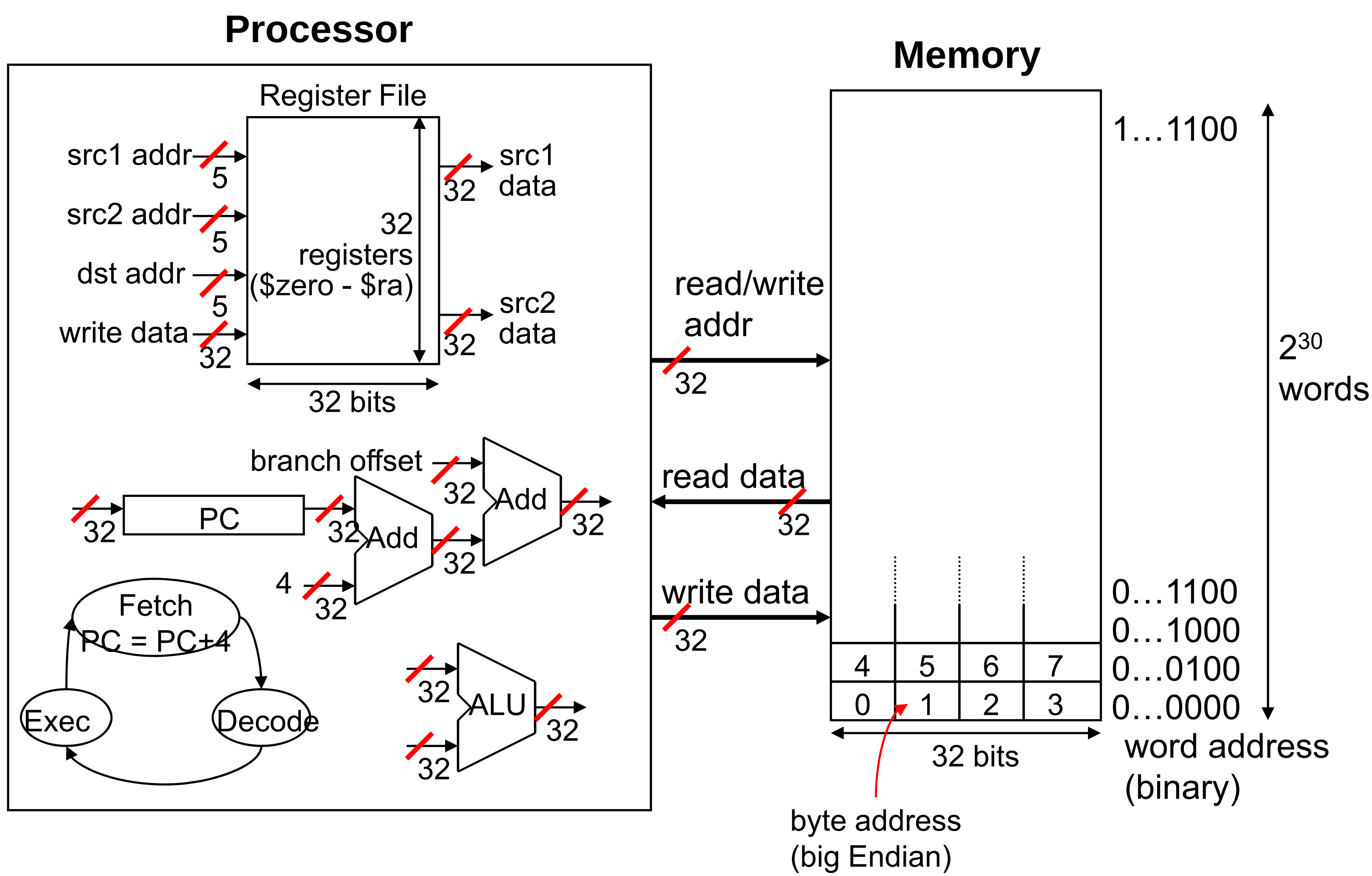
CS3350B
Computer Architecture
Winter 2015

