# Lectures 3-4: MIPS instructions

- Motivation
  - Learn how a processor's 'native' language looks like
  - Discover the most important software-hardware interface

# Outline

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

# Processor instructions

- Instruction set (IS): collection of all machine instructions recognized by a particular processor
- The instruction set abstracts away the hardware details from the programmer
  - The same way as an object hides its implementation details from its users
- Instruction Set Architecture (ISA): a generic processor implementation that recognizes a particular IS

# RISC – CISC machines

- There are many ways of defining the hardware-software interface defined by the instruction set
  - Depends on how much work the hardware is allowed to do
- RISC=Reduced Instruction Set Computer

  CISC=Complex Instruction Set Computer
- High-level language (HLL): a=b+10

  Assembly language:
  - RISC:

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

  - CISC:

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

# Assembly language

- Instructions are represented internally as binary numbers
  - Very hard to make out which instruction is which
- Assembly language: symbolic representation of machine instructions
- We use the MIPS IS, typical of a RISC processor

# Arithmetic & logical operations

- Data processing instructions look like:
  `operation destination var, 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 << shamt

`srl a,b,shamt` a = b >> shamt, logical shift

# Registers

- IS places restrictions on instruction operands
- RISC processors operate on registers only
- Registers are internal storage locations holding program variables
- Size of register equals the machine's word
- There is a relatively small number of registers present; MIPS has 32

# MIPS general-purpose registers

- Generally, any register available for any use
- Conventions exist for enabling code portability
- Java variables held in registers `$s0 - $s7`
- Temporary variables: `$t0 - $t9`
- Register 0 (`$zero`) is hardwired to 0
- Other registers with special roles
- Program Counter (PC) holds address of next instruction to be executed
  - Not one of the general purpose registers

# Immediate operands

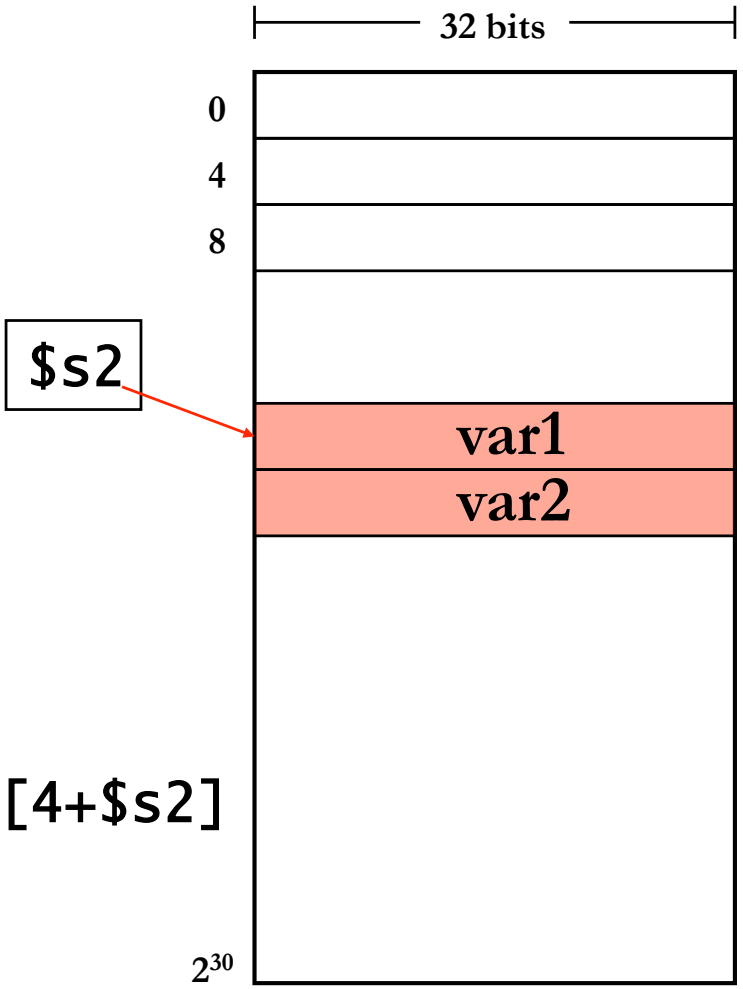
- MIPS has instructions with one constant (immediate) operand, e.g. `addi r1,r2,n  # r1=r2+n`
- Load a (small) constant into a register:

  `addi $s0,$zero,n  # $s0=n ($s0`$_{15\text{-}0}$`=n; $s0`$_{31\text{-}16}$`=0)`
- Assembler pseudo-instruction `li reg,constant`
  - Translated into 1 instruction for immediates < 16bits and to more instructions for more complicated cases e.g. for a 32-bit immediate

```
lui $s1,n1     # $s1_15-0=0;  $s1_31-16=n1
ori $s1,$s1,n2 # $s1_15-0=n2; $s1_31-16=n1
```

# Getting at the data

- Java:
```
Class MyClass {
    int var1,var2;
}
…
myObj = new MyClass()
…
temp = myObj.var2
```




- MIPS:

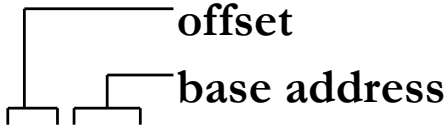
($s2 points to base of myObj)

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

offset of
var2 within myObj

# 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)   # r1_7-0= memory[n+r2]
                r1_31-8= sign extension
```

- Store Byte:

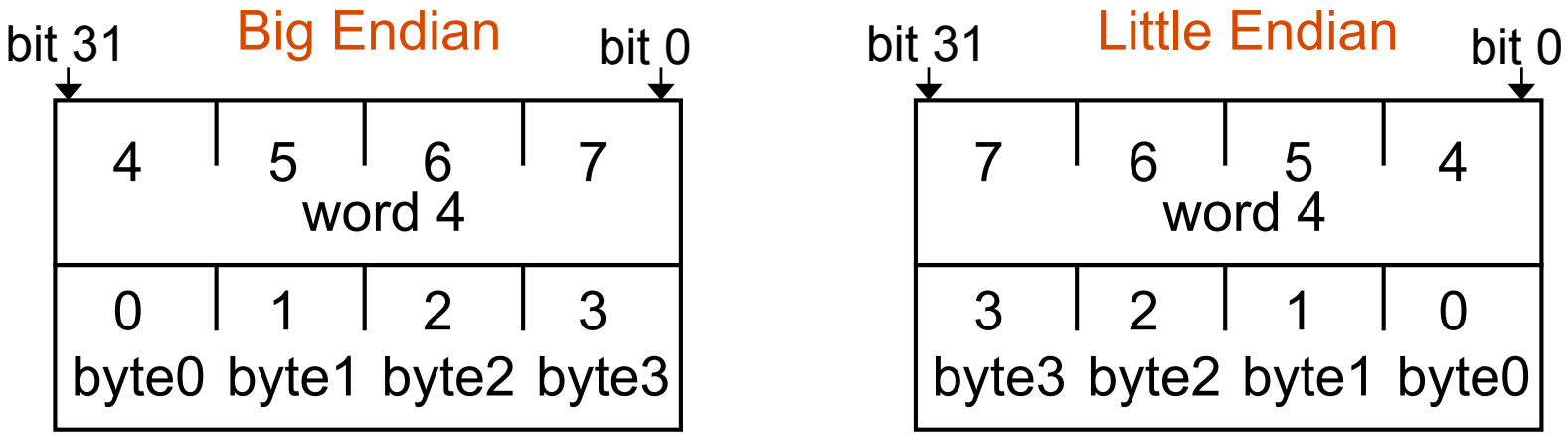
```
sb r1,n(r2)   # memory[n+r2]=r1_7-0
                no sign extension
```

# Memory addressing

- Memory is byte addressable, but it is organised so that a word can be accessed directly
- Where can a word be stored?

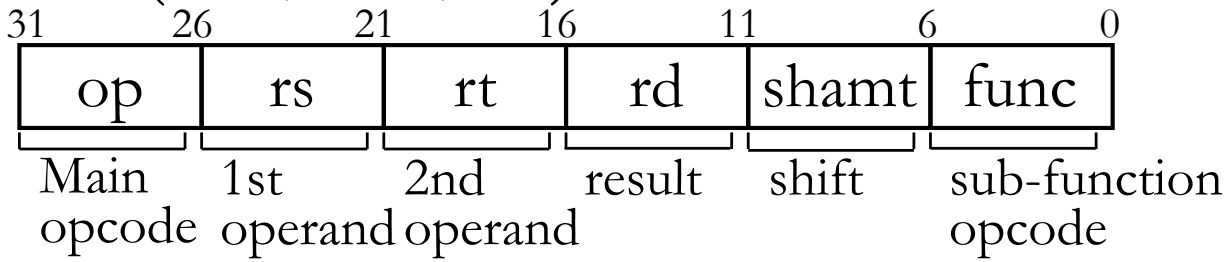
Anywhere (unaligned), or at an mult. 4 address (aligned)?

