LAST MIPS LECTURE REDO

LECTURE 10-1

JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN

A "REDO" OF THE SECOND HALF OF LAST WEDNESDAY

- LINKED LIST MIPS CODE
- SHIFTING A REGISTER'S BITS LEFT AND RIGHT
- MULTIPLICATION USING BASE TWO "SCHOOLBOOK METHOD"
- CALL STACK AND CALLING CONVENTIONS

THIS WEEK

- **HOMEWORK 09 ASSIGNED TONIGHT,** DUE NEXT MONDAY.
- **NO LABS TOMORROW.** (Happy election day.)
- WILL HOLD OFFICE HOURS 9:30-10:30, 3:30-4:30. (See the Slack.)
- **WEDNESDAY'S LECTURE WILL BE OVER ZOOM ONLY**

TRAVERSING A LINKED LIST

- MIPS code that outputs a linked list

```
print:
      move    $s1, $s0          # current = first;
print_loop:
      beqz    $s1, done         # if current==nullptr goto done;
print_data:
      lw      $a0, ($s1)        # print(current->data);
      li      $v0, 1
      syscall
      lw      $s1, 4($s1)       # current = current->next;
      b       print_loop
done:
```

- Check out my sample "inorder.s" that builds a linked list in sorted order.

TRAVERSING A LINKED LIST

- MIPS code that outputs a linked list

```
print:
    move    $s1, $s0        # current = first;
print_loop:
    beqz    $s1, done       # if current==nullptr goto done;
print_data:
    lw      $a0, ($s1)      # print(current->data);
    li      $v0, 1
    syscall
    lw      $s1, 4($s1)     # current = current->next;
    b       print_loop
done:
```

- Check out my sample "inorder.s" that builds a linked list in sorted order.

SAMPLE MIPS: INORDER.S

- This starting code sets up the data for 6 linked list nodes

```
        .data
next:       .asciiz "next\n"
eoln:       .asciiz "\n"
num_nodes:  .word   6
nodes:      .word   35, 0x0000, 6, 0x0000, 17, 0x0000,
                    3, 0x0000, 132, 0x0000, 20, 0x0000
        .globl main
        .text
```

- It then uses $a1 to point to the first, and scans through the rest using $a2.

```
main:
        la      $a1, nodes          # first = the first node
        addi    $a2, $a1, 8         # others = first + (8 bytes)
        lw      $a3, num_nodes      #
        addi    $a3, $a3, -1        # to_insert = num_nodes-1
insert_each:
        beqz    $a3, done_insert    # if to_insert == 0 goto done_insert
```

- We use $a3 to keep track of how many items still need to be inserted.

SAMPLE MIPS: INORDER.S

- This sets up $t3 to hold data for a value to be inserted

```
insert_in_order:
        lw      $t3, ($a2)              # load node->data
```

- This sets up $t4 and $t5 as list traversal pointers. Then we scan the list for the insertion place.

```
        move    $t4, $a1                # curr  = first
        li      $t5, 0x0000             # prev  = null
find_place:
        beqz    $t4, insert             # if curr == nullptr go to insert
        lw      $t6, ($t4)              # load curr->data
        ble     $t3, $t6, insert        # if node->data < curr->data
                                        #                    go to insert
        move    $t5, $t4                # prev = curr
        lw      $t4, 4($t4)             # curr = curr->next
        b       find_place
```

SAMPLE MIPS: INORDER.S

- In the code below we either update **`first`**, or update **`prev->next`**

```
insert:
        addi    $a3, $a3, -1            # to_insert -= 1
        sw      $t4, 4($a2)             # node->next = curr
        beqz    $t5, insert_in_front    # if prev == nullptr
                                        #    go to insert_at_front
insert_middle:
        sw      $a2, 4($t5)             # prev->next = node
        b       bump_node
insert_in_front:
        move    $a1, $a2                # first = node
```

- This code advances our pointer within the nodes array within **`.data`**.

```
bump_node:
        addi    $a2, $a2, 8       # node = next node in the node array
        b       insert_each
```

MULTIPLICATION

- Consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?

MULTIPLICATION

- Consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition.

