MIPS Procedure Calls
JVM and Assignment 3

CSE 378 – Section 3

Procedure Call Basics

- Jump to procedure:
  jal <label>
  - Saves return address to $ra
- Return from a procedure:
  jr $ra
- $a0 - $a3 to pass arguments
- $v0 and $v1 to return values
- Save certain registers to preserve across procedure calls.
  - Use the stack
- $t0-$t9, $a0-a3, $v0-v1 – *caller-saved*.
  - Caller's responsibility to save if expects to use these after a call.
- $s0-$s7, $ra, $fp – *callee-saved*.
  - Callee's responsibility to save if callee uses them.
  - Save at beginning of function, restore at end

Calling procedures

To call a procedure:

1. Put arguments into $a0-a3
2. Save caller-saved registers
3. jal <proc>
4. Restore caller-saved registers

Example:

```
<some stuff here, uses $t2>
…
# set up a call to myproc(4)
addi $a0, $0, 4
subu $sp, $sp, 4
sw $t2, 0($sp)
jal myproc
lw $t2, 0($sp)
addiu $sp, $sp, 4
…
<use $t2 again>
```


Setup at the start/end of procedure

Before any procedure starts running, it must:

1. Allocate memory for callee-saved registers
2. Save callee-saved registers
   - If calling another procedure inside, must save $ra! (why?)

At the end of procedure:

1. Place return value into $v0
2. Restore callee-saved regs
3. jr $ra

```
myproc: # wants to use $s0 inside
    subu $sp, $sp, 8
    sw $ra, 4($sp)
    sw $s0, 0($sp)
    …
    <do some computation in $s0>
    …
    addi $v0, $s0, 42
    lw $s0, 0($sp)
    lw $ra, 4($sp)
    addiu $sp, $sp, 8
    jr $ra
```


Miscellaneous

- MIPS stack conventions:
  - $sp double-word aligned
  - Minimum frame size is 24 bytes (fits four arguments and return address)
  - Don't use it for projects
  - Other rules flexible too: have to use common sense for what you need to save
- If >4 arguments, use the stack to pass them
  - Caller, callee must agree on where they go in the stack and who pops them off.

JVM and Assignment 3

- JVM is a stack machine
  - Portability
  - Compactness
- Our simplified JVM consists of:
  - execution stack
    - Instructions take parameters from the stack
    - Instructions place results onto the stack
  - Pointer to top of the stack
  - local storage
    - Just a big array for storing data
  - Java bytecode program
  - Program counter

Emulating JVM

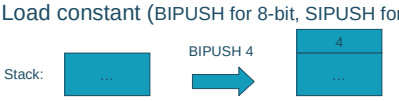
- Interpreter:
  - Get next instruction
  - Decode it
  - Execute
  - Store results
  - Repeat

Emulating JVM

- Probably need SPIM registers for:
  - Pointer to top of JVM stack
  - Pointer to current JVM instruction (PC)
  - Holding a couple of values from the stack (when pushing/popping) – v1, v2
- Use SPIM static data section for:
  - The entire execution stack (1024 bytes maximum)
  - the local storage area
  - The program itself
    - sequence of instruction opcodes and parameters

JVM Instructions

- Load constant (BIPUSH for 8-bit, SIPUSH for 16)

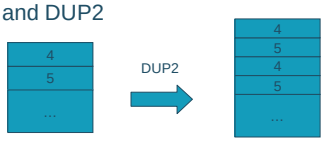


- Arithmetic (IADD, ISUB, IMUL, IDIV)

JVM Instructions 2

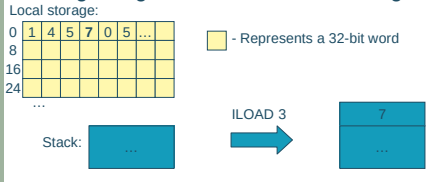
- POP

- DUP and DUP2



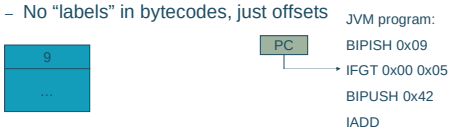
Loading from local storage

- ILOAD, ISTORE – load/store *32-bit word* using unsigned *8-bit index* into storage



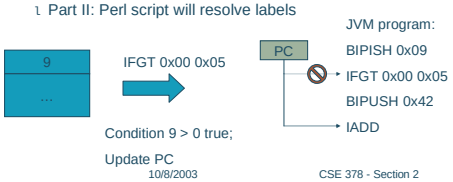
Branches

- Pop one thing off stack, compare with zero using specified condition, update PC if true
- Take a signed 2-byte offset from current PC
  - No “labels” in bytecodes, just offsets



Branches

- To understand offset destinations, add up opcodes (1 byte), along with any arguments
  - E.g. IFEQ 0x00 0x05 takes 3 bytes, IADD takes 1.
    - Part II: Perl script will resolve labels



Condition 9 > 0 true;

Update PC

Example: a=a+b+c

Add first 3 words in local storage

Store the result into the first local storage word

ILOAD 0
ILOAD 1
ILOAD 2
IADD
IADD
ISTORE 0

Example 2: if (b==0) a=3; else a=5;

- Assume a is local word 0,
  b is local word 1:

```
        ILOAD 1
        IFEQ skip
        BIPUSH 3
        ISTORE 0
        GOTO endif
skip:   BIPUSH 5
        ISTORE 0
endif:  …
```

To bytecodes
(use perl script)

```
  .align  2
test2:
  .byte   0x15, 0x01       # iload
  .byte   0x99, 0x00, 0x0a # ifeq
  .byte   0x10, 0x03       # bipush
  .byte   0x36, 0x00       # istore
  .byte   0xa7, 0x00, 0x07 # goto
  .byte   0x10, 0x05       # bipush
  .byte   0x36, 0x00       # istore
  .byte   0x00             #
  .align  2
end_test2:
```