MIPS

- Mircroprocessor w/o Interlocked Pipelin Stages
    - A type of RISC (Reduced Instruction Set Computer) as opposite to the CISC such as the X86 micro-archaetecture.
    - RISC philosophy: Simple yet fast instructions, complecated stuff should be done by the assembler.
    - CISC philosophy: Complex instrutions that can do complex things.

Mips Instructions

- Instructions are written in the format `<instr> <parameter>`
- Each instruction takes its own line
- All instructions are 32 bits long. (that is, 4 bytes)
- Instruction address are measured in bytes, start point is address 0
    - **Thus, all instruction addresses are divisible by 4!**

Assembly Language Instructions

Converting from Assemby to Machine Code

Consider the operation `$t3 = $t1 + $t2`, which has assembly equivalent `add $t3, $t1, $t2`. Then, we can convert this into the equivalent machine code.

```
| Opcode |         Operand         |Function|
| 000000 | 01001 01010 01011 XXXXX | 100000 |
```

Notice that here the opcode is 6'b0, since this operation is a R-type operation. The XXXXX in the syntax part is marks don't care values for the `<shamt>`

R-type VS i-type arithmetic

In general, some intructions are R-type (meaning all operands are registers) and some are I-type (meaning they use an immediate/constant value in their operation). Here are some examples for each flavour

- R-Type: `add, addu, div, divu, mult, multu, sub, subu...`
- I-Type: `addi, addiu...`

Converting From Assembly to Machine Code, Take II

We should do an example where we use a I-type instruction. Consider `$t2 = $t1 + 42`, which shall translate into `addi $t2, $t1, 42`. Notice that here the actual operation that was carried out is `$t = $s + SignExtend(i = 42)`. The equivalent machine code is

```
| Opcode |    Operand    |      Immediate     |
| 001000 | 01001 01010 | 0000000000101010 |
```

Shift Instructions

Here is a list of common shift instructions

Insturction	Opcode/Function	Syntax	Operation
`sll`	000000	`$d, $t, $s`	`$d = $t << a`
`sllv`	000100	`$d, $t, $s`	`$d = $t << $s`
`sra`	000011	`$d, $t, a`	`$d = $t >> a`
`srav`	000111	`$d, $t, $s`	`$d = $t >> $s`
`srl`	000010	`$d, $t, a`	`$d = $s >>> a`
`srlv`	000110	`$d, $t, $s`	`$d = $t >>> $s`

Note:

- `srl` = "shift right logical"
- `sra` = "shift right arithmetic"
- `v` denotes a variable number of bits, specified by `$s`
- `a` is the shift amount, and is stored in the `shamt` when encoding the R-type machine code instructions

Data Movement Instructions

Insturction	Opecode/Functions	Syntax	Operation
`mfhi`	010000	`$d`	`$d = hi`
`mflo`	010010	`$d`	`$d = lo`
`mthi`	010001	`$s`	`hi = $s`
`mtlo`	010011	`$s`	`lo = $s`

These are R-type instructions for operating on the HI and LO registers described earlier.

ALU instructions

For the ALU instructions, most are R-type instructions. Thus, they mostly have the 6'b0 opcodes. **However, not all R-type instructions have I-type equivalent**. This is due to the philosophy of the RISC archaetecture, which states that an operation doesn't need an instruction it it can be performed through multiple existing operations.

Converting Programs and Assembly - Fibonacci Sequence Example

```
int fib(void) {
    int n = 10;
    int f1 = 1, f2 = -1;

    while (n != 0) {
        f1 = f1 + f2;
        f2 = f1 - f2;
        n = n - 1;
    }
    return f1;
}
```

In assembly code

```
# fib.asm
# register usage: $t3 = n, $t4 = f1, $t5 = f2

FIB:    addi $t3, $zero, 10     # Init to n = 10
            addi $t4, $zero, 1      # Init to f1 = 1
            addi $t5, $zero, -1     # Init to f2 = -1
LOOP:   beq $t3, $zero, END     # Until n = 0, then jump to END
            add $t4, $t4, $t5       # f1 = f1 + f2
            sub $t5, $t4, $t5       # f2 = f1 - f2
            addi $t3, $t3, -1       # n = n - 1
            j LOOP                  # REPEAT
END:    sb $t4, 0($sp)          # store the result
```

Control Flow in Assembly

- Not all programs follow a linear set of intructions.
  - Some operations require the code to brach to one section of code or another, e.g. IF/ELSE calls
  - Some require the code to jump back and repeat a section of code again (FOR/WHILE)
- For this, we have labels on the left hand side that indicate the points that the program flow might need to jump to.
  - Reference to these points the assembly code are resolved at compile time to offset values for the program counter.