MULTIPLICATION

- Consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition. *Not how multiplication is perfomed. Too slow.*

MULTIPLICATION

- Consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition. *Not how multiplication is perfomed. Too slow.*
- A2: Use the MIPS **`MULT`** instruction, along with **`MFLO`** and **`MFHI`**

MULTIPLICATION

- Consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition. *Not how multiplication is perfomed. Too slow.*
- A2: Use the MIPS **`MULT`** instruction, along with **`MFLO`** and **`MFHI`**
- **A3: That's probably the best way. But let's consider a third way...**

ANSWER 3: USE BIT SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
   111   <= binary for the value 7
  1110   <= binary for the value 2*7=14
111000   <= binary for the value 8*7=56
```

ANSWER 3: USE BIT SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
   111    <= binary for the value 7
  1110    <= binary for the value 2*7=14
111000    <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?

ANSWER 3: USE BIT SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
   111   <= binary for the value 7
  1110   <= binary for the value 2*7=14
111000   <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?
- NOTE: $10\,x = (2+8)\,x = 2\,x + 8\,x$

ANSWER 3: USE BIT SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
   111   <= binary for the value 7
  1110   <= binary for the value 2*7=14
111000   <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?
- NOTE: $10\,x = (2+8)\,x = 2\,x + 8\,x$
- **A:** We can multiply by 2, then by 8, and sum the two results.
- **I.E...**

ANSWER 3: USE BIT SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
   111    <= binary for the value 7
  1110    <= binary for the value 2*7=14
111000    <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?
- NOTE: $10\,x = (2+8)\,x = 2\,x + 8\,x$
- **A:** We can multiply by 2, then by 8, and sum the two results.
- **I.E.** We can shift left one bit and also shift left three bits. Then add.

ANSWER 3: USE BIT SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3
addu $t0,$t1,$t2
```

ANSWER 3: USE BIT SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3              tmp = tmp * 10
addu $t0,$t1,$t2
```

- It has the effect of multiplying t0 by 10.

ANSWER 3: USE BIT SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3                tmp = tmp * 10
addu $t0,$t1,$t2
```

- It has the effect of multiplying t0 by 10.
- **Q:** So how might we multiply by 100?

ANSWER 3: USE BIT SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3                    tmp = tmp * 10
addu $t0,$t1,$t2
```

- It has the effect of multiplying t0 by 10.
- **Q:** So how might we multiply by 100?
- SAME IDEA: 100 = 64 + 32 + 4
- **A:** So we shift 2, 5, and 6 places left. Add.

MULTIPLICATION BY 100

- The code below multiplies t0 by 100:

```
sll  $t1,$t0,2
sll  $t2,$t0,5
sll  $t3,$t0,6
addu $t0,$t1,$t2
addu $t0,$t0,$t3
```

SHIFTING BITS LEFT (LOGICAL)

SHIFT A REGISTER'S BITS **LEFT** SOME NUMBER OF POSITIONS

`SLL` *destination* , *positions*

- NOTES:
  - This is a "shift logical value left"
  - The bits of the ***destination*** are shifted left, with the leftmost bits "lost."
  - The rightmost bits shifted into the register are `000..00`
  - It's a multiplication by $2^{positions}$ but with limited precision.

SHIFTING BITS RIGHT (LOGICAL)

SHIFT A REGISTER'S BITS **LEFT** SOME NUMBER OF POSITIONS

`SRL` *destination* , *positions*

- NOTES:
  - This is a "shift logical value right"
  - The bits of the ***destination*** are shifted right, with the rightmost bits "lost."
  - The leftmost bits shifted into the register are `000..00`
  - It's a division by $2^{positions}$ but with limited precision, *treating the number as an unsigned value.*

SHIFTING BITS RIGHT (ARITHMETIC)

SHIFT A REGISTER'S BITS **RIGHT** SOME NUMBER OF POSITIONS

`SRA` *destination* , *positions*

- NOTES:
  - This is a "shift arithmetic value right"
  - The bits of the ***destination*** are shifted right, with the rightmost bits "lost."
  - The leftmost bits shifted in are `sss..ss` where `s` is the sign bit.
  - It's a division by $2^{positions}$ with limited precision, *treating the number as a two's complement encoded integer.*

SAMPLE MIPS: BITSINREVERSE.S

- This outputs the bits of a register's value, in reverse order:

```
output_loop:
        beqz    $t1, done          # if y == 0 go to done
        andi    $t0, $t1, 1        # bit = x % 2
output_bit:
        li      $v0, 1             # print(bit)
        move    $a0, $t0           #
        syscall                    #
shift_right:
        sra     $t1, $t1, 1        # x /= 2
        b       output_loop
done:
```

SAMPLE MIPS: BITS.S

- This outputs the bits of a register's value in the correct order:

```
   li      $t2, 0x80000000  # set up the bit mask
   li      $t4, 0           # start = false
output_loop:
   beqz    $t2, done        # if mask == 0 go to done
   and     $t0, $t1, $t2    # extract the bit using the bit mask
   li     $t3, 0            # bit = 0
   beqz    $t0, after_set_1
   li      $t3, 1           # bit = 1
   li      $t4, 1           # start = true
after_set_1:
   beqz    $t4, shift_right
output_bit:
   li      $v0, 1           # print(bit)
   move    $a0, $t3         #
   syscall                  #
shift_right:
   srl     $t2, $t2, 1      # shift the bit mask right
   b   output_loop
done:
```

SCHOOLBOOK MULTIPLICATION IN BINARY

- Suppose we want to multiply 34 by 11 using binary notation:

SAMPLE MIPS: MULTIPLY.S

- The resulting "schoolbook" code is surprisingly compact

```
multiply:
        li      $t0, 0              # product = 0
multiply_loop:
        beqz    $t2, report         # if y == 0 go to report
        andi    $t3, $t2, 1         # bit = y % 2
        beqz    $t3, skip           # if bit == 0 go to skip
        add     $t0, $t0, $t1       # sum += x
skip:
        sll     $t1, $t1, 1         # x *= 2
        sra     $t2, $t2, 1         # y /= 2
        b       multiply_loop
report:
```

FUNCTION CALLS IN MIPS

The MIPS ystem calls hint at a more general mechanism we need, namely...

Q: How do we mimic C++'s function calling mechanism in MIPS?

A: By following the MIPS function calling conventions and stack discipline.

OUTLINE:

- SOME SIMPLE C++ EXAMPLES
- CALL/RETURN WITH JAL/JR ; PARAMETER PASSING
- CREATE/PUSH AND TAKE-DOWN/POP OF STACK FRAME
- EXAMINE CONVENTIONS FOR SAVING REGISTERS' VALUES ON THE FRAME

RECALL: FUNCTIONS IN C++

- We considered this C++ program:

```
int two_digits(int tens, int ones) {
    return 10 * tens + ones;
}
int times100(int number) {
    return 100 * number;
}
int main(void) { int A, B, C, D;
    cin >> A;
    cin >> B;
    cin >> C;
    cin >> D;
    int hi = two_digits(A,B);
    int lo = two_digits(C,D);
    int n = times100(hi) + lo;
    cout << n << endl;
}
```

RECALL: FUNCTIONS IN C++

- We considered this C++ program:

```
int two_digits(int tens, int ones) {
    return 10 * tens + ones;
}
int times100(int number) {
    return 100 * number;
}
int main(void) { int A, B, C, D;
    cin >> A;
    cin >> B;
    cin >> C;
    cin >> D;
    int hi = two_digits(A,B);
    int lo = two_digits(C,D);
    int n = times100(hi) + lo;
    cout << n << endl;
}
```

RECALL: FUNCTIONS IN C++

- We considered this C++ code