Lecture 4.3: MIPS ISA -- Procedures, Compilation

Marc Moreno Maza
www.csd.uwo.ca/Courses/CS3350b

[Adapted from lectures on
Computer Organization and Design,
Patterson & Hennessy, 5th edition, 2013]

MIPS Organization So Far



Procedure Calling

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

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
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: **jump** and **link**

  `jal ProcedureLabel`

  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: **jump register**

  `jr $ra`

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

Leaf Procedure Example

- C code:

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

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

Leaf Procedure Example

- MIPS code:

`leaf_example:`	
`addi $sp, $sp, -4` `sw   $s0, 0($sp)`	Save $s0 on stack

`add  $t0, $a0, $a1`
`add  $t1, $a2, $a3`
`sub  $s0, $t0, $t1`
	Procedure body

`add  $v0, $s0, $zero`	Result
`lw   $s0, 0($sp)` `addi $sp, $sp, 4`	Restore $s0
`jr   $ra`	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);
}
```

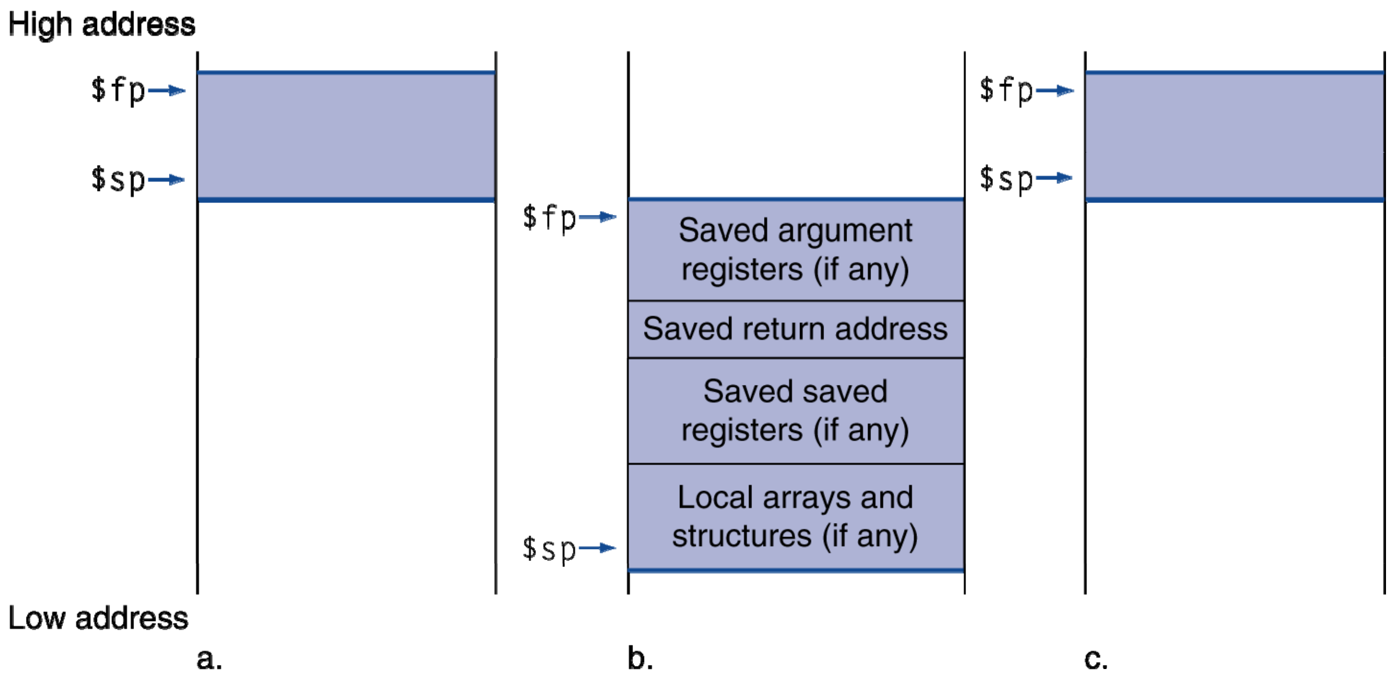
  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

- MIPS code:

```
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
```

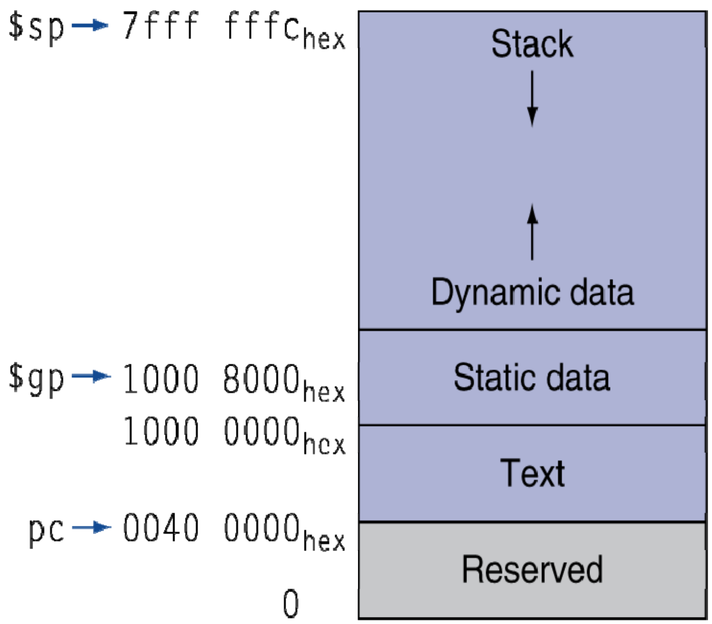
Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

String Copy Example

- C code (naïve):
  - Null-terminated string

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

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- MIPS code:

```
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
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rt, constant
```

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

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

Revisit: Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

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

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Revisit: Jump Addressing

