MIPS Functions

Common Problems on Homework

- 1.3: Convert -3000_{ten} to binary in 8bit, 16bit, and 32bit
  - Even though it overflows with 8bits, there is plenty of room with 16 and 32 bit.

Common Problems on Homework

- 2.3: Convert $01111111111111111111111110111111_{two}$ to decimal
  - It says that it is in two's complement but that is just a standard to distinguish positive numbers from negative. You don't have to do two's complement conversion. That's used to see what the decimal equivalent is or to convert an unsigned number to a signed one

Common Problems on Homework

- 3.3: Convert -0.3_{ten} to a floating-point binary number
  - In single precision, you would get 0xBE999999 and not 0xBE99999A because when you run out of hardware, instead of rounding, you truncate the number.

In Class Exercise 2

Write the MIPS assembly code to do insertion sort (shown by the following C code segment). You may use any MIPS instructions that you've learned inside or outside of class.

```
int main() {
        int i, j, v;
        int A[10] = {6, 3, 7, 2, 0, 9, 1, 8, 4, 5};
        for (i = 1; i < 10; ++i) {
                v = A[i];
                for (j = i - 1; j >= 0 && A[j] >= v; --j) {
                        A[j+1] = A[j];
                }
                A[j+ 1] = v;
        }
        return 0;
}
```

In Class Exercise 2 Solution

```
            .data
A:          .word 6, 3, 7, 2, 0, 9, 1, 8, 4, 5

            .text
            .globl main
main:       la $s0, A               # Array A
            li $s1, 10              # Length of A
            li $s2, 1               # i
            li $s3, 0               # j
            li $s4, 0               # v
            li $t0, 0               # Address of A[i]
            li $t1, 0               # Address of A[j]
            li $t2, 0               # Value of A[i]
            li $t3, 0               # Value of A[j]

Exit:       li $v0, 10              # Load exit operation
            syscall                 # Exit
```

Set up program

Initialize ‘variables’

In Class Exercise 2 Solution

```
            .data
A:          .word 6, 3, 7, 2, 0, 9, 1, 8, 4, 5

            .text
            .globl main
main:       la $s0, A                   # Array A
            li $s1, 10                  # Length of A
            li $s2, 1                   # i
            li $s3, 0                   # j
            li $s4, 0                   # v
            li $t0, 0                   # Address of A[i]
            li $t1, 0                   # Address of A[j]
            li $t2, 0                   # Value of A[i]
            li $t3, 0                   # Value of A[j]
Loop1:


            addi $s2, $s2, 1            # ++i
            blt $s2, $s1, Loop1         # Loop while i < 10

Exit:       li $v0, 10                  # Load exit operation
            syscall                     # Exit
```

Set up outer loop

In Class Exercise 2 Solution

```
            .data
A:          .word 6, 3, 7, 2, 0, 9, 1, 8, 4, 5

            .text
            .globl main
main:       la $s0, A               # Array A
            li $s1, 10              # Length of A
            li $s2, 1               # i
            li $s3, 0               # j
            li $s4, 0               # v
            li $t0, 0               # Address of A[i]
            li $t1, 0               # Address of A[j]
            li $t2, 0               # Value of A[i]
            li $t3, 0               # Value of A[j]
Loop1:
            addi $s3, $s2, -1       # j = i - 1

Loop2:
            addi $s3, $s3, -1       # --j
            bge $s3, $0, Loop2      # Loop if j >= 0

            addi $s2, $s2, 1        # ++i
            blt $s2, $s1, Loop1     # Loop while i < 10

Exit:       li $v0, 10              # Load exit operation
            syscall                 # Exit
```

