CCSYA

Instructions: Language of the Computer

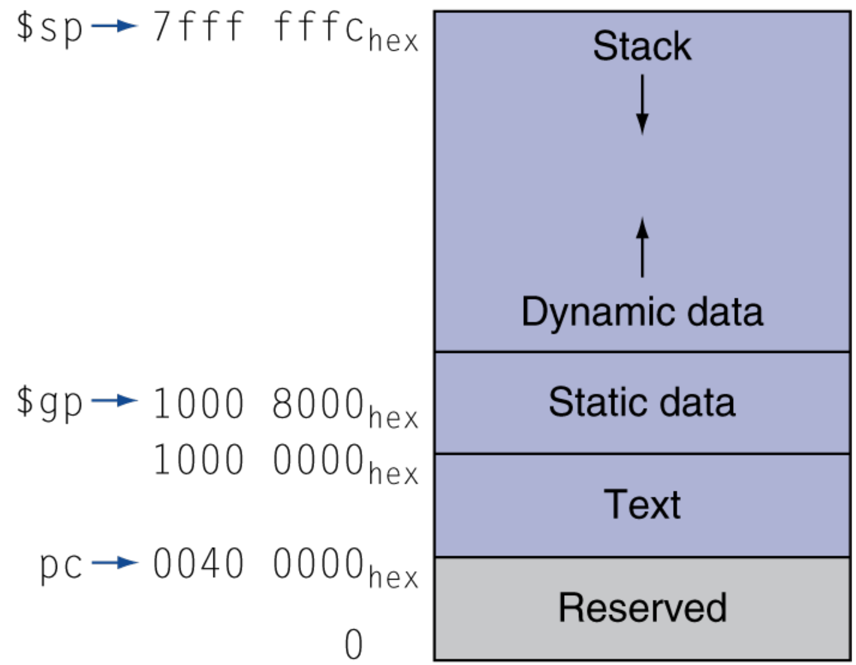
Part II

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

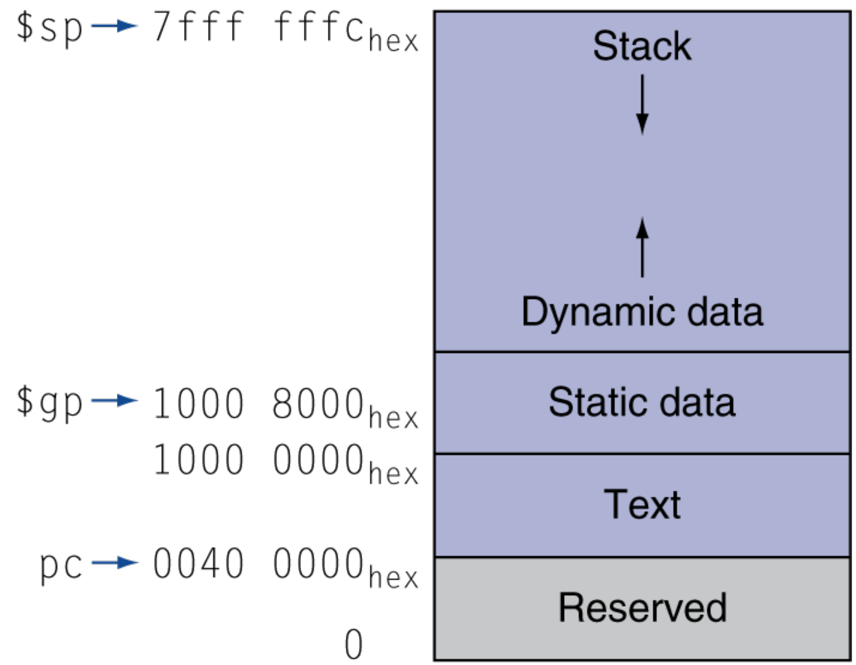
Address Space

- **Text**
  - Program code
- **Static data**
  - Global variables
  - Static variables in C, constant arrays and strings
  - `$gp` initialized to address allowing ± offsets into this segment



Address Space

- **Heap**
  - Memory dynamically allocated during runtime
- **Stack**
  - Temporary data for handling procedure calls
- **The heap and stack segments grow toward each other**
  - Allowing the efficient use of memory as the two segments expand and shrink



Procedure Calling

- **The execution of a procedure call happens when one procedure (the caller) invokes another procedure (the callee)**

- **Steps required**
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

Procedure Calling

- **How to ensure that a procedure call does not change data that is outside its scope?**

- **Programmers who write code in a HLL never see the details of how one procedure calls another**
  - The compiler takes care of the low-level details

- **Programmers who write code in assembly must explicitly implement every procedure call and return**
  - The caller may have to save data before calling the callee
  - The callee may have to save data before running its operations

