EECE 321: Computer Organization

Mohammad M. Mansour
Dept. of Electrical and Compute Engineering
American University of Beirut

Lecture 9: MIPS ISA

Announcements

- Reading assignment
  - Ch2: Sections 2.1-2.15, 2.18, 2.19
- Install and learn the SPIM tool (from textbook CD)
  - MIPS assembler simulator
- HW3 posted
  - Due Monday Mar. 15, 5:00pm
- Machine problem 1 due Friday March 12 @ 5:00pm
  - Submit on Moodle (one submission per team)
- No lectures on the following dates:
  - M: Mar. 15
  - W: Mar. 17
  - F: Mar. 19
  - M: Mar. 22
  - W: Mar. 24
- Makeup lectures on Saturday March 13
  - Time and place TBA

Steps for Making a Procedure Call

- Steps:
  1. Save all necessary values onto stack (using sw).
  2. Assign argument(s), if any.
  3. Use jal
  4. Restore values from stack (using lw).
- Rules for procedures:
  - Called with a jal instruction, returns with a jr $ra
  - Accepts up to 4 arguments in $a0,$a1,$a2,$a3
  - Return value is always in $v0 (and if necessary in $v1)
  - Must follow register conventions (even in functions that only you will call).
- Basic structure of a function:

Prologue

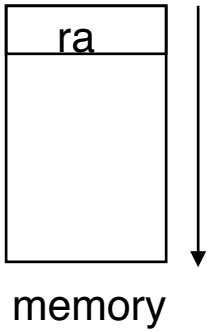
Body

Epilogue

```
entry_label:
addi $sp,$sp, -framesize   # allocate frame area
sw $ra, framesize-4($sp)   # save $ra
save other regs if need be

        ...                # (call other functions…)

restore other regs if need be
lw $ra, framesize-4($sp)   # restore $ra
addi $sp,$sp, framesize
jr $ra
```