Set up Inner Loop

In Class Exercise 2 Solution

```
            .data
A:          .word 6, 3, 7, 2, 0, 9, 1, 8, 4, 5

            .text
            .globl main
main:       la $s0, A                   # Array A
            li $s1, 10                  # Length of A
            li $s2, 1                   # i
            li $s3, 0                   # j
            li $s4, 0                   # v
            li $t0, 0                   # Address of A[i]
            li $t1, 0                   # Address of A[j]
            li $t2, 0                   # Value of A[i]
            li $t3, 0                   # Value of A[j]
Loop1:      sll $t0, $s2, 2             # Convert i to memory access
            add $t0, $t0, $s0           # Add offset i to base A
            lw $s4, 0($t0)              # v = A[i]
            addi $s3, $s2, -1           # j = i - 1

Loop2:
            addi $s3, $s3, -1           # --j
            bge $s3, $0, Loop2          # Loop if j >= 0

            addi $s2, $s2, 1            # ++i
            blt $s2, $s1, Loop1         # Loop while i < 10

Exit:       li $v0, 10                  # Load exit operation
            syscall                     # Exit
```

Load A[i] into v

In Class Exercise 2 Solution

```
            .data
A:          .word 6, 3, 7, 2, 0, 9, 1, 8, 4, 5

            .text
            .globl main
main:       la $s0, A               # Array A
            li $s1, 10              # Length of A
            li $s2, 1               # i
            li $s3, 0               # j
            li $s4, 0               # v
            li $t0, 0               # Address of A[i]
            li $t1, 0               # Address of A[j]
            li $t2, 0               # Value of A[i]
            li $t3, 0               # Value of A[j]
Loop1:      sll $t0, $s2, 2         # Convert i to memory access
            add $t0, $t0, $s0       # Add offset i to base A
            lw $s4, 0($t0)          # v = A[i]
            addi $s3, $s2, -1       # j = i - 1
            sll $t1, $s3, 2         # Convert j to memory access
            add $t1, $t1, $s0       # Add offset j to base A
Loop2:      lw $t3, 0($t1)          # load A[j]

            addi $s3, $s3, -1       # --j
            addi $t1, $t1, -4       # Keep memory access consistent with j
            bge $s3, $0, Loop2      # Loop if j >= 0

            addi $s2, $s2, 1        # ++i
            blt $s2, $s1, Loop1     # Loop while i < 10

Exit:       li $v0, 10              # Load exit operation
            syscall                 # Exit
```

Load A[j]

In Class Exercise 2 Solution

```
            .data
A:          .word 6, 3, 7, 2, 0, 9, 1, 8, 4, 5

            .text
            .globl main
main:       la $s0, A                   # Array A
            li $s1, 10                  # Length of A
            li $s2, 1                   # i
            li $s3, 0                   # j
            li $s4, 0                   # v
            li $t0, 0                   # Address of A[i]
            li $t1, 0                   # Address of A[j]
            li $t2, 0                   # Value of A[i]
            li $t3, 0                   # Value of A[j]
Loop1:      sll $t0, $s2, 2             # Convert i to memory access
            add $t0, $t0, $s0           # Add offset i to base A
            lw $s4, 0($t0)              # v = A[i]
            addi $s3, $s2, -1           # j = i - 1
            sll $t1, $s3, 2             # Convert j to memory access
            add $t1, $t1, $s0           # Add offset j to base A
Loop2:      lw $t3, 0($t1)              # load A[j]
            blt $t3, $s4, Break         # Continue looping if A[j] >= v

            addi $s3, $s3, -1           # --j
            addi $t1, $t1, -4           # Keep memory access consistent with j
            bge $s3, $0, Loop2          # Loop if j >= 0
Break:
            addi $s2, $s2, 1            # ++i
            blt $s2, $s1, Loop1         # Loop while i < 10

Exit:       li $v0, 10                  # Load exit operation
            syscall                     # Exit
```