- 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)

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

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

Synchronization

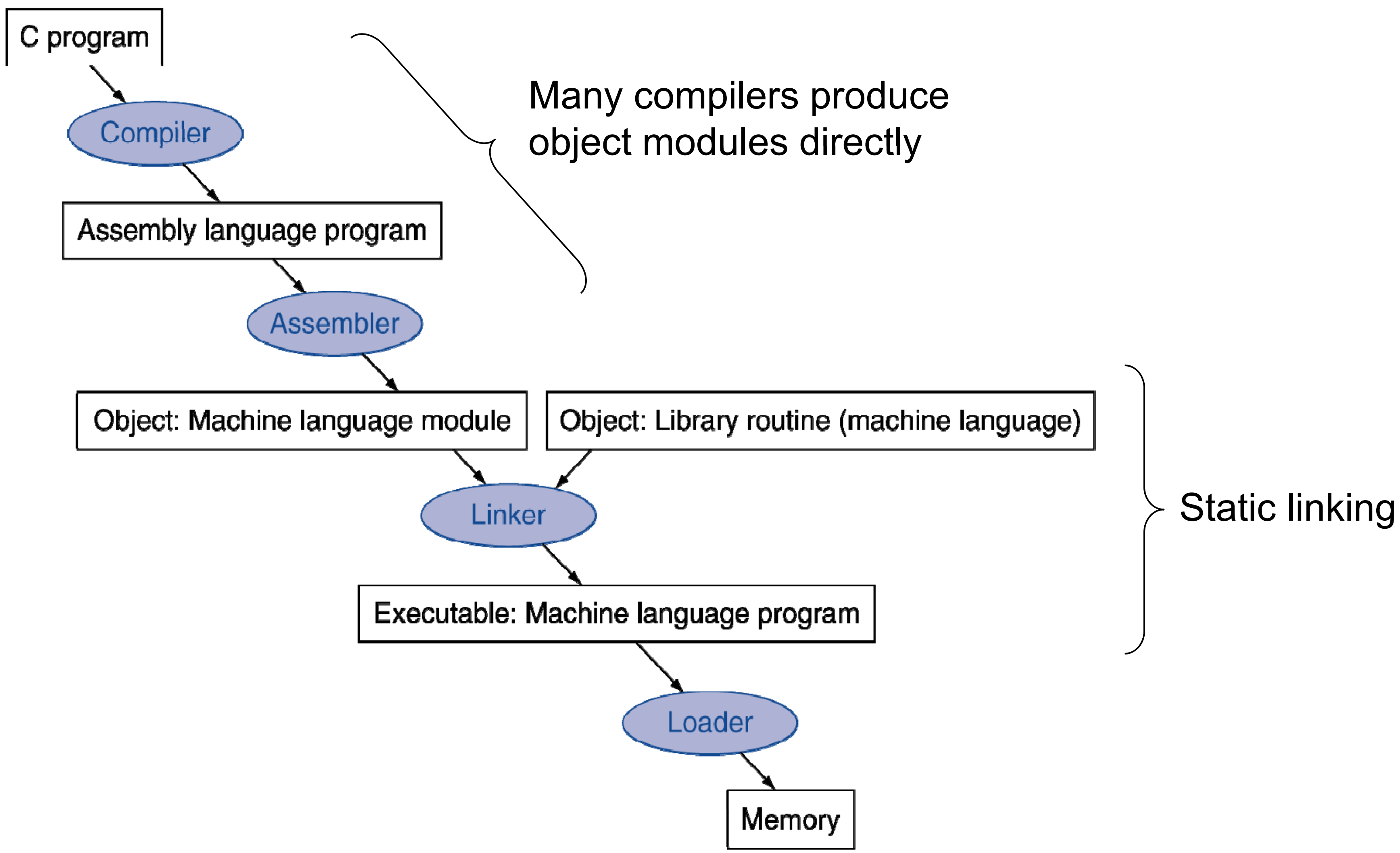
- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - **Atomic read/write** memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in MIPS

- **Load linked**: `ll rt, offset(rs)`
- **Store conditional**: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: **atomic swap** (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

Translation and Startup



Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1     → add $t0, $zero, $t1
blt $t0, $t1, L   → slt $at, $t0, $t1
                    bne $at, $zero, L
```