The Stack

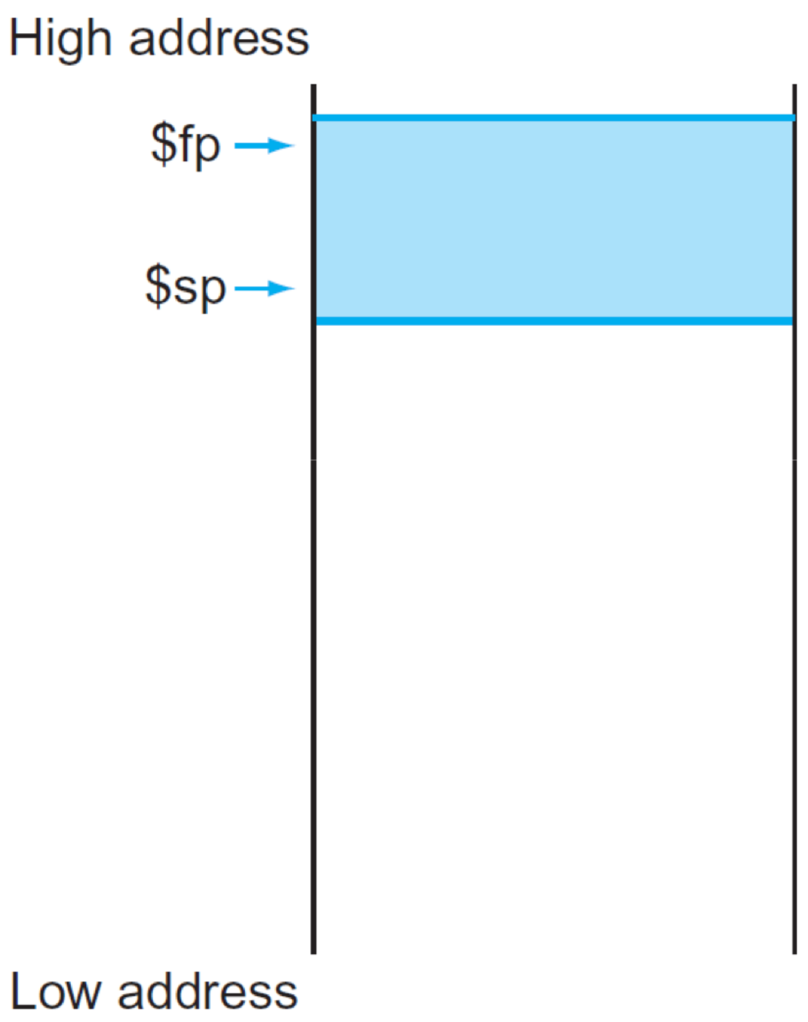
- **The bookkeeping associated with procedure calls is done in the stack segment around blocks of memory called procedure frames**

- **Register `$fp` (frame pointer) points to the base of the current procedure frame**
  - Offers a stable base register as it does not change in a procedure

- **Register `$sp` (stack pointer) points to the top of the stack (the top of the current procedure frame)**
  - Since it can change within a procedure, different references to the same (local) variable might have different offsets in the procedure

The Stack

- The stack grows from higher to lower addresses
- This convention means that you push values onto the stack by subtracting from the stack pointer
- Adding to the stack pointer shrinks the stack, thereby popping values off the stack



Procedure Call Instructions

- **Procedure call: jump and link**

  `jal ProcedureLabel`

  - Address of following instruction (PC+4) put in `$ra`
  - Jumps to target address

- **Procedure return: jump register**

  `jr $ra`

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

Register Usage

- **`$a0 – $a3`**
  - Arguments (reg's 4 – 7)
- **`$v0, $v1`**
  - Result values (reg's 2 and 3)
- **`$t0 – $t9`**
  - Temporaries
  - Can be overwritten by callee
- **`$s0 – $s7`**
  - Saved
  - Must be saved/restored by callee

Register Usage

- **`$gp`**
  - Global pointer for static data (reg 28)
- **`$sp`**
  - Stack pointer (reg 29)
- **`$fp`**
  - Frame pointer (reg 30)
- **`$ra`**
  - Return address (reg 31)