```
int two_digits(int tens, int ones) {
    return 10 * tens + ones;
}
int times100(int number) {
    return 100 * number;
}
int four_digits(int w,int x,int y,int z) {
    return times100(two_digits(w,x)) + two_digits(y,z);
}
int main(void) { int A, B, C, D;
    cin >> A;
    cin >> B;
    cin >> C;
    cin >> D;
    cout << four_digits(A,B,C,D) << endl;
}
```

- We're going to work to convert this and the earlier example into MIPS code.

JUMPING FOR CALL AND RETURN

- There are two "jump" instructions used to call and return from functions:
  - `JAL` ***label***
    - ➡This jumps to the callee code at that ***label***.
    - ➡It saves the *return address* into register **$ra**
    - ➡The return address is for the caller's instruction just below the `JAL`.
  - `JR` **$ra**
    - ➡This jumps from the callee back to the instruction below the call site.
    - ➡The caller then continues executing.

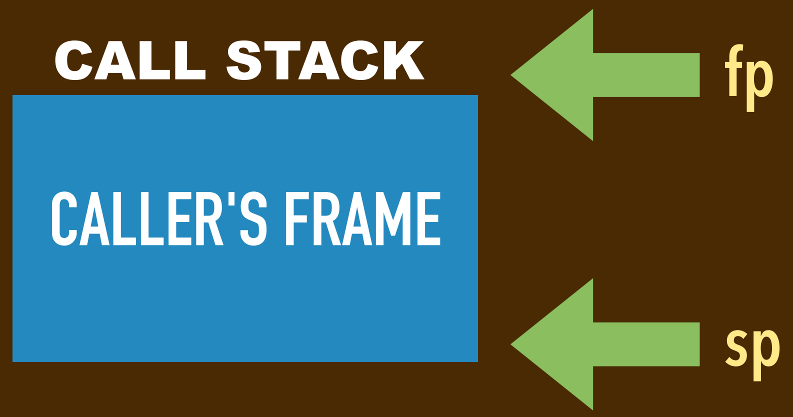
SPECIAL REGISTERS IN MIPS

- There are several conventions for registers in MIPS:
  - Registers **$a0**-**$a3** hold the arguments for the call. (The first 16 bytes.)
  - Registers **$v0**-**$v1** hold the result of the call.
  - Registers **$ra** holds the return address of the call.
  - Registers **$fp** and **$fp** mark the top and bottom of the stack frame.

STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register **sp** points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

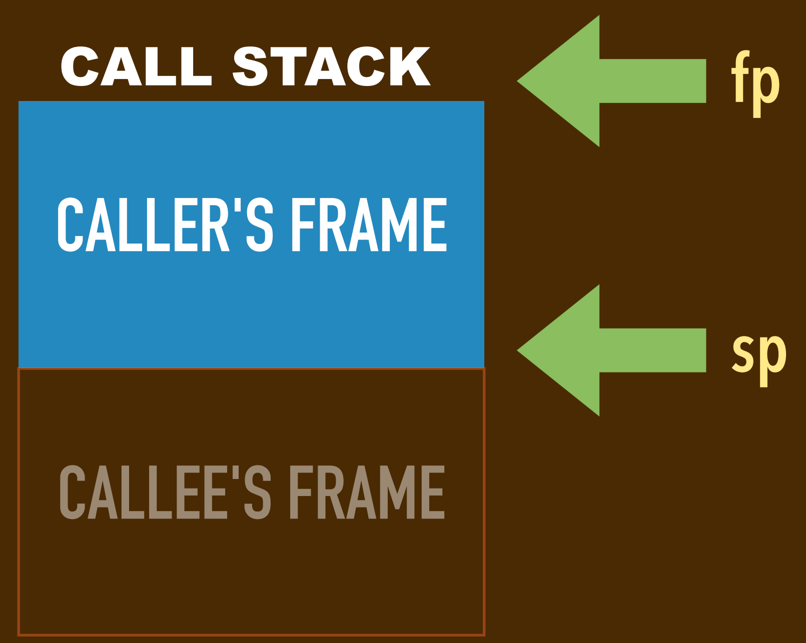
BEFORE THE CALL



STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register **sp** points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

BEFORE THE CALL



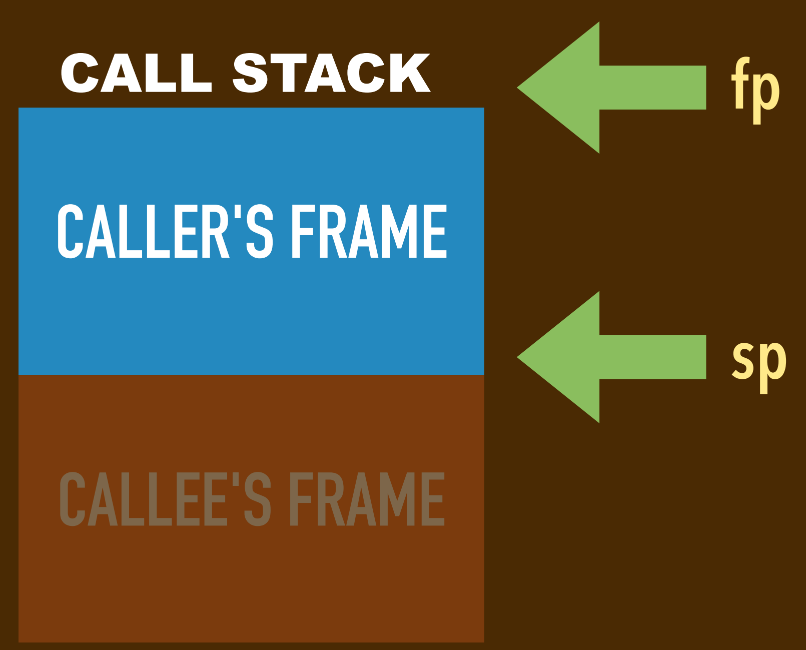
STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register **sp** points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register **sp** points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

AFTER THE CALL



STACK FRAME DISCIPLINE (CONT'D)

- The MIPS calling conventions designate that...
  - the frame size should be at least 32 bytes
  - the addresses in fp and sp should be ***word-aligned*** (multiples of 4).
  - (some say they should be ***double-word aligned*** (multiples of 8)

CODE STRUCTURE

- Every call site has a prologue and an epilogue:
  - The caller's prologue saves registers and sets up arguments.
  - Its epilogue gets the return value and restores saved registers.
- Every function's code has a prologue and an epilogue:
  - The callee's prologue sets up its frame, saves registers, grabs arguments.
  - Its epilogue restores registers, takes down the frame, sets the return value.

CALLEE-SAVED REGISTERS

- The MIPS calling conventions designate that...
