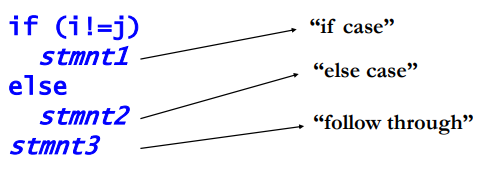
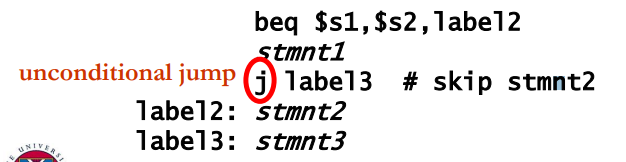
Computer Systems Lecture 6

Control Transfers: If Structures

Java/C:

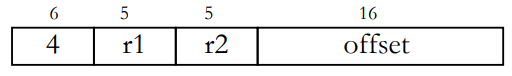


MIPS uses ‘branch if equal’: beq $s1, $s2, label. This compares the value in $s1 with the value in $s2, if they’re equal it will branch to the instruction marked label, so our earlier if would look like:



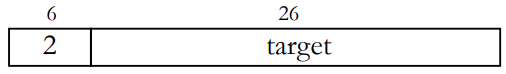
Control Transfer Instructions

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



In assembly code, the label is usually a string, in machine code the lable is a PC-relative offset, the branch address is found using: branch target = PC + 4 \* offset (we multiply by 4 to ensure word alignment). Similarly we have bne r1,r2,label (if r1 != r2 go to label (branch not equal)).

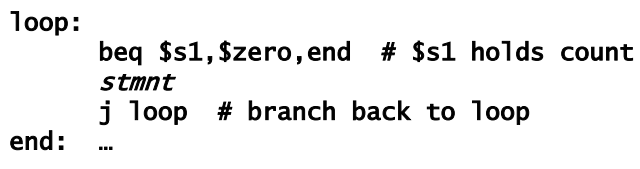
The unconditional jump has its own format know as J-format: j label



Loops in Assembly Language

Java/C: 

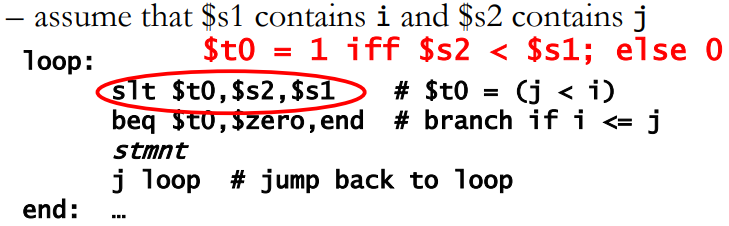
In MIPS this would be written:



Comparisons

In mips we have ‘set if less than’ (in R-format): slt r1 r1,r2,r3 => set r1 to 1 if r2 < r3, otherwise set r1 to 0

A use for this would be to build the equivalent of while (i > j) stmnt in MIPS:

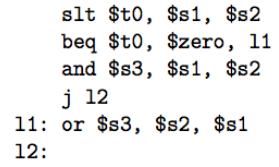


Instruction Format Summary

* R-type
  + Three register operands
  + Most arithmetic, logical and shift instructions
* I-type
  + 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 instructison with a 26 bit address

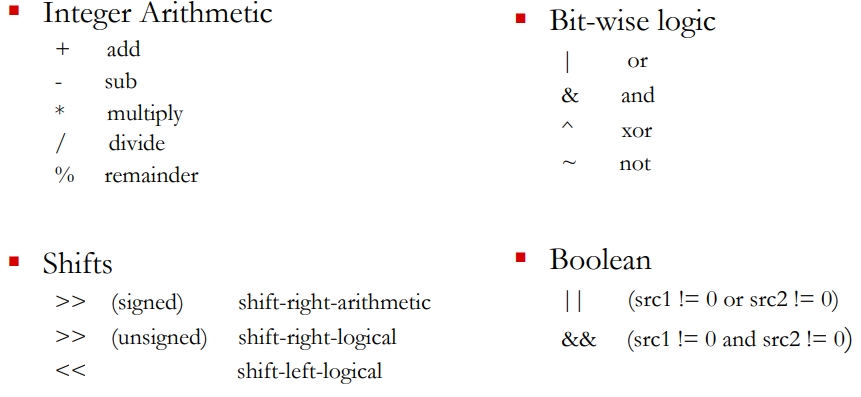
Practice Problem

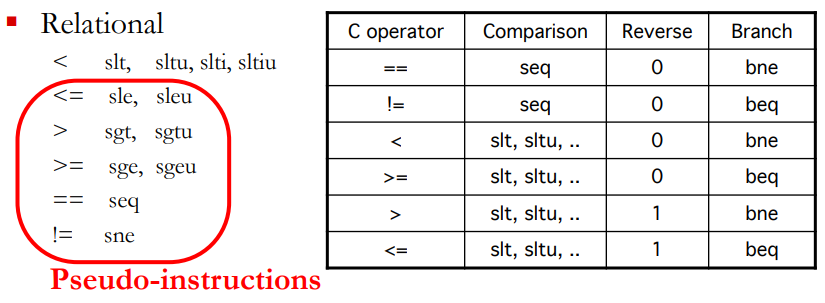
Translate this into C:



if ( s1 < s2) s3 = s1 & s2; else s3 = s1 | s2;