Caller side

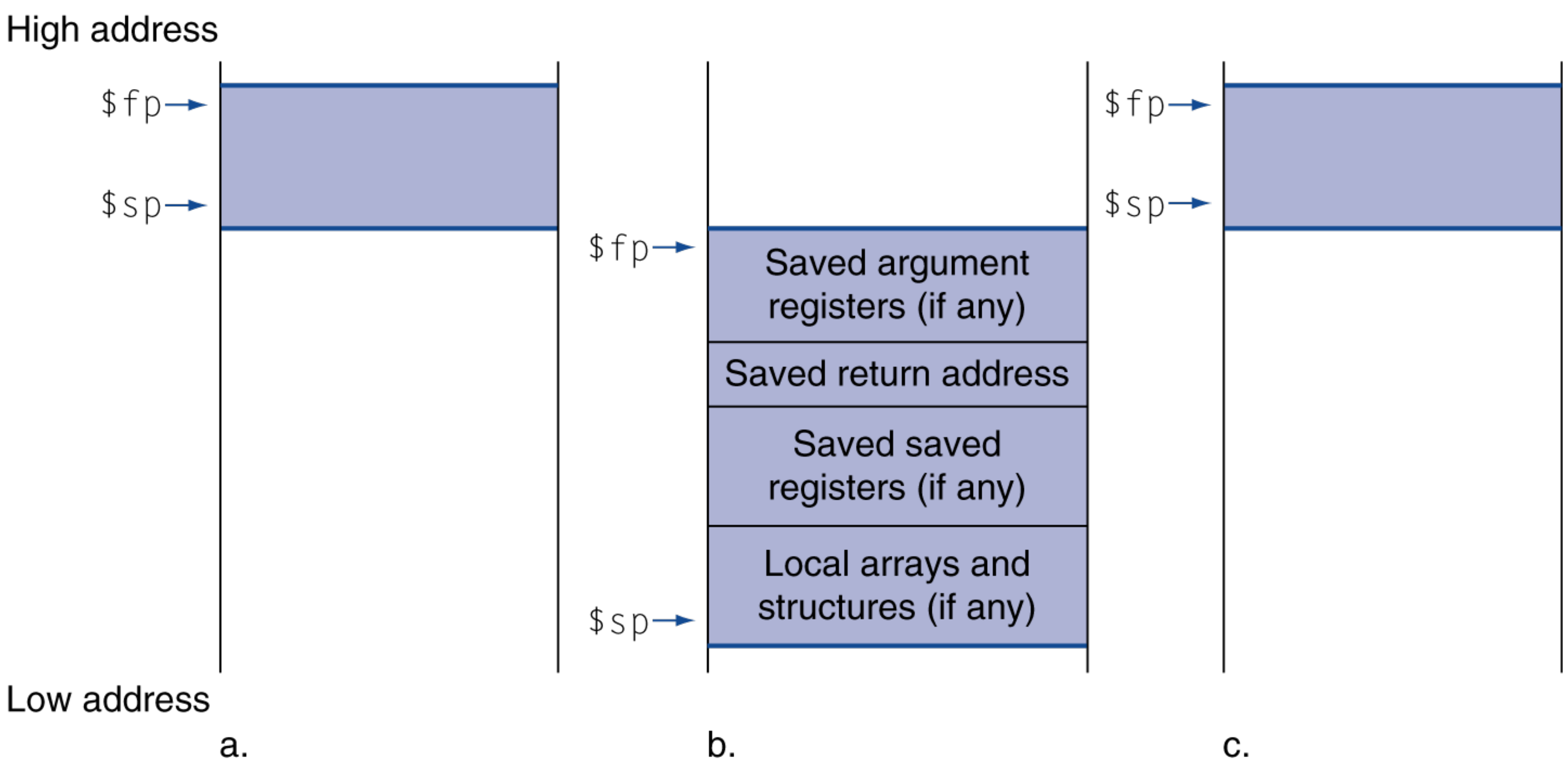
- **If is uses registers `$t0–$t9`, `$a0–$a3` and `$v0–$v1` after the call**
  - Save its values before the call in the current procedure frame
- **The first 4 arguments are put in registers `$a0–$a3`**
  - Additional arguments are pushed on the stack and appear at the beginning of the procedure frame (`$fp` points to the base of the procedure frame)
- **Execute a `jal` instruction to jump to the callee's first instruction**
  - Saves the return address in `$ra`
- **Restores saved registers and handles return values**

Callee Side

- **Allocates memory to local data**
  - Subtracting the required size from `$sp`
- **Saves preserved registers**
  - If it uses registers `$s0`–`$s7`, saves its values in the new procedure frame before altering them
  - `$fp` only needs to be saved if the frame's size is not zero
  - `$ra` only needs to be saved if the callee itself makes a call
- **Returns control to the caller**
  - If the callee returns something, put the result(s) in `$v0`–`$v1`
  - Restore all callee-saved registers (`$fp`, `$ra` and `$s0`–`$s7`)
  - Pop the procedure frame by adding its size to `$sp`
  - Execute a `jr` instruction, jumping to the address in `$ra`

The Stack and Procedure Call

- **The stack allocation (a) before, (b) during, and (c) after the procedure call**



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;
}
```

Leaf Procedure Example

- **MIPS code:**
  - Arguments g, …, j in `$a0`, …, `$a3`
  - f in `$s0` (hence, need to save `$s0` on stack)
  - Result in `$v0`

