MIPS procedures
Compiling and linking

CS207, Fall 2004
September 22, 2004

Procedures in MIPS: data

- Stack
  - for (temporarily) storing data elsewhere
  - stack pointer register ($sp) stores the location of the “top” (last) item on the stack
  - add items: subtract from stack pointer
  - remove items: add to stack pointer

More on the stack

- Necessary for procedures that call other procedures, recursive procedures, etc.
  - push $ra, $a0-$a3, $t0-$t9, etc.
  - factorial example (pg 83-84 in text)
- frame pointer (optional): always points to the first line of a procedure
- heap: space in memory below the stack for dynamically generated objects, data structures, linked lists, etc.
  - picture, pg. 87 in text

Procedures in MIPS: control passing

- jal address: pass control from the main program to the procedure by jumping to the address of the first instruction in the procedure, and storing the return address in $ra
- jr $ra: “jump register”: jump to the address stored in register $ra
- Note: Current address is stored in the program counter (PC) register; next instruction is at PC+4
MIPS character data

- `lw` and `sw` will read in bytes from words
- alternate: `lb` and `sb` (load byte/store byte)
  - useful for text processing (ASCII is 1 byte)
  - `lb`: byte in memory --> rightmost 8 bits in register
  - `sb`: rightmost 8 bits of register --> byte in memory
- Unicode: 2 bytes per word
  - `lh`, `sh`: same as `lb` and `sb`, but operates on "halfwords" (2 bytes)

Representing strings

- Strings = arrays of characters
- Variable length! How to denote?
- C/C++: special character denotes end of string
  - one extra byte per string
- Java:
  - variable stores the length of the string
  - Unicode

MIPS addressing for larger operands

- Q: What happens if we have to add a large constant to a register?
  - A: load it in "pieces"
- Example: Load the following into memory: 1000 0000 0000 0001 0000 0000 0000 1111
  - `lui $t0, 32769` # value in upper 16 bits
  - `ori $t0, $t0, 15` # value in lower 16 bits

Dealing with large addresses

- J-type instructions: 6 bits for operand + 26 bits for register
  - jump, jump-and-link
- PC-relative addressing: address is sum of PC+4+specified address
  - conditional branches
  - exploits the fact that most branches in conditionals/loops are "nearby" (half are < 16 instructions away!)
  - specified address = number of "words" to add to PC
Summary: MIPS addressing modes

- Register addressing (operand = register)
- Base/displacement addressing (operand address = register + constant)
- Immediate addressing (operand = constant)
- PC-relative addressing
- Pseudodirect addressing (address = concat(6 upper PC bits, 26 instruction bits)

Compiling and executing programs

- Source: Patterson and Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed

Compiling and running a Java program

- Source: Patterson and Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed

Linking

- Combining several independently-assembled programs into one executable
  - e.g., your program plus library functions
- Static: link all components before execution
- Dynamic: linking occurs as program is running
  - DLLs and most Unix library functions

- Source: Patterson and Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed
Java

- Java Virtual Machine = interpreter for Java bytecode
  - simulates the instruction set architecture of a machine
- Bytecode = assembly-like language with few optimizations
- JIT (Just in Time) compiling: commonly-used parts of the program are compiled into native machine code as the program is running

More about compiling

- *Optimizations*: “tricks” the compiler takes to improve the performance of a program under {all, certain} conditions
- High-level optimizations: first pass; close to source code
  - *procedure inlining*: replace function call w/ function body
  - *loop unrolling*: replicate the loop body to reduce the number of iterations
- Note: compilers will always act *conservatively*!

Compiler optimizations (cont.)

- Local optimizations: optimization within a single unit or “block” of instructions
  - replace multiple instances of an expression with references
  - replace complex instructions with simpler ones
  - eliminating “dead” (unused) code
  - Note: all of these can be global optimizations too

Compiler optimizations (cont.)

- Global optimizations: optimization across multiple blocks
  - register allocation (of variables)
  - *code motion*: moves “loop invariant” code outside of the loop
  - *induction variable elimination*: replace array indexing with pointers