Add second condition to inner loop

In Class Exercise 2 Solution

```
            .data
A:          .word 6, 3, 7, 2, 0, 9, 1, 8, 4, 5

            .text
            .globl main
main:       la $s0, A               # Array A
            li $s1, 10              # Length of A
            li $s2, 1               # i
            li $s3, 0               # j
            li $s4, 0               # v
            li $t0, 0               # Address of A[i]
            li $t1, 0               # Address of A[j]
            li $t2, 0               # Value of A[i]
            li $t3, 0               # Value of A[j]
Loop1:      sll $t0, $s2, 2         # Convert i to memory access
            add $t0, $t0, $s0       # Add offset i to base A
            lw $s4, 0($t0)          # v = A[i]
            addi $s3, $s2, -1       # j = i - 1
            sll $t1, $s3, 2         # Convert j to memory access
            add $t1, $t1, $s0       # Add offset j to base A
Loop2:      lw $t3, 0($t1)          # load A[j]
            blt $t3, $s4, Break     # Continue looping if A[j] >= v
            sw $t3, 4($t1)          # A[j+1] = A[j]
            addi $s3, $s3, -1       # --j
            addi $t1, $t1, -4       # Keep memory access consistent with j
            bge $s3, $0, Loop2      # Loop if j >= 0
Break:      sw $s4, 4($t1)          # A[j+1] = v
            addi $s2, $s2, 1        # ++i
            blt $s2, $s1, Loop1     # Loop while i < 10
Exit:       li $v0, 10              # Load exit operation
            syscall                 # Exit
```

Update A[j+1]

Procedures and Functions

- We programmers use procedures and functions to structure and organize programs
  - To make them easier to understand
  - To allow code to be reused

Function

- A function carries out a well-defined functionality, that can be called and produce the result to be used by the caller.

```
int addfun(int a, int b)
{
    int res = a+b;
    return res;
}

void main(void)
{
    int i=0;
    int j=100;
    int k = addfun(i,j);
}
```

Functions

- **A function is a consecutive piece of code stored in the memory.**
- To invoke (or call) a function, **we must go to that piece of code**. Then it does certain things, and get the result we need.
- What do we know about going to a piece of code?

Functions

- So, we can call a function by

  ```
  j Function
  ```

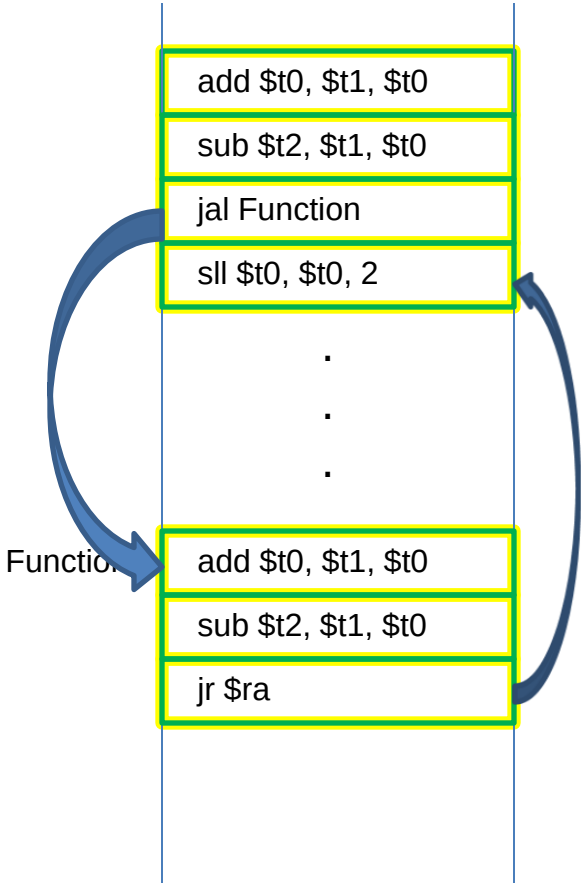
- And, the function code will do something we need.
- Problem: how to come back to the caller?

Two Interesting Instructions and One Interesting Register

- **`jal`**: jump and link
  - `jal L1`:
    - does **TWO** things
      1. Goto `L1`. (the next instruction to be executed is at address `L1`)
      2. Save the address of the next instruction in **`$ra.`** `$ra` is the interesting register that stores the **r**eturn **a**ddress
- **`jr $ra`**
  - Does **ONE** thing. Goto the instruction whose address is the value stored in `$ra`.
- This is **ALL** we need to support function calls in MIPS!