```
leaf_example:
     addi $sp, $sp, -4      # adjust stack for 1 item
     sw   $s0, 0($sp)       # Save $s0 on stack
     add  $t0, $a0, $a1     # Procedure body
     add  $t1, $a2, $a3
     sub  $s0, $t0, $t1
     add  $v0, $s0, $zero   # Result
     lw   $s0, 0($sp)       # Restore $s0
     addi $sp, $sp, 4       # pop 1 item from stack
     jr   $ra               # Return
```

String Copy Example

- **C code (naive):**
  - Null-terminated string

```
void strcpy (char x[], char y[]){
  int i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

String Copy Example

- **MIPS code:**
  - Addresses of x, y in `$a0`, `$a1`
  - i in `$s0`

```
strcpy:
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw   $s0, 0($sp)       # save $s0
    add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1     # addr of y[i] in $t1
    lbu  $t2, 0($t1)       # $t2 = y[i]
    add  $t3, $s0, $a0     # addr of x[i] in $t3
    sb   $t2, 0($t3)       # x[i] = y[i]
    beq  $t2, $zero, L2    # exit loop if y[i] == 0
    addi $s0, $s0, 1       # i = i + 1
    j    L1                # next iteration of loop
L2: lw   $s0, 0($sp)       # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr   $ra               # and return
```

Non-Leaf Procedures

- **Procedures that call other procedures**
- **For nested call, caller needs to save on the stack:**
  - Its return address
  - Any arguments and temporaries needed after the call
- **Restore from the stack after the call**

Non-Leaf Procedure Example

- **C code:**

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

Non-Leaf Procedure Example

- **MIPS code:**
  - Argument n in $a0, result in $v0

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       #   pop 2 items from stack
    jr   $ra               #   and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

C Sort Example

- **In the next slides, we derive the MIPS code from two procedures written in C:**
  - one to swap array elements…
  - …and one to sort them

- **Swap procedure (leaf)**

```
void swap(int v[], int k){
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

The Swap Procedure

- **MIPS code:**
  - v in `$a0`, k in `$a1`, temp in `$t0`

```
swap:
   sll $t1, $a1, 2    # $t1 = k * 4
   add $t1, $a0, $t1 # $t1 = v+(k*4)
                     #   (address of v[k])
   lw $t0, 0($t1)    # $t0 (temp) = v[k]
   lw $t2, 4($t1)    # $t2 = v[k+1]
   sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
   sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
   jr $ra            # return to calling routine
```

The Sort Procedure in C

- **Non-leaf (calls swap)**
  - v in $a0, k in $a1, i in $s0, j in $s1

```
void sort (int v[], int n){
  int i, j;
  for(i = 0; i < n; i += 1){
    for(j = i - 1; j >= 0 && v[j] > v[j + 1];j -= 1){
        swap(v,j);
    }
  }
}
```

The Procedure Body

Move params

Outer loop

Inner loop

Pass params & call

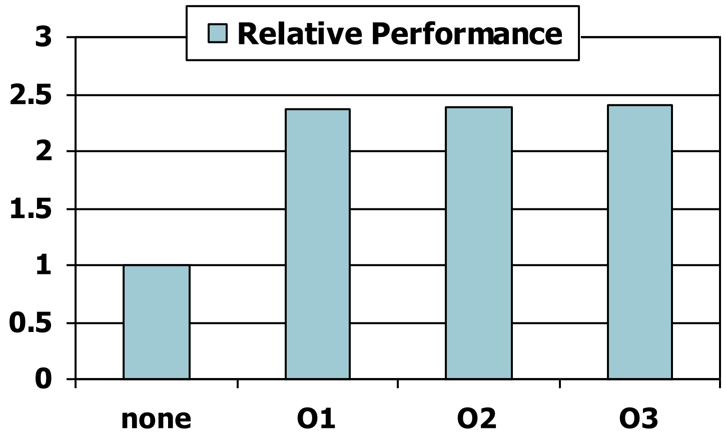
```
         move $s2, $a0          # save $a0 into $s2
         move $s3, $a1          # save $a1 into $s3
         move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
         beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, –1      # j = i – 1
for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2       # $t1 = j * 4
         add  $t2, $s2, $t1     # $t2 = v + (j * 4)
         lw   $t3, 0($t2)       # $t3 = v[j]
         lw   $t4, 4($t2)       # $t4 = v[j + 1]
         slt  $t0, $t4, $t3     # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
         move $a0, $s2          # 1st param of swap is v (old $a0)
         move $a1, $s1          # 2nd param of swap is j
         jal  swap              # call swap procedure
         addi $s1, $s1, –1      # j –= 1
         j    for2tst           # jump to test of inner loop
exit2:   addi $s0, $s0, 1       # i += 1
         j    for1tst           # jump to test of outer loop
```

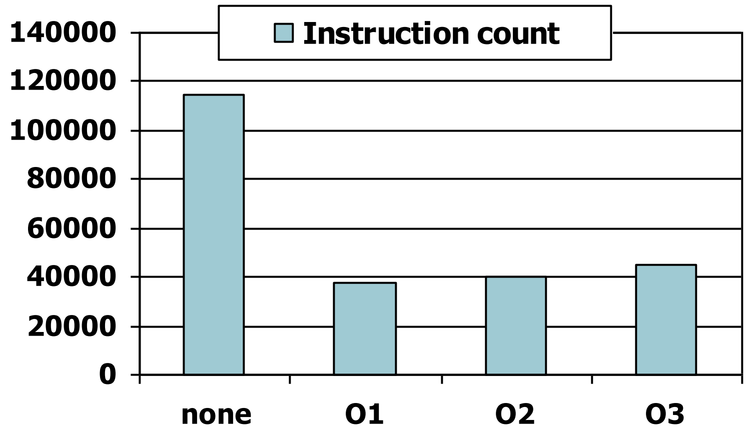
The Full Procedure

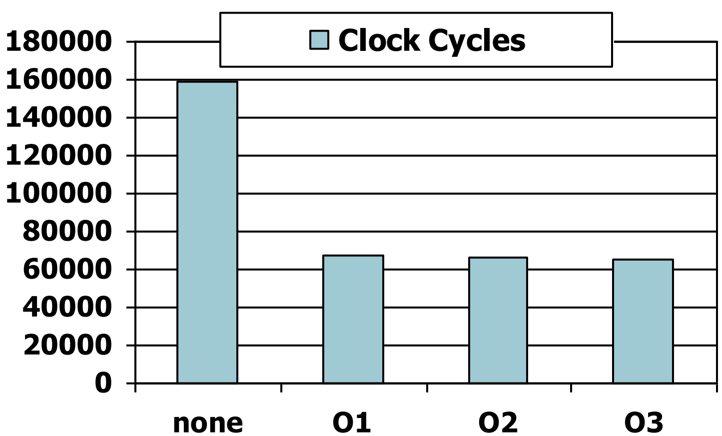
```
sort:  addi $sp,$sp, -20    # make room on stack for 5 registers
       sw $ra, 16($sp)      # save $ra on stack
       sw $s3,12($sp)       # save $s3 on stack
       sw $s2, 8($sp)       # save $s2 on stack
       sw $s1, 4($sp)       # save $s1 on stack
       sw $s0, 0($sp)       # save $s0 on stack