  - Registers need to be preserved with a function call. No clobbering!
- Some registers are "***callee-saved***"
  - ➡ The function called must save the values of these registers on the stack before using them.
  - ➡ It must restore their values from the stack before it retuns to the caller.
  - These registers' values are guaranteed to be preserved with a function call.

CALLER-SAVED REGISTERS

- The MIPS calling conventions designate that...
  - Registers need to be preserved with a function call. No clobbering!
- Some registers are "***caller-saved***"
  - ➡ The caller saves these on the stack before calling a function.
  - ➡ The caller restores them from the stack after the call.
  - These registers' values may not be preserved with a function call.

FOUR_DIGITS IN MIPS USING T REGISTERS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp)
    sw   $a3,-24($fp)
    jal  two_digits
    move $t0,$v0
    sw   $t0,-12($fp)
    lw   $a0,-20($fp)
    lw   $a1,-24($fp)
    jal  two_digits
    move $t1,$v0
    sw   $t1,-16($fp)
    lw   $t0,-12($fp)
    move $a0,$t0
    jal  times100
    lw   $t1,-16($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

MIPS CALLING CONVENTIONS SUMMARY: THE CALLER

- Before the caller calls a function...
  - It saves caller-saved registers (a0-a3, t0-t9) onto its stack frame.
  - It places the parameters into registers a0-a3.
  - It pushes 5th, 6th, etc parameters onto the bottom of its stack frame.
- Using `JAL` saves a return address to register ra.
- After the function is called...
  - The caller restores registers it has saved, if needed.
  - It extracts the return value from register v0 and v1.

MIPS CALLING CONVENTIONS SUMMARY: THE CALLEE

- When a function is called…
  - It saves callee-saved registers (fp, sp, ra, s0-s7) onto its stack frame.
  - It extracts argument registers a0-a3 and from slots just above its frame.
  - It normally sets fp to the old sp, subtracts an offset from sp.
    - ➡ The offset it chooses is the callee's frame size. It has to be a multiple of 8.
- Before a function returns…
  - It puts the return value into register v0 and v1.
  - It restores registers for the caller, including fp, sp, and ra.
- It then performs `JR $RA` to return control back to the caller.

FOUR_DIGITS IN MIPS WITH SOME CLEAN-UP

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-16($fp)
    sw   $a3,-20($fp)
    jal  two_digits
    sw   $v0,-12($fp)
    lw   $a0,-16($fp)
    lw   $a1,-20($fp)
    jal  two_digits
    lw   $a0,-12($fp)
    sw   $v0,-12($fp)
    jal  times100
    lw   $t1,-12($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

CAN SEE COMPILER BEHAVIOR ONLINE

`Check out https://godbolt.org/`