Jump Instructions

Instruction	Opcode/Function	Syntax	Operation

Instruction	Opcode/Function	Syntax	Operation
`j`	000010	`label`	`pc = (pc & 0xF00000000) | (i << 2)`
`jal`	000011	`label`	`$31 = pc + 4; pc = (pc & 0xF00000000) | (i << 2)`
`jalr`	001001	`$s`	`$31 = pc + 4; pc = $s`
`jr`	001000	`$s`	`pc = $s`

- `jal` = "jump and link"
  - Register `$31` (aka `$ra`) stores the address that is used when returning from a subroutine
- **Note:** `jr` and `jalr` are jumps, but not J-type instructions.

More on `jr` and `jalr` (Jump to Registers)

- For instructions such as `jr $ra` and `jalr $t0`.
- The processor moves the address stored in `$ra` and `$t0` into the program counter.
  - The next instruction to be fetched will be at this new address, and the program will continue from there.

More on `j` and `jal` (Jump to Label)

- For `j` and `jal` instructions, the address is supplied by the instruction, and this can potentially cause a problem.
  - Problem:

  ```
  | 6'b opcode | 26'b address |
  ```

  If we have 32'b instructions and the first 6'b are occupied by the opcode, the remaining bits ***aren't*** enough for a entire address!
- **Work Around:**
  - Trailing Zeros: Since jump instructions load new addresses into the program counter, the values being loaded must be diisibe by 4.
    - Therefore, the binary values of these addresses will always end in "00".
    - Therefore, we are 100% sure that the last two bits are zero!
    - This, combined with the 26'b provided as input gives in total 28'b which is still not enough.
  - First Four bits: Several solutions exsits, we will dicuss the solution that MIPS adapts.
    - We simply keep the first four bits of the previous PC value, and this gave rise to the update rule we saw earlier.

    ```
    pc = (pc & 0xF00000000) | (i << 2)
    ```

        - Notice that the bitwise and part preserves first four bits of the previous program counter.
        - The last part adds to zeros
        - Bitwise OR them together produces the desired result.

Branch Instructions

Branching statements are used widely in IF statements and WHILE loops, or generally anything that requires the assembly code to jump around. **Note:** the labels are memory locations, assigned to each label at compile time.

Instruction	Opcode/Function	Syntax	Operation
`beq`	000100	`$s, $t, label`	`if ($s == $t) pc += i<< 2`
`bgtz`	000111	`$s, label`	`if ($s > 0) pc += i << 2`
`blez`	000110	`$s, label`	`if ($s <= 0) pc += i << 2`
`bne`	000101	`$s, $t, label`	`if ($s != $t) oc += i << 2`

Branch's Immediate (i) Value

- Branch statements are I-type instructions
- The immediate value (i) is a 16bit offset to add to the current instruction if the brach condition is satisfied (not the absolute address like with jumps)
    - Calculated as the difference between the current PC value and the address of the instruction youa re branchng to.
    - Stored here as number of instructions and not number of bytes
        - Again, not storing the trailing '00' if it is not necessary.
    - The i value can be positive (if you are jumping i instructions forward) or negative (if you are jumping i instructions backward)

Calculating the i value

- The offset is computed differently, depending on the implementation (i.e. if the PC is incremented by 4 before or after the brach offset calculation).
- In this course, we assume that i is computed as
    - `i = (label - (current PC)) >> 2`, i.e. PC is incremented first.
- Let's see an example where the i is calculated at compile time

```
.text
main:   addi $t0, $zero, 1
        beq $t0, $zero, END
        addi $t1, $zero, 1
END:    addi $t3, $zero, 1
```

We can see, using a simulator, that the immediate value for the `beq` call, which is END, is 2. (Since END is 2 instructions down from the branch instruction)

Conditional Branch Terms

- When the branch condition is met, we say the *branch is taken*
- When the branch condition is not met, we say that the *branch is not taken*.
    - in the case that the brach is not taken, the next value for the PC is `PC = PC + 4`, which is exacly the next instruction since each instruction is 4 bytes long (32'b).
- **Since branching relies on a 16bit value, branching distance is not unlimited**. Since the 16'b is signed, we have `+/- 15'b` to vary in total, which gives 2^16 total different values, corresponding to a range of 2^16 instructions.