       …                    # procedure body
       …

exit1: lw $s0, 0($sp)       # restore $s0 from stack
       lw $s1, 4($sp)       # restore $s1 from stack
       lw $s2, 8($sp)       # restore $s2 from stack
       lw $s3,12($sp)       # restore $s3 from stack
       lw $ra,16($sp)       # restore $ra from stack
       addi $sp,$sp, 20     # restore stack pointer
       jr $ra               # return to calling routine
```

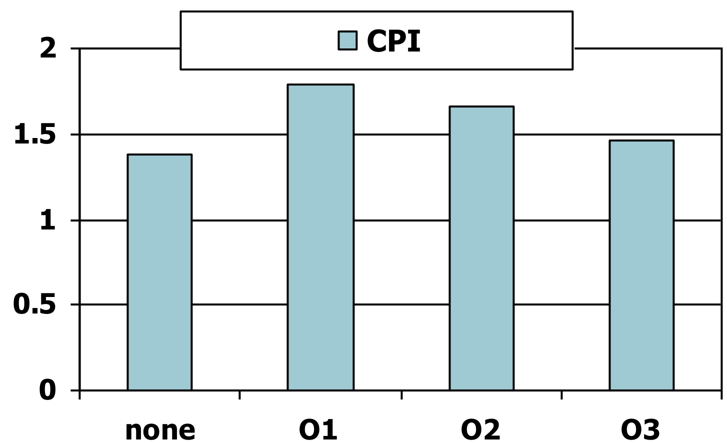
Effect of Compiler Optimization

Compiled with gcc for Intel i5 under Linux

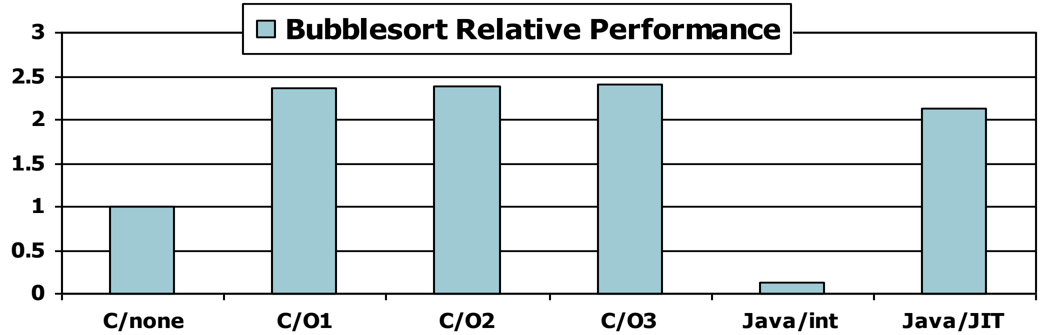


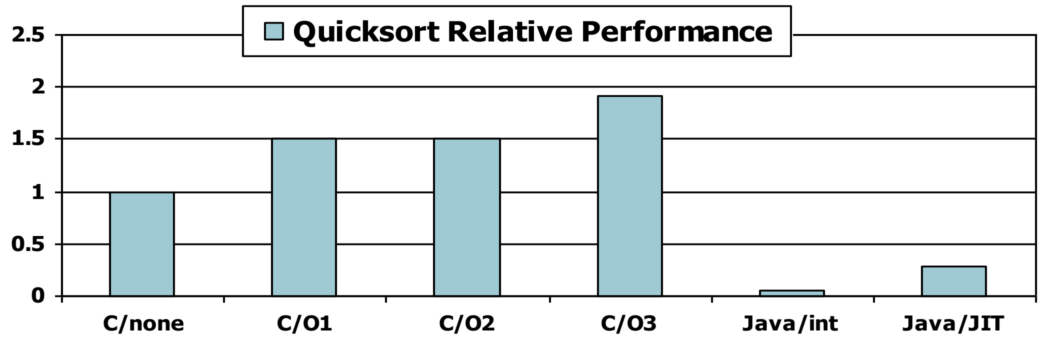


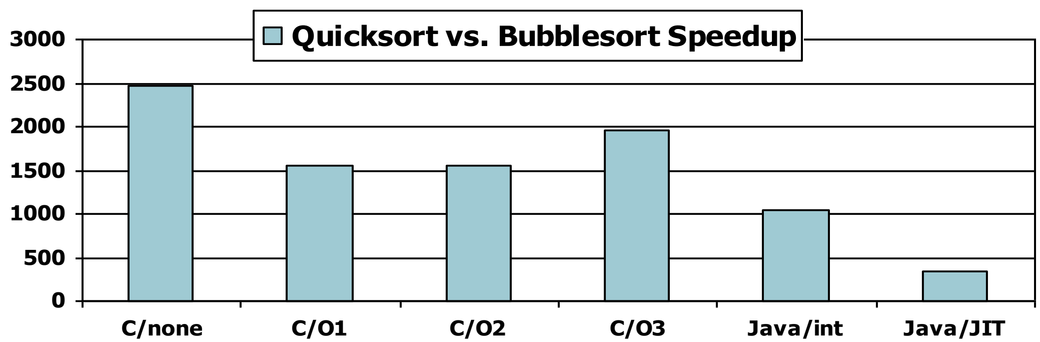




Effect of Language and Algorithm







Lessons Learnt

- **Instruction count and CPI are not good performance indicators in isolation**
  - Reminding us that **time is the only accurate measure of program performance**
- **Compiler optimizations are sensitive to the algorithm**
- **Java/JIT compiled code is significantly faster than JVM interpreted**
  - Comparable to optimized C in some cases
- **Nothing can fix a dumb algorithm!**

Representing Instructions

- **We are now ready to see the difference between:**
  - the way humans instruct computers and…
  - …the way computers see instructions

- **Instructions are encoded in binary**
  - Called machine code

- **Each segment of an instruction is called a field**
  - Each piece of an instruction can be considered as an individual number, and placing these numbers side by side forms the instruction

Representing Instructions

- **MIPS instructions**
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

- **Register numbers**