Functions

- The procedure



A simple function

```
int addfun(int a, int b)
{
    int res = a + b;
    return res;
}


int t = 0;
for (int i=0;i<10;i+=2)
{
    t += addfun(A[i],A[i+1]);
}
```

```
            .data
A:          .word 12, 34, 67, 1, 45, 90, 11, 33, 67, 19

            .text
            .globl main
main:
            la $s7, A
            li $s0, 0 #i
            li $s1, 0 #res
            li $s6, 9

loop:
            sll $t0, $s0, 2
            add $t0, $t0, $s7
            lw $a0, 0($t0)
            lw $a1, 4($t0)
            jal addfun
            add $s1, $s1, $v0
            addi $s0, $s0, 2
            bgt $s0, $s6, done
            j loop

done:
            li $v0,10
            syscall


addfun:
            add $v0, $a0, $a1
            jr $ra
```

Key things to keep in mind

1. A function is just a segment of code stored sequentially in the memory. To call a function is just to **go there**.
2. The name of a function in MIPS is **JUST A LABEL or JUST AN ADDRESS**.
3. We cannot simply use “j addfun” to go to addfun, because we do not know where come back. Therefore, we need to store the address of the instruction that should be executed after going to the function somewhere, and in MIPS, it is $ra.
4. At the end of a function, we write “jr $ra”, to **go back**.

MIPS Calling Conventions

- MIPS assembly follows the following convention in using registers
  - $a0 - $a3: four argument registers in which to pass parameters
  - $v0 - $v1: two value registers in which to return values
  - $ra: one return address register to return to the point of origin

MIPS Conventions

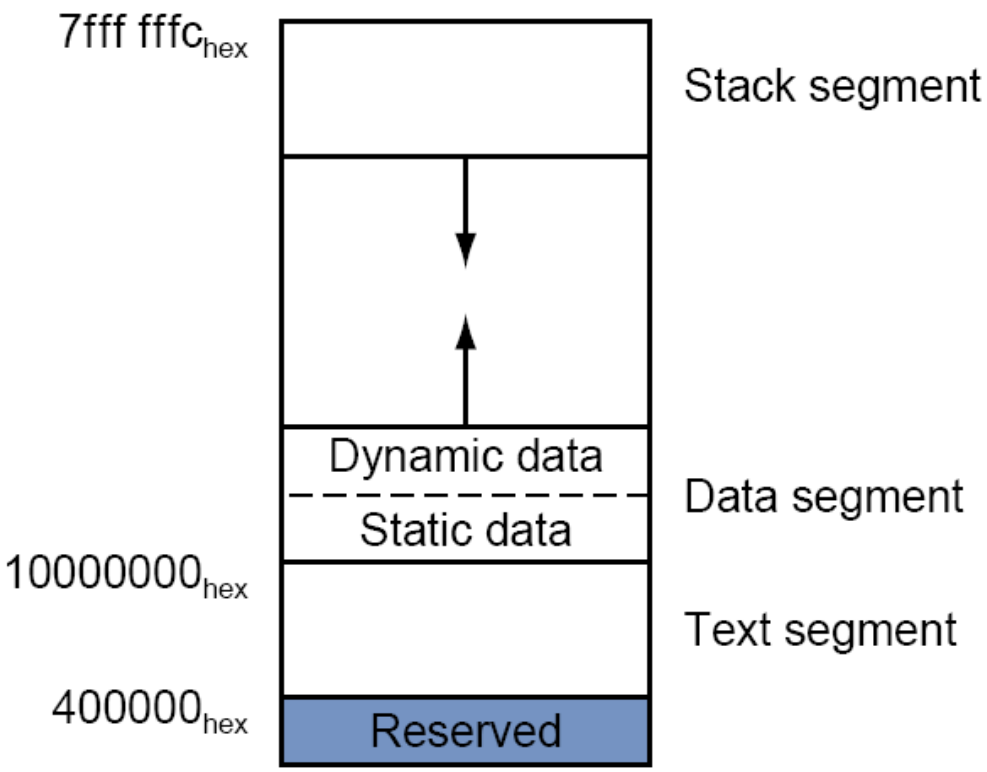
- Quite often, our function needs to use some registers to do dome calculation. So we will modify the values of them.
- We can use $t0-$t9 freely inside a function, because the caller does not expect the values inside $t0-$t9 to stay the same after the function call.
- But, the caller do expect the values in $s0 to $s7 to be the same after a function call.