Comparison Instructions

Instruction	Opcode	Syntax	Operation
`slt`	101010	`$d, $s, $t`	`$d = ($s < $t)`
`sltu`	101001	`$d, $s, $t`	`$d = ($s < $t)`
`slti`	001010	`$t, $s, i`	`$t = ($s < SE(i))`
`sltiu`	001011	`$t, $s, i`	`$t = ($s < ZE(i))`

- **Note:** Comparison operations stores a 1 in the destination register if the less-than comparison is true, and stores a zero in that location otherwise. This is not used very often, but useful in combination with branch instructions that only depend on one register. For example, `bgtz` branching on greater than. (You can do the comparison and store the result and use `bgtz` to branch on the computed 1/0 value.)

Using Branches and Jumps

If Statements

- If statements test a condition and then execute lines of code if the condition is true. For instance

```
if (i == j) {
    i++;
}
j += i;
```

- Testing conditions is done using either a `beq` instruction or a `bne` instruction.
- To achieve this, we can use the `bne` instruction to skip the `i++` steo and proceed to the `j += i` step.

```
```

```
# st1 = i, $t2 = j
main:   bne $t1, $t2, END
        addi $t1, $t1, 1
END:    add $t2, $t2, $t1
```

If/Else Statements

- Possible approach to if/else statements:
  - Test condition, and jump to `if` logic block whenever the condition is true.
  - Otherwise, perform the `else` logic block, and jump to the first line after `if` logic block.
- Example

  ```
  if (i == j)
      i++;
  else
      i--;
  j += i;
  ```

  Can be translated into the following

  ```
  # $t1 = i, $t2 = j
  IFPART:     bne $t1, $t2, ELSEPART
              addi $t1, $t1, 1
              j END
  ELSEPART:   addi $t1, $t1, -1
  END:        add $t2, $t2, $t1
  ```

Multiple If Statements

There are cases where the condition for the `if` statement has several conditions, such as the follows

```
if ( i == j || i == k )
    i ++;
else
    i --;
j = i + k;
```

we have the equivalent form in assembly

```
```

```
# $t1 = i, $t2 = j, $t3 = k
main:       beq $t1, $t2, IF        # equal -> short circuit behavior
            bne $t1, $t3, ELSE      # if equal, do nothing and jump to else
IF:         addi $t1, $t1, 1
            j END                   # skip the else part
ELSE:       addi $t1, $t1, -1
END:        add $t2, $t1, $t3
```

Above we looked at the case where `if` was done on two conditions `or`'ed together. We shall now see an example where it has a `and` condition.

```
if (i == j && i == k)
    i ++;
else
    j --;
j = i + k;
```

in assembly, this can be represented as

```
# $t1 = i, $t2 = j, $t3 = k
MAIN:       bne $t1, $t2, ELSE
            bne $t1, $t3, ELSE
            addi $t1, $t1, 1
            j END
ELSE:       addi $t1, $t1, -1
END:        add $t2, $t1, $t3
```

While Loops

Loops are, in some sense, similar to `if` statements. Here is the general recepe for the assembly procedure.

- Test if the loop condition fails
    - If it does, branch to the end
- Otherwise, execute the `while` loop contents
    - Make sure to update the loop condition values
- Jump back to the beginning Consider he following simple program

```
int i = 0;
while (i < 100) {
    i++;
}
```

we have assembly code equivalent

```
# $t0 = i, $t1 = 100 (constant variable to compare to)
main:       add $t0, $zero, $zero       # set $t0 to 0
            addi $t1. $zero, 100        # set $t1 to 100
START:      beq $t0, $t1, END           # while $t0 < $t1
            addi $t0, $t0, 1            #   $t0 = $t0 + 1
            j START                     #   jump back to the start
END:                                    # do nothing after finishing
```