  - `$t0` – `$t7` are reg's 8 – 15
  - `$s0` – `$s7` are reg's 16 – 23
  - `$t8` – `$t9` are reg's 24 – 25

Representing Instructions

- **Instruction fields**
  - **op**: operation code (opcode)
  - **rs**: first source register number
  - **rt**: second source register number
  - **rd**: destination register number
  - **shamt**: shift amount (how many positions to shift)
  - **funct**: function variant (extends opcode)

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
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

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

`add $t0, $s1, $s2`

- The *op* and *funct* fields in combination (0 and 32 in this case) tell that this instruction performs addition (add)
- The *rs* and *rt* fields, registers `$s1` (17) and `$s2` (18), are the source operands, and the *rd* field, register `$t0` (8), is the destination operand
- The *shamt* field is unused in this instruction, so it is set to 0

Register Addressing

- **Register addressing is a form of direct addressing**
  - The value in the register is an operand instead of being a memory address to an operand
  - Instructions using registers execute fast because they do not have the delay associated with memory access

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- **Immediate arithmetic and load/store instructions**
  - *rt*: destination or source register number
  - *constant*: -2^{15} to $+2^{15} - 1$
  - *address*: offset added to base address in *rs*

Design Principle 4: Good design demands good compromises

Different formats complicate decoding, but allow 32-bit instructions uniformly

Keep formats as similar as possible

From Assembly to Machine Code

