested calls \Rightarrow must save return address & $\$t$ registers to prevent overwriting. Solution: use a stack in memory



Using a stack for method calls

- Nested calls \Rightarrow must save return address & $\$t$ registers to prevent overwriting. Solution: use a stack in memory



- to push a word:

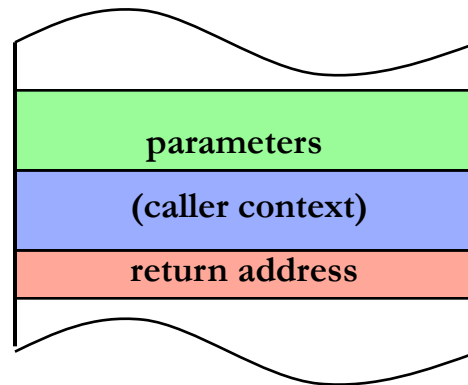
```
addi $sp,$sp,-4 # move sp down
sw    $ra,0($sp) # save r1 on top of stack
```

- to pop a word:

```
lw    $ra,0($sp) # fetch value from stack
addi $sp,$sp,4   # move sp up
```

Other uses of the stack

- Stack used to save caller's registers, so that they can be used by the callee
 - “caller save” or “callee save” convention
- Stack can also be used to pass and return parameters
 - Gets around the limited number of parameter and return value registers



Main () in MIPS

```
1  main:
2      # Initialize variable
3      la    $s0, array      # $s0: base addr of the array
4      addi  $s1, $zero, 0   # $s1: index into the array
5      addi  $s2, $zero, 6   # $s1: array size
6
7  loop:
8      move  $a0, $s0        # $a0 = $s0 (array base pointer)
9      move  $a1, $s1        # $a1 = $s1 (index)
10     jal   swap            # call swap
11
12     addi  $s1, $s1, 2      # increment the index
13     bne   $s1, $s2, loop
```

Swap () : what's missing?

1 **swap:**

2 # Compute the address into the array

3 sll \$t1, \$a1, 2 # reg \$t1 = idx * 4

4 add \$t1, \$a0, \$t1 # reg \$t1 = v + (idx*4)

5 # \$t1 holds the addr of v[idx]

6
7 # Load the two values to be swapped

8 lw \$t0, 0(\$t1) # reg \$t0 = v[idx]

9 lw \$t2, 4(\$t1) # reg \$t2 = v[idx+1]

10
11 # Store the swapped values back to memory

12 sw \$t2, 0(\$t1) # v[idx] = \$t2

13 sw \$t0, 4(\$t1) # v[idx+1] = \$t0

Swap () : complete version

```
1  swap:
2      # Compute the address into the array
3      sll $t1, $a1, 2      # reg $t1 = idx * 4
4      add $t1, $a0, $t1    # reg $t1 = v + (idx*4)
5                          # $t1 holds the addr of v[idx]
6
7      # Load the two values to be swapped
8      lw  $t0, 0($t1)      # reg $t0 = v[idx]
9      lw  $t2, 4($t1)      # reg $t2 = v[idx+1]
10
11     # Store the swapped values back to memory
12     sw  $t2, 0($t1)      # v[idx] = $t2
13     sw  $t0, 4($t1)      # v[idx+1] = $t0
14
15     jr  $ra              # return to main
```

Should an ISA be simple or complex?

- ISAs range in complexity
- MIPS is a relatively simple ISA.
 - But is that the right design choice?
- Consider the earlier example:

High-level language (HLL): $a=b+10$

Assembly language:

- MIPS:

```
lw    r4,0(r2)    # r4=memory[r2+0]
add   r5,r4,10     # r5=r4+10
sw    r5,0(r3)     # memory[r3+0]=r5
```
- x86:

```
ADDW3 (R5),(R2),10
```



CISC vs RISC ISAs

- Complex Instruction Set (CISC)
 - Appeared in early computers, including x86
 - Computers programmed in assembly → high-level language features as instructions
 - Very few registers → operands can be in memory
 - Very little memory → variable length instructions to minimize code size
- Reduced Instruction Set (RISC)
 - Appeared in the 80s. Used today in ARM, MIPS, and SPARC ISAs.
 - Compilers → Simple instructions
 - More registers → load-store architecture
 - More memory & faster clock frequency → fixed length, fixed format instructions for easy, fast decoding logic