For Loops

```
for ( <init> ; <cond> ; <update> ) {
    <for body content>
}
```

Cosinder the problem

```
for ( i = 0 ; i < 100 ; i++ ) {
    j  = j + i;
}
```

which translates into assembly as

```
# $t0 = i, $t1 = j, $t9 = 100 (constant var)
main:       add $t0, $zero, $zero       # Init $t0 <- 0
            add $t1, $zero, $zero       # Init $t1 <- 0
            addi $t9, $zero, 100        # Init $t9 <- 100
START:      beq $t0, $t9, EXIT
            add $t1, $t1, $t0
UPDATE:     addi $t0, $t0, 1
            j START
EXIT:
```

Note: Without the initialization and update sections, this is the same as a `while` loop.

Interactng With Memory

- All of the previous instructions perform operations on registers and immediate values, **what about momory?**

- All program must fetch values from memory into registers, operate on them, and then store the values back into memory.
- Memory operations are I-type, with the form

```
|Load/Store Operation|Load Data Register|offset(Address in Memory)|
|         lw         |       $t0        |         12 ($t0)        |
```

Load Versus Stores

- The terms "laod" and "store" are seen from the perspective of the processor, looking at memory
- Load are read operations
  - We load (**i.e., read**) from memory
  - We load a value from a memory address into a register
- Stores are write operations
  - We store (**i.e. write**) a data value from a register to a memory address.
  - Store instructions do not have a destination register, and therefore do not write to the register file.

Memory Instructions in MIPS Assembly

- Load and store instructions are I-type operations

```
| 6'b Opcode | 5'b rs | 5'b rt | 16'b immediate |
```

- Here is a nice way to organize all the possible assembly commands in this category, represented in regular expression

```
(load + store) (size of value) (signed or unsigned)
(l + s)        (w + h + b)     (u / <>)
```

  - `w = word`
  - `h = half`
  - `b = byte` **not bit!**
- Load and store instructions (omitted since not necessary to reproduce here)
- Above we have discussed a way to organize all possible commands, we will represent them using `<1><2><3>` to further discuss the trailling arguments

```
| Command Name | Destination, offset (access location) |
|   <1><2><3>  |      $t    ,       i($s)              |
```

The `access location` specifies the location to access as `MEM[$s + SE(i)]` while `Destination` stores the destination register for loads, source register for store. (These are register locations in the register file, so from the processos' perspective of view we load from memory into a destination register OR we write to memory using a source register as source)

Alignment Requirements

- Misaligned memory accesses result in errors
  - Word access should word aligned (divisible by four). This is used in addresses specified in a `lw` or `sw` instruction.
  - Half word access should only involve half-word aligned address (i.e., even addresses)
  - **No** constraints for byte access.

Notes on Memory

Big/Small Endian-ness

- **Big Endian**
  - The **most significant byte** of the word is stored first. The second most significant byte is stored at address that immediately follows and so on and so forth.
- **Small Endian**
  - The **least significant byte** of the word is stored first. The second least significant byte is stored at address that immediately follows and so on ans so dorth.

MIPS Endianness

- MIPS processors are bi-endian i.e., they can operate with either big/small endian byte order.

Reading From Devices

- The offset value is useful for objects or stack parameters, when multiple values are needed from a given memory location.
- Memory is also used to communicate with outside deices, such as keyboards and monitors
  - known as **memory mapped IO**
  - Invoked with a **trap** or **syscall** function

Trap Instructions

- Trap instructions send system calls to the operating system
  - For example, interacting with the user, and exiting the program etc.
- This is similar, but not quite the same, as compared to the `syscall` command.

Memory Segments and Syntax

- Program are divided into two main sections in memory:
  - `.data`

        - Indicates the start of the data values section. (Typically, the beginning of the program)
    - `.text`
        - Indicates the start of the program instruction section.
- Withhin the instruction section are program labels and branch addresses.
    - `main:`
        - The initial line to run when executing the program
    - Other labels are determined by the function names used in one's program.