```
lw  $t0, 1200($t1)   # $t0 = a[300]
add $t0, $s2, $t0    # $t0 = h + a[300]
sw  $t0, 1200($t1)   # a[300] = $t0
```

35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

- **The similarity between instructions facilitates the design of the CPU**

Immediate addressing

- **Immediate addressing means that one operand is a constant within the instruction itself**
  - The advantage of using it is that there is no need to have extra memory access to fetch the operand
  - But keep in mind that the operand is limited to 16 bits in size

op	rs	rt	Immediate

32-Bit Immediate Operands

- **Although constants are frequently short and fit into the 16-bit field, sometimes they are bigger**

- ***load upper immediate*** **(`lui`) specifically to set the upper 16 bits of a constant in a register**
  - Allowing a subsequent instruction to specify the lower 16 bits of the constant

```
lui rt, constant
```

  - Copies 16-bit constant to left 16 bits of *rt*
  - Clears right 16 bits of *rt* to 0

32-Bit Immediate Operands

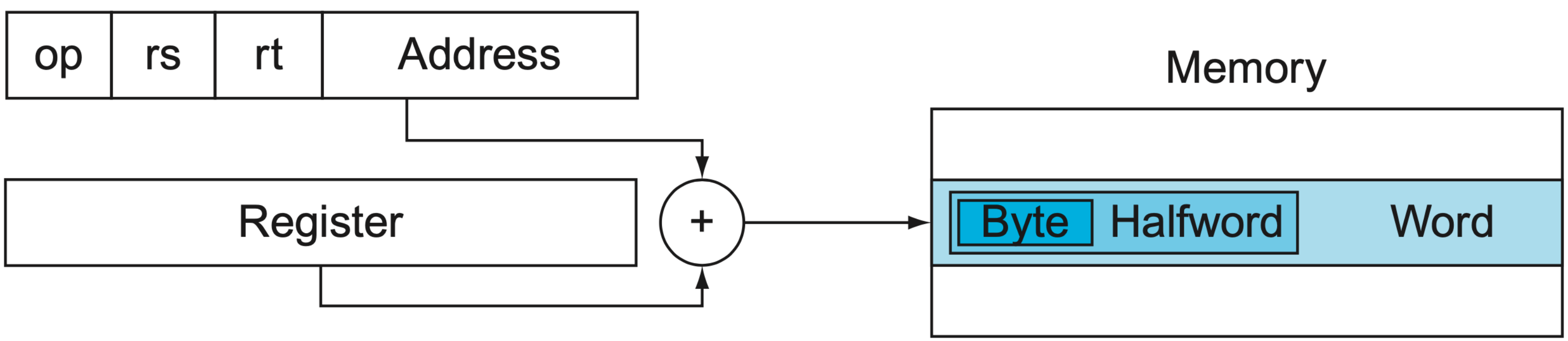
- **What is the MIPS assembly code to load the value 4000000 into register `$s0`?**
  - $40000_{(10)}$ = $0000\ 0000\ 0011\ 1101\ 0000\ 1001\ 0000\ 0000_{(2)}$
  - $61_{(10)}$ = $0000\ 0000\ 0011\ 1101_{(2)}$
  - $2304_{(10)}$ = $0000\ 1001\ 0000\ 0000_{(2)}$

`lui $s0, 61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

- **The compiler or the assembler must break large constants into pieces and then reassemble them into a register**

Base Register Addressing

- **In base register addressing we add a small constant to a pointer held in a register**
  - The register may point to a structure or some other collection of data, and we need to load a value at a constant offset from the beginning of the structure



Branch Addressing

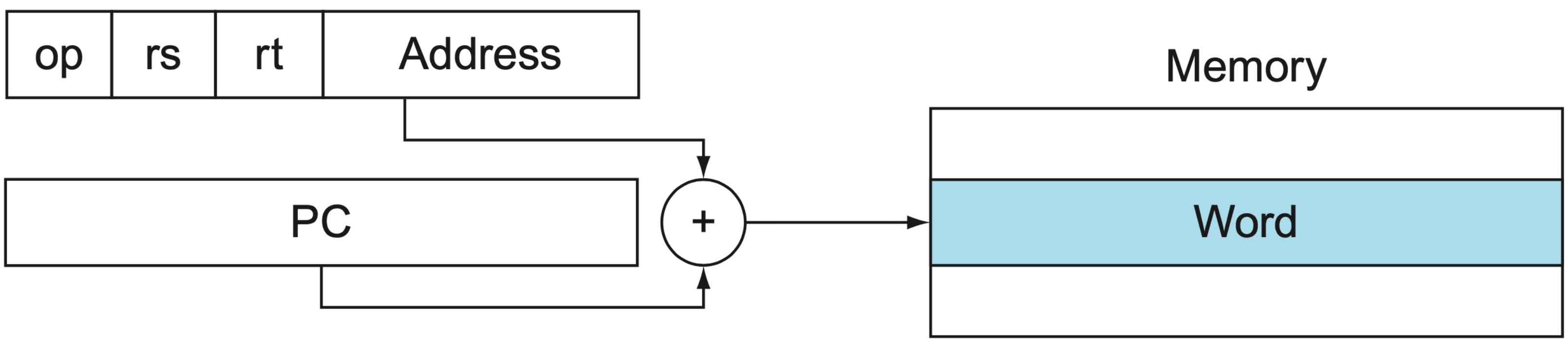
- **Branch instructions specify**
  - Opcode, two registers, target address

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- **Most branch targets are a nearby instruction**
  - Forward or backward
- **PC-relative addressing**
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time
  - Otherwise, it would mean that no program could be bigger than 2^{16}, which is far too small to be a realistic option today

PC-relative Addressing

- **PC-relative addressing is used for conditional branches**