  - $at (register 1): assembler temporary

Producing an Object Module

- **Assembler** (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - **Header**: described contents of object module
  - **Text segment**: translated instructions
  - **Static data segment**: data allocated for the life of the program
  - **Relocation info**: for contents that depend on absolute location of loaded program
  - **Symbol table**: global definitions and external refs
  - **Debug info**: for associating with source code

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

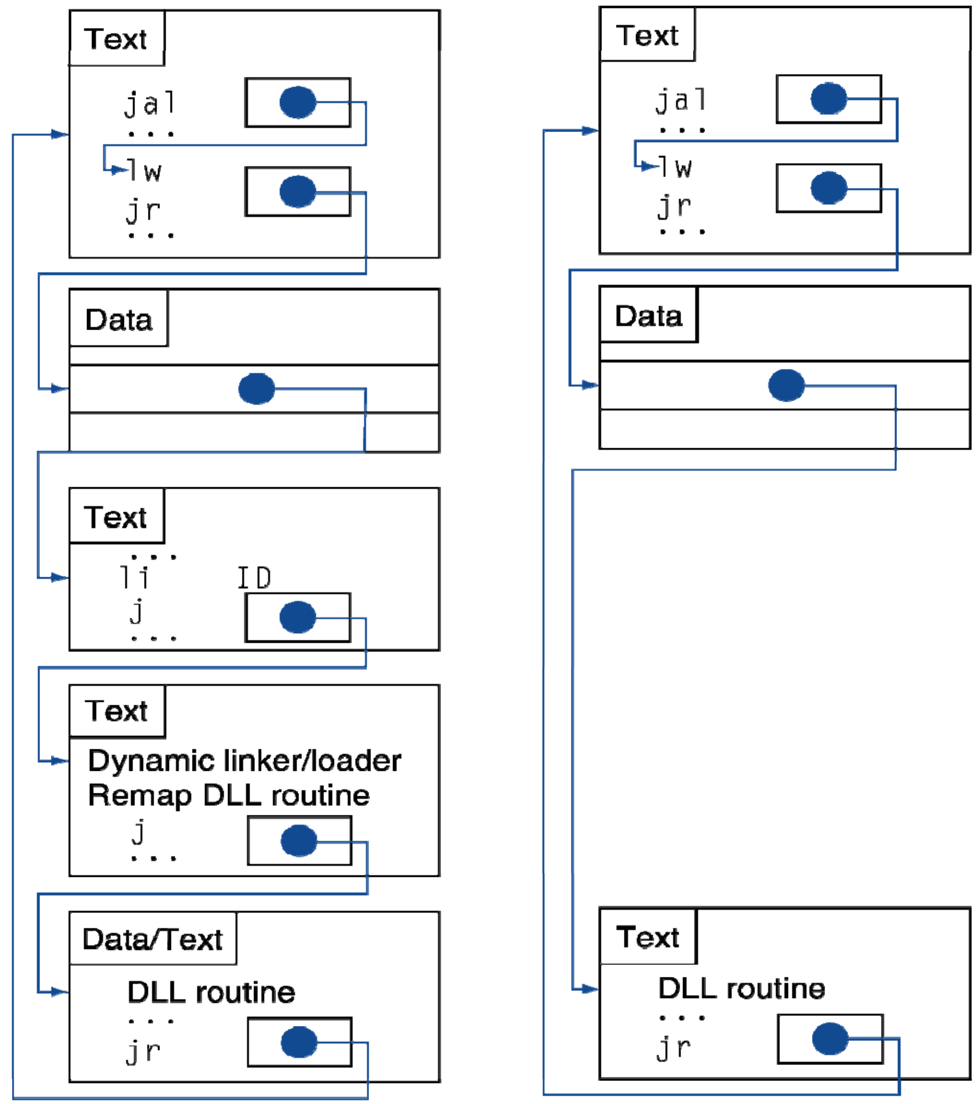
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

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

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
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)

```
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);
    }
  }
}
```