Labeling data values

- Data storage
    - At beginning of program, create labels for memory locations that are used to store values.
    - Always in form `label .type value(s)`
    - Create a single integer variable with initial value 3

```
var1:       .word       3
```

    - Create a 2-element character array with elements initialized to a and b

```
array1:     .byte       'a', 'b'
```

    - Allocate 40 consecutive bytes, with uninitialized storage. Could be used as a 40 element character array, or a 10 element integer array

```
array2:     .space      40
```

Pseudo-Instructions

- Pseudo-instructions are there for the convenience of the programmer.
- The assembler translates them into 1 ot more *real* MIPS assembly instructions
    - **Real MIPS instructions have opcodes, pesudo-instructions do not!**
    - The assembler often uses the special `$at` register (also written as `$t`) when mapping pseudo-instructions to MIPS instructions.

Example: The `la` instruction

- `la` (load address) is a pseudo-instruction written in the format
    - `la $d, label`
    - Loads a register `$d` with memory address that `label` corresponds to.

- Usually translated by the assembler into the following two MIPS instructions
    1. `lui $at, immediate`, load upper immediate
        - The immediate represents the upper 16 bits of the memory address label corresponds to. These bits are loaded in the upper 16 bits of the destination register. Lowest 16 bits are set to 0.
        - Register `$at($1)` is the register used by the assembler
    2. `ori $d, $at, immediate2`
        - `immediate2` represents the lower 16 bits of the memory address label corresponds to. (Here `ori` is the OR immediate operation.)

Example: `bge` branching

- Some branch instructions are pseudo instructions, for example
    - `bge $s, $t, label`
        - Branch to label if and only if `$s >= $t`
            - (Comparing regiser contents)
    - Implemented by using one of comparison instructions followed by `beq` or `bne`. One plausible implementation is

      ```
      slt $at, $s, $t         # set $at to 1 if $s<$t
      beq $at, $zero, $label  # branch if $at == 0
      ```

      Notice that here we have usde the `$at` register. This is legal because we are now doing the role of assembler and `$at` is reserved for the assembler.

Arrays and Structs

Arrays

- Arrays in assembly languarge:
    - Arrays are stored in consecutive locations in memory.
        - The address of the array is the address of the array's first element
        - To access element i of an aray, use i to calculate an offset distance. Add that offset to the address of the first element to get the address of the i-th element.
            - `offset = i * sizeof(single element)`
    - To operate on array elements, load the array values into registers. Operate on them, then store them back into memory.

Translating Arrays

Consider the code snippet

```
int A[100], B[100];     # B is an array of 100 slots, each of value 42
for (i = 0; i < 100; i++) {
    A[i] = B[i] + 1
}
```

Then, in assembly

```
        .data
A:      .space 400
B:      .word  42:100

        .text
main:   la $t9, A               # Load From Memory
        la $t9, B               # Load From Memory
        add $t0, $zero, $zero   # $t0 <- i, Init to 0
        addi $t1, $zero, 100    # $t1 <- 100, stop condition

LOOP:   bge $t0, $t1, END
        sll $t2, $t0, 2         # sizeof(int) = 4 byte, $t2 = offset
        add $t3, $t8, $t2       # $t3 = (*)A + offset
        add $t4, $t9, $t2       # $t4 = (*)B + offset
                                # Note: In memory, above two are different
addresses
        lw $t5, 0($t4)          # $t4 = addr(B[i]), $t5 = B[i]
        addi $t5, $t5, 1        # Increment by one on loaded value
        sw $t5, 0($t3)          # Store the value into appropraiate A
position

UPDATE: addi $t0, $t0, 1        # i++
        j LOOP                  # Jump Back

END:    ...
```

In above implementation, we used a seperate loop variable to count the number of loops. It is also possible to do this with just the offset value where we constantly compare to 400, the final offset in the last iteration corresponding to the last element in both arrays.

Example: Struct Program

```
        .data
a1:     .space 12

        .text
main:   addi $t0, $zero, a1     # store array head a1 in $t0
```

```
        addi $t1, $zero, 5      # initialize $t1 to 5, prep
        sw $t1, 0($t0)          # write to position 0($t0), value of $t1
        addi $t1, $zero, 13     # ... (repeat above)
        sw $t1, 4($t0)          # second slot, off-set 4 bytes (32 bits)
        addi $t1, $zero, -7
        sw $t1, 8($t0)          # ... (repeat above)
```

- How can we figure out the purpose of this code?
- The `sw` lines indicate that values in `$t1` are being stored at `$t0`, `$t0 + 4`, and `$t0 + 8`.
    - Each previous line sets the value of `$t1` to store
- Therefore, this code stores the value 5, 13, and -7 into the struct at location `a1`.