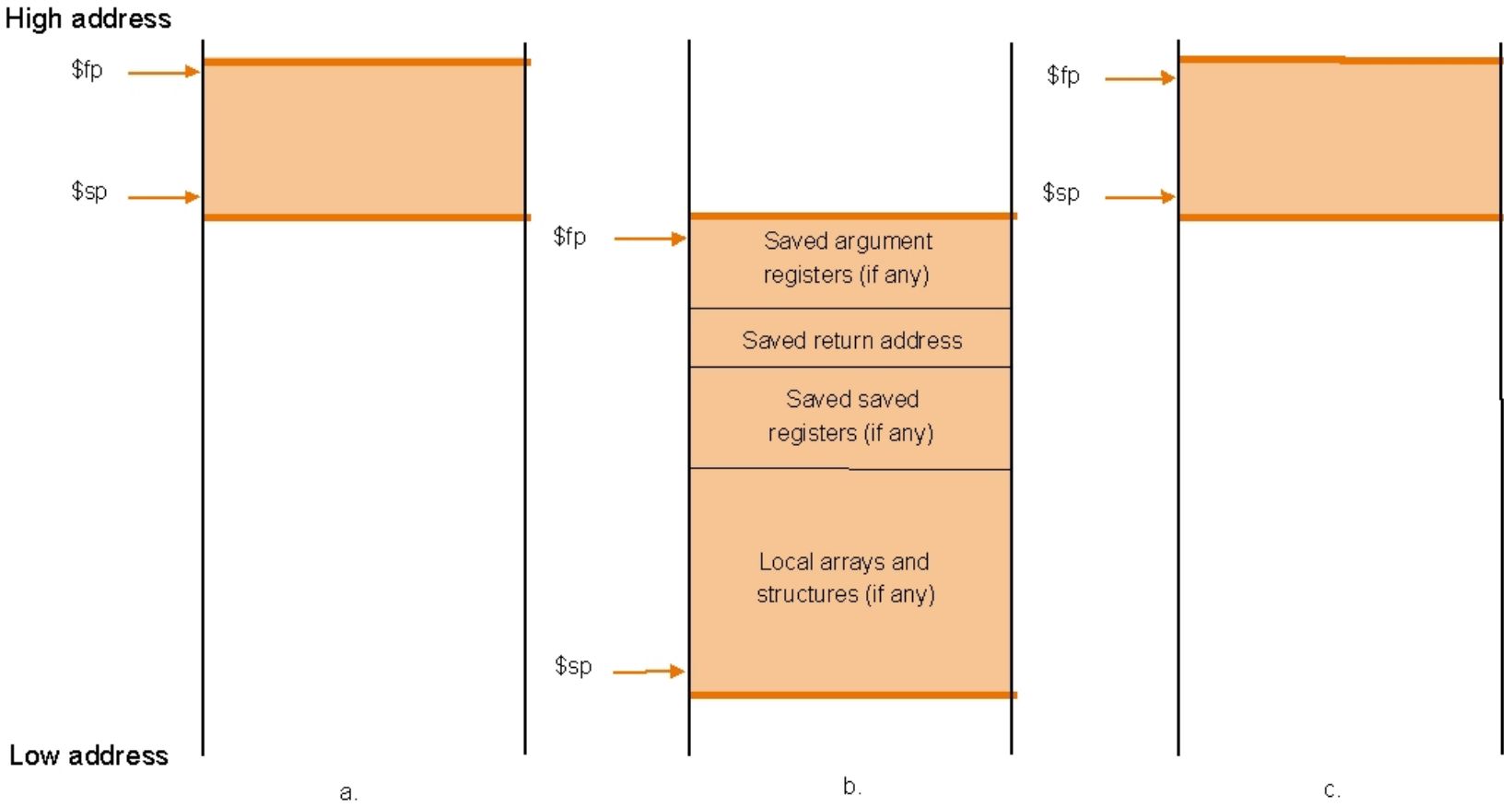


MIPS Registers

- $at: may be used by the assembler at any time; unsafe to use
- $k0-$k1: may be used by the OS at any time; unsafe to use
- $gp, $fp : don't worry about them
  - Read more about them in Appendix B (section B.6)
  - You can write perfectly good MIPS code without them

	Number	Symbol
The constant 0	$0	$zero
Reserved for Assembler	$1	$at
Return Values	$2-$3	$v0-$v1
Arguments	$4-$7	$a0-$a3
Temporary	$8-$15	$t0-$t7
Saved	$16-$23	$s0-$s7
More Temporary	$24-$25	$t8-$t9
Used by Kernel	$26-27	$k0-$k1
Global Pointer	$28	$gp
Stack Pointer	$29	$sp
Frame Pointer	$30	$fp
Return Address	$31	$ra

Allocating Space for New Data



Stack pointer may change during the procedure, making procedure harder to understand

Frame pointer offers a stable base register within a procedure for local memory references.

Register Conventions When Calling Procedures

- CalleR: the calling function
- CalleE: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (jal) and which may be changed.
- Saved:
  - $0: No Change. Always 0.
  - $s0-$s7: Restore if you change. That's why they're called saved registers. If the callee changes these in any way, it must restore the original values before returning.
  - $sp: Restore if you change. The stack pointer must point to the same place before and after the jal call, or else the caller won't be able to restore values from stack.
- Volatile:
  - $ra: Can Change. The jal call itself will change this register. Caller needs to save on stack if nested call.
  - $v0-$v1: Can Change. These will contain the new returned values.
  - $a0-$a3: Can change. These are volatile argument registers. Caller needs to save if they'll need them after the call.
  - $t0-$t9: Can change. That's why they're called temporary: any procedure may change them at any time. Caller needs to save if they'll need them afterwards.

Register Conventions

- What do these conventions mean?
  - If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a jal call.
  - Function E must save any S (saved) registers it intends to use before modifying their values.
  - Remember: Caller/callee need to save only temporary/saved registers they are using, not all registers.
- Example:
  -
```
R: ...          #   R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
   ...          ### PUSH REGISTER(S) TO STACK?
   jal E        # Call E
   ...          # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
   jr $ra       # Return to caller of R

E: ...          # R/W $s0, $v0, $t0, $a0, $sp, $ra, mem
   jr $ra       # Return to R
```
- What does R have to push on the stack before "jal E"?
  - $s0, $sp, $v0, $t0, $a0, $ra?
- Answer: *R* needs to save any registers it will use after the function call, and which are not preserved in E by convention: $v0, $t0, $a0, $ra

Example: Fibonacci Numbers

- The Fibonacci numbers are defined as follows: F(n) = F(n – 1) + F(n – 2), where F(0)=1 and F(1)=1.
- In C:

```
int fib(int n) {
    if(n == 0) { return 1; }
    if(n == 1) { return 1; }
    return (fib(n - 1) + fib(n - 2));
}
```

- Compile into MIPS. Start by writing the prologue:
  - For now, need to save $ra register. If we need more later, come back and revise FRAMESIZE in prologue, and save registers to be modified:

prologue

```
FIB:    addi $sp, $sp, -FRAMESIZE    # space for FRAMESIZE>=4 words
        . . .                        # save more registers if any
        sw $ra, 0($sp)               # save return address
```

  - Write epilogue: (later modify FRAMESIZE, and registers to be restored)

epilogue

```
FINISH: lw $ra, 0($sp)               # restore $ra
        . . .                        # restore saved registers
        addi $sp, $sp, FRAMESIZE     # adjust stack pointer
        jr $ra                       # return to caller
```

Example: Fibonacci Numbers

- Translate first 2 if statements

```
int fib(int n) {
   if(n == 0) { return 1; }
   if(n == 1) { return 1; }
   return (fib(n - 1) + fib(n - 2));
}
```