  - v in $a0, k in $a1, i in $s0, j in $s1

The Procedure Body

```
         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
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

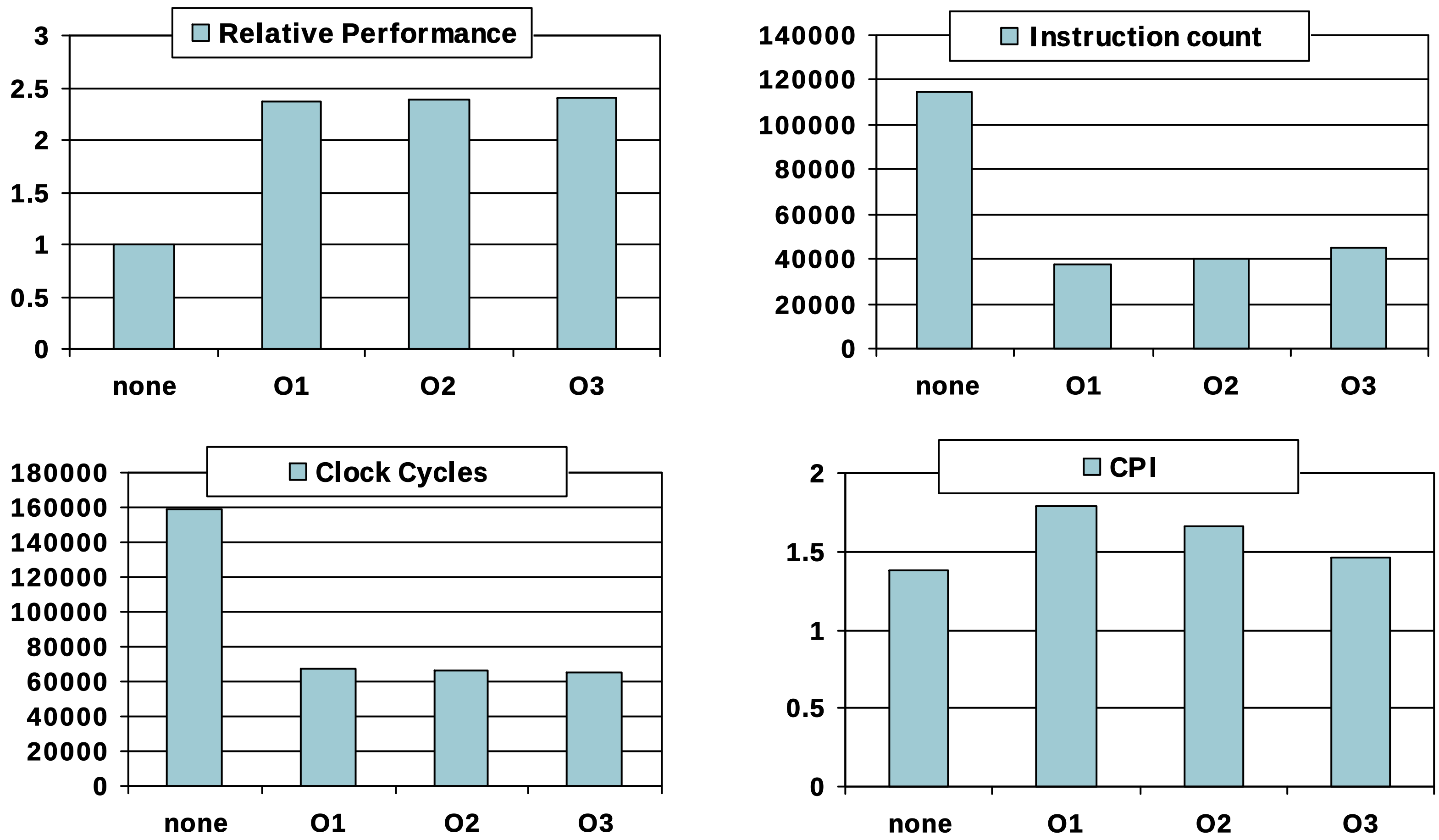
Outer loop

The Full Procedure

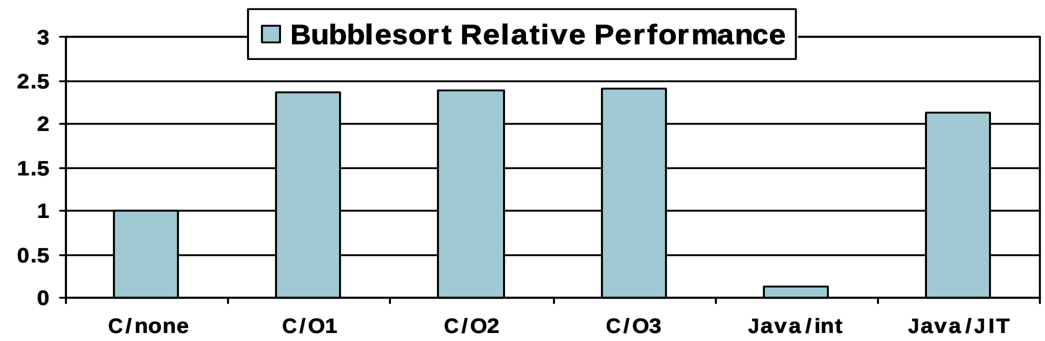
```
sort:     addi $sp,$sp, -20       # make room on stack for 5 registers
          sw $ra, 16($sp)         # save $ra on stack
          sw $s3,12($sp)          # save $s3 on stack
          sw $s2, 8($sp)          # save $s2 on stack
          sw $s1, 4($sp)          # save $s1 on stack
         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