Designing Assembly Code

Making sense of assembly code

- Assembly language looks intimidating because the programs involve a lot of code.
    - No worse than your CSC108 assignments would look to the untrained eye!
- The key to reading and designing assembly code is recognizing portions of code that represent higher-level operations that you're familiar with.

Functions in Assembly - Introducing Stacks

Function Calls and Returns

- To suppirt functions, we need to be able to:
    - Define the start of a function
        - Label the first line to provide a target address to jump to.
    - Take in function arguments and return values
        - Could use registers but ***might need another solution***
    - Store variables local to the fnction and also ensure functions don't clobber useful data on registers
        - Registers for storing data, ***but are any off-limit?***
    - Return to the calling site
        - after the last line in the function, return to the instruction after the one that dod the function call.

How do we call a function?

- `jal FUNCTION_LABEL`
    - This jumps to the first line of the function, which has the specified label (i.e. `function_X` here)
- `jal` is a J-type instruction
    - It updates register `$31` (`$ra`, return address register) and also the program counter
    - After it's executed, `$ra` contains the address of the instruction ***after*** the line that called `jal`.

How do we return from a function?

- `jr $ra`
    - The PC is stet to the address in `$ra`
- But how do we knwo what is in `$ra` ?
    - `$ra` was set by the most recent `jal` instruction (function call)!
- **Problem: nested function calls are not able to run using this simple scheme!**

Solving the `$ra` dilemma

- How do we make sure tat we don't clobber the value of `$ra` every time we call a nested function?
- Need to put `$ra` away somewhere for safe keeping if we know we are about to overwrite it.
- Would be ideal if the sturcture used for this mode it it easy to find the last item that was stored away. **STACK!!!**

The Stack

The Stack and the Stack Pointer

- The statck is a spot in memory used to store values independent of the registers (which can get overwritten easily)
- A special register stores the **stack pointer**, which points to the last element pushed onto the top of the stack.
    - For MIPS the stack pointer is `$sp($29)`. This holds the address of the last element pushed to the top of the stack
    - In other systems, `$sp` could point to the first empty location on top of the stack.
- We can push data, like `$ra`, onto the stack (which makes it ***grow***) and pop data from the stack (which makes it *shrink*).
- The stack is allocated a maximum space in memory. If it grows too large, there is the risk of it exceeding this predefined size and/or overlapping with heap (Whoops, bad memory from 209 : / Stack frames are allocated with 6MB each, and huge recursions can potetially overflow in this.)

The programmer's view of memory

- The stack is a part of memory used for function calls etc.
- The stack grows towards samller (lower) addresses
- The stack uses LIFO (last in first out order)

```
--------------------------
0x00000000 |  Reserved   |
...        |             |
--------------------------
           |  Code(.text)|
...        |             |
--------------------------
           |   (.data)   |  <- Global Variables
...        |             |
-------------------------
           |     Heap     |  (Heap grows downward)
...        |              |
--------------------------
...        |              |
           | Unallocated |
...        |              |
--------------------------
           |         (top)|  (<- $sp points to current top)
           |    Stack     |  (Stack grows upward, to small address)
0x7FFFFFFF |         (bot)|
--------------------------
0xFFFFFFFF |   OS Code    |
--------------------------
```

The stack to the rescue!

- When do you store values onto the stack?
  - Whenever you call a function and want to preserve values from getting overwritten (like `$ra`)
- What happens when you have nested function calls, each of which stores `$ra` on the stack?
  - Different `$ra` value will exists in layers on the stack over time.
- We can also use the stack to store
  - Function Arguments
  - Function return values

Popping Values off the stack

```
# pop a word off the stack
lw $ra, 0($sp)
# move stack pointer a word (4 bytes)
addi $sp, $sp, 4
```

- **Note:** `addi $sp, $sp, 4` is adding 4 to the stack pointer, hence moving downward by a word length

Pushing Values to the stack

```
# move stack pointer a word (new location at top)
addi $sp, $sp, -4
# push a word onto the stack
sw $ra, 0($sp)
```

Advice on using the stack

- Any space you allocate on the stack, you should later deallocate. (Similar to in C, `malloc()` on heap and de-allocate using `free()`)
- If you push items in a certain order, you should pop the items in the reverse order.
    - might help to draw out an image of how the stack looks like
- When pushing more than one item onto the stack, you can:
    - Either allocate all the spaces in the beginning or allocate spaces as you go. Same principle applies for popping

Passing Function Parameters

Functions versus Code

- Since functions have entry and exit points, they also need input and output parameters.
    - In other languages, these parameters are assumed to be available at the start of the function.
    - In Assembly, you have to fetch those values from memory first before you can operate on them.