MIPS Conventions

- So, just try to avoid using $s0 and $s7 inside a function whenever possible.
- But what if do need it? Such occasions will arise…

Stack

- So, if we do have to use $s0 - $s7, we MUST save it somewhere before entering the main part of the function, and restore it before you return (before we execute “jr $ra”).
- In MIPS, we save them in the **stack**.
- Stack is a part in the memory allocated for functions. It starts at 0x7ffffffc and grows **down** as we add more stuffs to it.
- Stack is “first in last out.”



$sp

- The top address of the stack, the address of the first word that is storing value, is (should be) always be stored in `$sp`.
- So, adding a word into the stack (pushing a word onto the stack) is a two-step thing, because **you** have to maintain the correctness of $sp:
  - `addi $sp, $sp, -4`
  - `sw $s0, 0($sp)`

Suppose we want to

```
int weirdfun(int a, int b)
{
    int res;
    res = a + a + b - a / 2;
    return res;
}
```

```
int t = 0;
for (int i=0;i<10;i+=2)
{
    t += weirdfun(A[i],A[i+1]);
}
```

Stack and $sp

- Suppose we want to store a/2 in $s0.
  - How do we get a/2?
- At the beginning, we do
  - `addi $sp, $sp, -4`
  - `sw $s0, 0($sp)`
- At the end, we do
  - `lw $s0, 0($sp)`
  - `addi $sp, $sp, 4`

```
                .data
A:              .word 12, 34, 67, 1, 45, 90, 11, 33, 67, 19

                .text
                .globl main
main:
                la $s7, A
                li $s0, 0 #i
                li $s1, 0 #res
                li $s6, 9

loop:
                sll $t0, $s0, 2
                add $t0, $t0, $s7
                lw $a0, 0($t0)
                lw $a1, 4($t0)
                jal weirdfun
                add $s1, $s1, $v0
                addi $s0, $s0, 2
                bgt $s0, $s6, done
                j loop

done:
                li $v0,10
                syscall


weirdfun:
                addi $sp, $sp, -4
                sw $s0, 0($sp)

                srl $s0, $a0, 1
                add $t0, $a0, $a0
                add $t0, $t0, $a1
                sub $t0, $t0, $s0

                ori $v0, $t0, 0

                lw $s0, 0($sp)
                addi $sp, $sp, 4

                jr $ra
```

Function calls inside a function

- What if we need to call another function inside a function? Will this work?

```
int twofun(int a, int b)
{
    int res;
    res = addfun(a,b) - a / 2;
    return res;
}
```

```
twofun:
        addi $sp, $sp, -4
        sw $s0, 0($sp)

        jal addfun
        srl $s0, $a0, 1
        sub $v0, $v0, $s0

        lw $s0, 0($sp)
        addi $sp, $sp, 4

        jr $ra
```

Function calls inside a function

- The problem is that the value of $ra is changed whenever you use jal somelabel.
- How to deal with it?

```
twofun:
        addi $sp, $sp, -4
        sw $s0, 0($sp)

        jal addfun
        srl $s0, $a0, 1
        sub $v0, $v0, $s0

        lw $s0, 0($sp)
        addi $sp, $sp, 4

        jr $ra
```

The working versions

```
twofun1:
   addi $sp, $sp, -4
   sw $s0, 0($sp)
   addi $sp, $sp, -4
   sw $ra, 0($sp)

   jal addfun
   srl $s0, $a0, 1
   sub $v0, $v0, $s0,

   lw $ra, 0($sp)
   addi $sp, $sp, 4
   lw $s0, 0($sp)
   addi $sp, $sp, 4

   jr $ra
```

```
twofun2:
        addi $sp, $sp, -8
        sw $s0, 4($sp)
        sw $ra, 0($sp)

        jal addfun
        srl $s0, $a0, 1
        sub $v0, $v0, $s0,

        lw $ra, 0($sp)
        lw $s0, 4($sp)
        addi $sp, $sp, 8

        jr $ra
```

Saving registers

- In case of nested function calls, before calling a function, to be safe, the caller should
  - save $t0-$t9
  - save $a0-$a3

  If such registers are needed later. and
  - save $ra

  Because $ra is going to be needed later and it will be changed
- The callee should
  - save $s0-s7