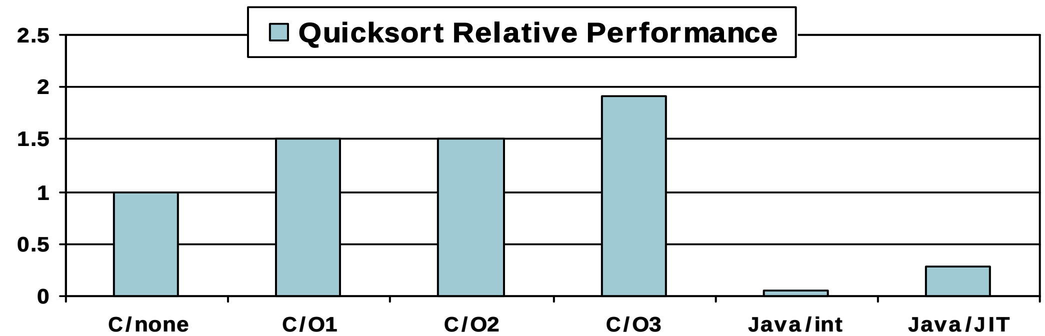
Effect of Compiler Optimization

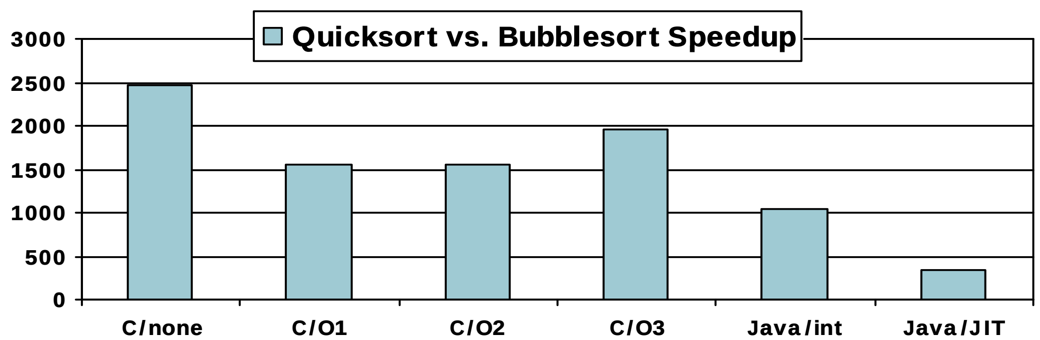
Compiled with gcc for Pentium 4 under Linux



Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- 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!

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing and Array

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p <
&array[size];
       p = p + 1)
     *p = 0;
}
```

```
       move $t0,$zero    # i = 0
loop1: sll $t1,$t0,2     # $t1 = i * 4
       add $t2,$a0,$t1   # $t2 =
                         #   &array[i]
       sw $zero, 0($t2)  # array[i] = 0
       addi $t0,$t0,1    # i = i + 1
       slt $t3,$t0,$a1   # $t3 =
                         #   (i < size)
       bne $t3,$zero,loop1 # if (…)
                           # goto loop1
```

```
       move $t0, $a0     # p = & array[0]
       sll $t1,$a1,2     # $t1 = size * 4
       add $t2,$a0,$t1   # $t2 =
                         #   &array[size]
loop2: sw $zero,0($t0)   # Memory[p] = 0
       addi $t0,$t0,4    # p = p + 4
       slt $t3,$t0,$t2   # $t3 =
                         #(p<&array[size])
       bne $t3,$zero,loop2 # if (…)
                           # goto loop2
```

Comparison of Array vs. Pointer

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%

Takeaway: MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions – 32-bits
  - small number of instruction formats
  - opcode always the first 6 bits
- Good design demands good compromises
  - 3 basic instruction formats
- Smaller is faster
  - limited instruction set
  - limited number (32) of registers in register file
  - limited number (5) of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

Question: How Can We Make It Even Faster?

Aside: ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Aside: The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

Aside: The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

Aside: The Intel x86 ISA

- And further…
  - **AMD64 (2003): extended architecture to 64 bits**
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - **AMD64 (announced 2007): SSE5 instructions**
    - **Intel declined to follow, instead…**
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success