- Which is the address of a word?

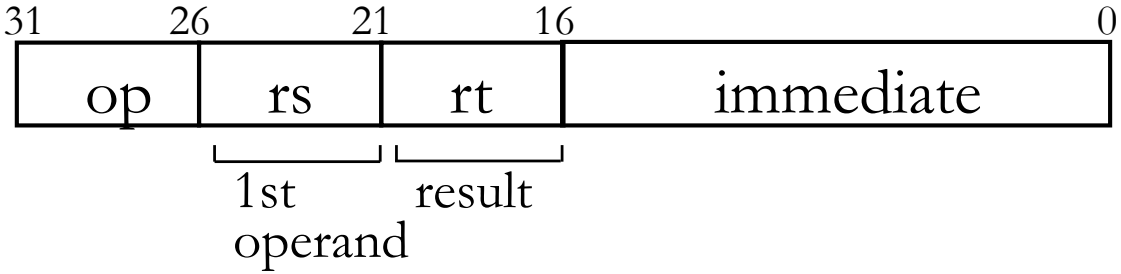



# Instruction formats

- Instruction representation composed of bit-fields
- Similar instructions have the same format
- MIPS instruction formats:
  - R-format (`add`, `sub`, …)




  - I-format (`addi`, `lw`, `sw`, …)



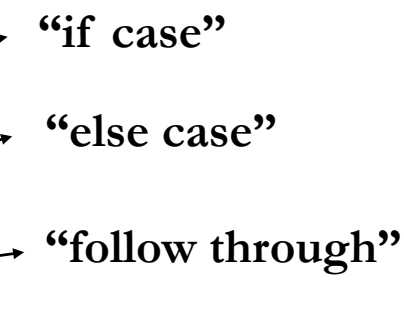
# MIPS instructions – part 2

- 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

# Control transfers: If structures

Java:

```
if (i!=j)
   stmnt1
else
   stmnt2
stmnt3
```



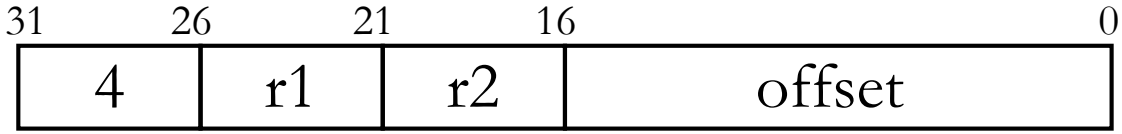

MIPS: `beq $s1,$s2,label`

"branch if equal": compare value in $s1 with value in $s2 and if equal then branch to instruction marked label

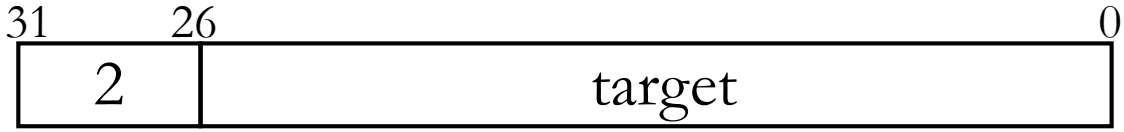
```
        beq $s1,$s2,label1
        stmnt1
        j label2  # skip stmnt2
label1: stmnt2
label2: stmnt3
```

# 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 value as: branch target = PC + 4 * offset
- Similarly: **`bne r1,r2,label  # if r1!=r2 go to label`**
- Unconditional jump, J-format: `j label`



# Loops in assembly language

- Java: `while (count!=0) stmnt`
- MIPS:

```
loop: beq $s1,$zero,end  # $s1 holds count
      stmnt
      j loop  # branch back to loop
end:  …
```

- Java: `while (flag1 && flag2) stmnt`
- MIPS:

```
loop: beq $s1,$zero,end  # $s1 holds flag1
      beq $s2,$zero,end  # $s2 holds flag2
      stmnt
      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: `while (i > j) stmnt`
- MIPS example:
  - assume that $s1 contains i and $s2 contains j

```
loop: slt $t0,$s2,$s1    # $t0 = (i > j)
      beq $t0,$zero,end  # true if i <= j
      stmnt
      j loop  # jump back to loop
end:  …
```

# Method calls

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

```
…
foo();
…
foo();
…

void foo() {
…
return;
}
```

call to foo at line L1

call to foo at line L2



where do we return to?

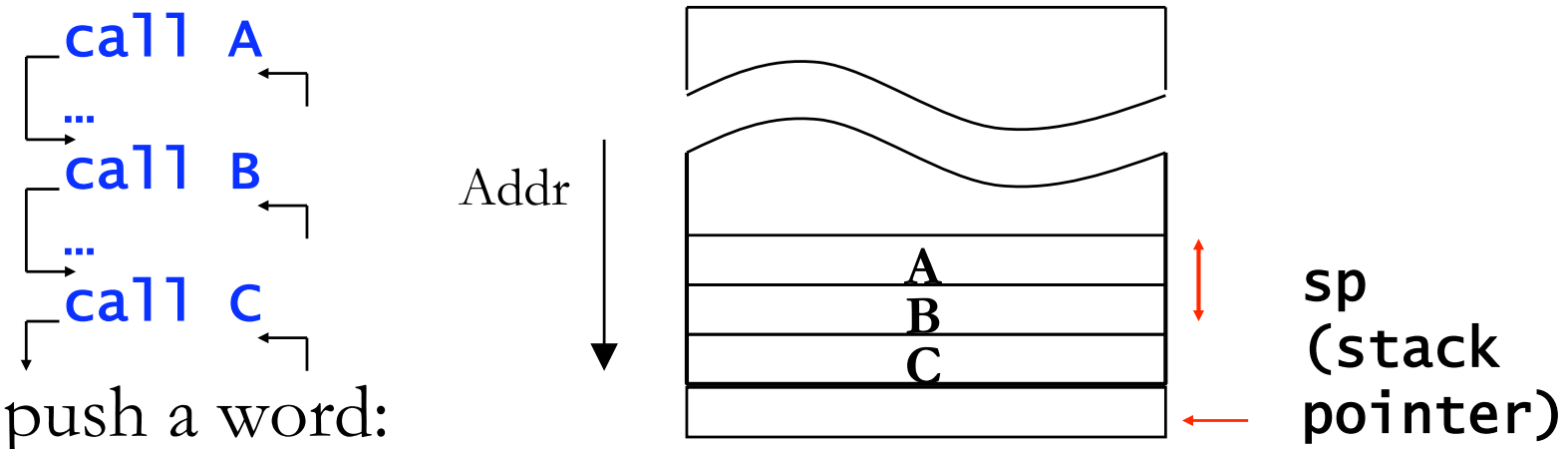
# MIPS support for method calls

- Jumping into the method: `jal label`
  - "jump and link": set $ra to PC+4 and set PC to label
  - Another J-format instruction

- Returning: `jr r1`
  - "jump register": set PC to value in register r1

# Using a stack for method calls

- Nested calls ⇒ must save return address 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
  - MIPS uses $a0 – $a4 for the first 4 word-length parameters, and $v0, $v1 for return values