  - The address is the sum of the program counter and a constant in the instruction



MIPS J-Format Instructions

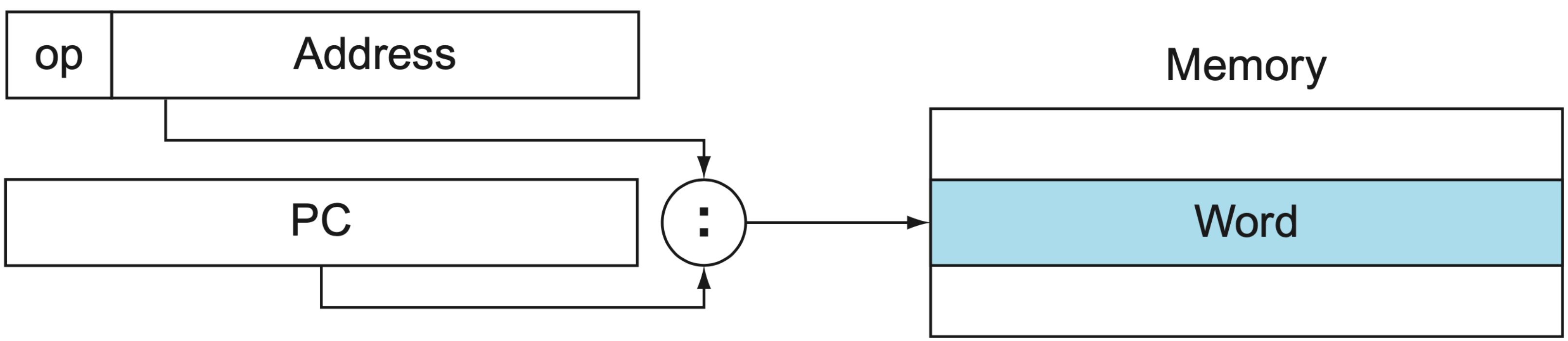
- **Jump (`j` and `jal`) targets could be anywhere in text segment**
  - Encode full address in instruction

op	address
6 bits	26 bits

- **Pseudo-direct jump addressing**
  - Target address = $PC_{31\ldots28}$: (address × 4)
  - The jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

Pseudo-direct Addressing

- **Pseudo-direct addressing is used in jumps**
  - The jump address is a constant in the instruction concatenated with the upper bits of the PC



Target Addressing Example

- **The `bne` instruction on the fourth line**
  - Adds 2 words (8 bytes) to the address of the *following* instruction (80016) instead of using the full destination address (80024)
- **The `j` instruction on the last line**
  - Uses the full address (20000 x 4 = 80000), corresponding to the label Loop

`Loop: sll $t1, $s3, 2`	80000	0	0	19	9	4	0
`add $t1, $t1, $s6`	80004	0	9	22	9	0	32
`lw $t0, 0($t1)`	80008	35	9	8	0		
`bne $t0, $s5, Exit`	80012	5	8	21	2		
`addi $s3, $s3, 1`	80016	8	19	19	1		
`j Loop`	80020	2	20000				
`Exit: …`	80024						

Branching Far Away

- **If branch target is too far to encode with a 16-bit offset, the assembler rewrites the code**
  - Inserts an unconditional jump to the branch target, and inverts the condition so that the branch decides whether to skip the jump

- **Example**

```
beq $s0,$s1, L1
```

→

```
    bne $s0,$s1, L2
    j L1
L2: …
```

RISC Design Principles in MIPS

1. **Simplicity favours regularity**
   - All instructions have a single size
   - Three register operands in all arithmetic instructions
   - Keeping the register fields in the same place in each instruction format

2. **Smaller is faster**
   - Use just 32 registers

3. **Good design demands good compromises**
   - Providing for larger addresses and constants in instructions VS keeping all instructions the same length

RISC Design Principles in MIPS

4. **Make the common case faster**
   - PC-relative addressing for conditional branches and immediate addressing for larger constant operands
   - Most procedures are satisfied with 4 arguments, 2 registers for a return value, 8 saved registers, and 10 temporary registers without ever going to memory