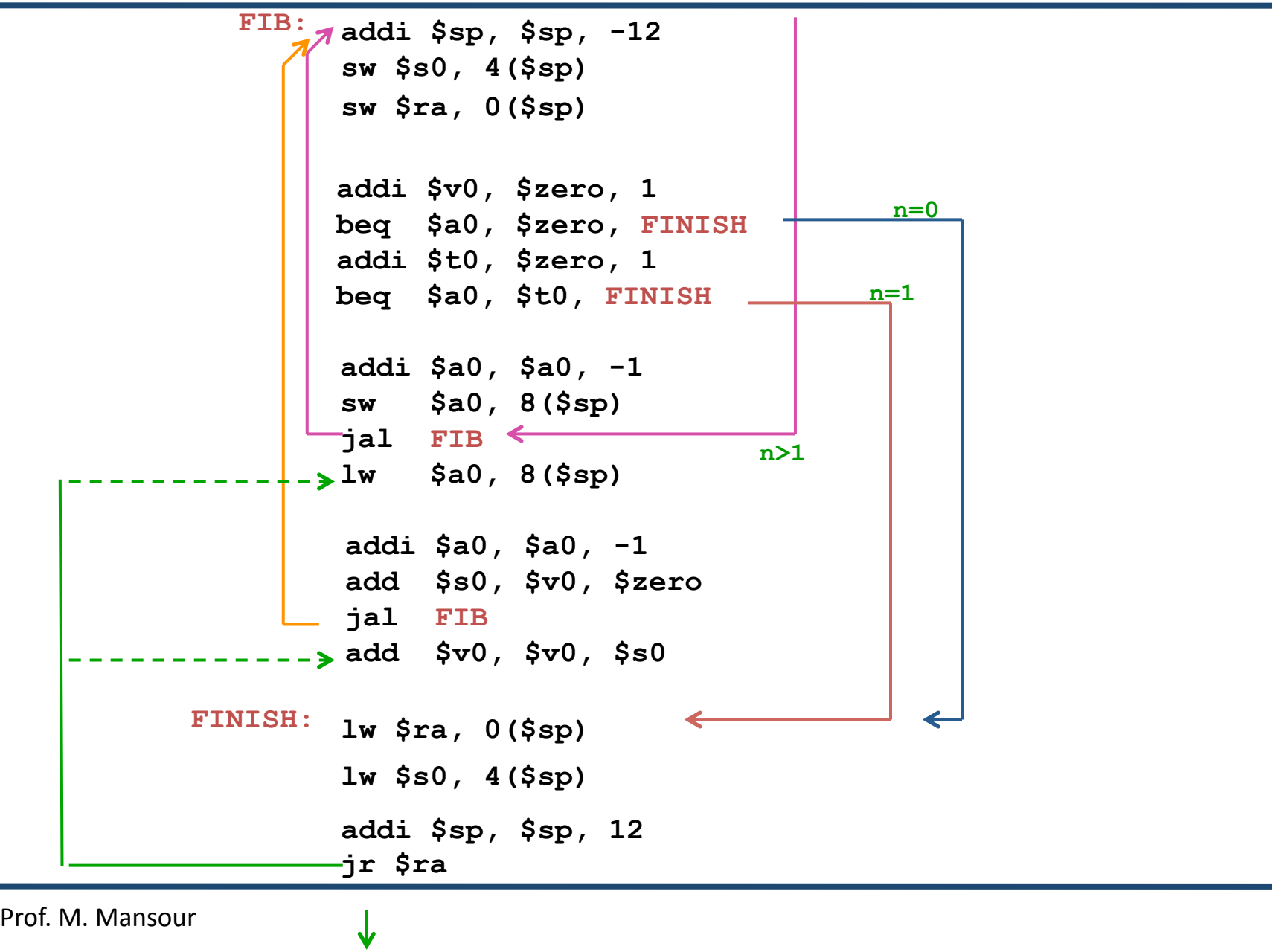
```
addi $v0, $zero, 1          # here we have set the return value
beq  $a0, $zero, FINISH     # if n == 0  (note argument n -> $a0)
addi $t0, $zero, 1
beq  $a0, $t0, FINISH       # if n == 1
```

- Next, translate the last return statement:

```
addi $a0, $a0, -1           # argument for fib(n-1)
sw   $a0, 8($sp)            # save n-1 on stack
jal  FIB                    # call fib(n-1)
lw   $a0, 8($sp)            # pop n-1 from stack
                            # note: return result of fib(n-1) is now in $v0
addi $a0, $a0, -1           # argument for fib(n-2)
add  $s0, $v0, $zero        # save returned value in $s0 (note we could have used also stack)
jal  FIB                    # call fib(n-2), return value in $v0
add  $v0, $v0, $s0          # fib(n-1) + fib(n-2)

# since $s0, is the only other registers to be saved on stack,
# set FRAMESIZE=12 in prologue/epilogue; add sw/lw for $s0
```

Example: Fibonacci Numbers



MIPS Instruction Summary So Far

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Conclusion

- Functions called with jal, return with jr $ra.
- The stack is your friend: Use it to save anything you need. Just be sure to leave it the way you found it.
- Instructions we know so far
  - Arithmetic/Logic: add, addi, sub, addu, addiu, subu, sll, srl
  - Memory: lw, sw, lb, sb, lbu
  - Decision: beq, bne, slt, slti, sltu, sltiu
  - Unconditional Branches (Jumps): j, jal, jr
- Registers we know so far
  - All of them!