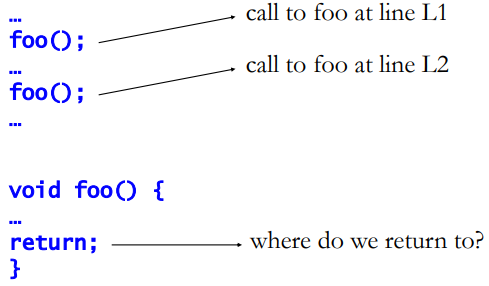
Common MIPS Arithmetic And Logical Operators





Method Calls

Method calls are essential even for a small program, Most ISAs provide support for method calls, in java/C:



MIPS Support for Method Calls

Jumping into the method: jal label

* This stands for ‘jump and link’:
  + Set $ra to PC+4 (+4 to move onto the next instruction)
  + Set PC to label
* This is a J-format instruction

Returning uses: jr ra

* Stands for ‘jump register’ and sets the PC to the value in the register $ra
* Note that any register can be used as a jump target, you need to specify $ra.

MIPS Register Convention on Method Calls

Method parameters go in $a0-$a4

Return values go in $v0, $v1

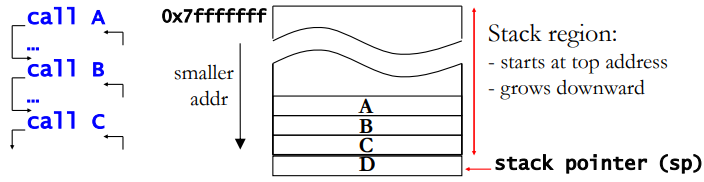
Regs preserved across call boundaries are $s0-$s7

Regs not preserved across call boundaries are $t0-$t9

What about nested method calls?

Using a Stack for Method Calls

Nested calls must save return address to prevent overwriting, the solution for this uses a stack in memory:



To push a word we use:



To pop a word we use:



In MIPS, sp always points to the last valid word on the stack.

Other Uses of the Stack

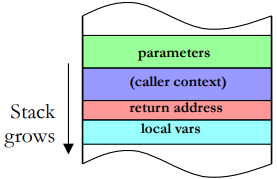
The stack can be used to save caller’s registers so they can be used by the callee

* In MIPS these are the $t registers
* “caller save” vs “callee save”

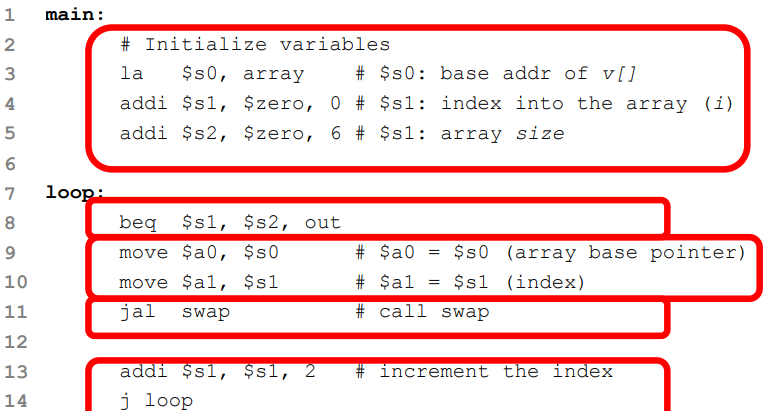
The stack can also be used to pass and return parameters

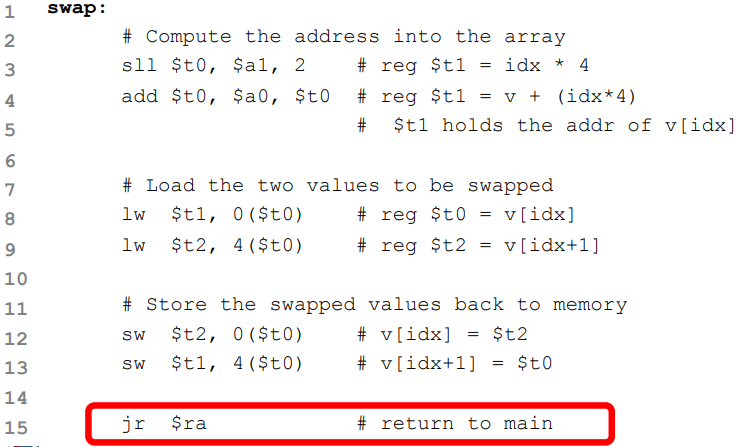
* Gets around the limited number of parameter and return value registers

The stack is also used for local variables within the function.



A Simple Program to Swap Array Elements





Should an ISA be Simple or Complex

ISAs range in complexity

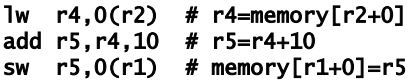
MIPS is a relatively simple ISA

Consider the earlier example:

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

Assembly language:

* Simple:



* Complex:



CISC vs RISC ISAs

Complex Instruction Set (CISC)

* Appeared in early computers, including x86
* Computers programmed in assembly -> HLL features as instructions
* Very few registers -> operands can be in memory
* Very little memory -> variable length instructions to minimize code size

Reduced Instruction Set

* Appeared in the 80s. Used today in ARM, MIPS, and SPARC ISAs
* Compilers -> simple instructions
* More registers -> load-store architecture
* More memory and faster clock frequency -> fixed length, fixed format instructions for easy, fast decoding logic.