Common Calling Conventions

- While most programs pass parameters through the stack, it is also possible to use registers to pass values to and from programs:
    - Registers 2 - 3 (`$v0, $v1`): return values
    - Registers 4 - 7 (`$a0 - $a3`): function arguments
- If your function has up to 4 arguments you would use the `$a0` to `$a3` registers in that order. Any additional arguments would be pushed on the stack.
    - More common convention is to just push all arguments to the stack. (Specific way of doing this will be provided on he final exam.)

Example: String function `strcpy` program

```
void strcpy (char* x, char* y) {
    int i = 0;
    while ((x[i] = y[i]) != '\0')
        i += 1;
    return 1;
}
```

We will convert the above program into assembly, using parameters from the stack. In this case, the parameters `x` and `y` are passed into the function, in that order.

```
!#! Each char is 1 byte

strcpy:     lw      $a0, 0($sp)         # load *y* into $a0
```

```
            addi    $sp, $sp, 4         # move to next word
            lw      $a1, 0($sp)         # load *x* into $a1
            addi    $sp, $sp, 4         # move new stack top
            add     $t0, $zero, $zero   # $t0 = i, init to 0

L1:         add     $t1, $t0, $a0       # $t1 = y + i = ystart + offset
            lb      $t2, 0($t1)         # $t2 = current y[i]
            add     $t3, $t0, $a1       # #t3 = x + i = xstart + offset
            sb      $t2, 0($t3)         # x[i] = $t2 (x[i] = y[i])
            beq     $t2, $zero, L2      # y[i] ?= '\0'
            addi    $t0, $t0, 1         # i++
            j       L1                  # loop again

L2:         addi    $sp, $sp, -4        # Move $sp to 1 above top
            addi    $t0, $zero, 1       # $t0 = 1
            sw      $t0, 0($sp)         # Write $t0 to stack top
            jr      $ra                 # return
```

Function Considerations

Maintaining Register Values

- We have already demonstrated why we'd need to push `$ra` onto the stack when having nested function calls.
- What about other registers?
  - How do we know that a fnction we called didn't overwrite registers that we were using?
    - **Remember there is one and only one register file!**
- **Solution: caller and callee calling conventions**

Calling Conventions

- Caller vs. Callee
  - Caller is the function calling another function.
  - Callee is the function being called.
  - A function canbe both a caller and a callee
- We seperate registers into
  - Caller-Saved Registers (`$t0 - $t9`)
  - Callee-Saved Registers (`$s0 - $s7`)

Register Saving Conventions

- Caller-Saved Registers
  - Regisers `8-15,24,25 ($t0 - $t9)`: **temporaries**
  - Registers that the caller should save to the stack before calling a function. If they don't save them, there is no gurantee that the contents of these registers will bot be clobbered.

  - Push them to the stack just before you call another function and restore them immediately after.
- Calle-Saved Registers
  - Registers `16-23 ($s0 - $s7)`: **saved temporaries**
  - It is the responsiblility of te callee to save these registers and later restore them, if it is going to modify them.
  - Push them to the stack first thing in your function body and restore them just before you return. (preserving the values in the registers using the stack)

Stack & Function Summary

- Before Calling a subroutine
  - Push registers onto the stack to preserve their values
  - Push the input paraeters onto the stack
- At the start of the subroutine
  - Pop the input parameters from the stack
- At the end of subroutine
  - Push the return values onto the stack
- Coming back from a subroutine call:
  - Pop the return values from the stack
  - Pop the saved register values and restore them

Recursion in Assembly

Recursive Programs

```
int fact (int x) {
    if (x == 0)
        return 1;
    else
        return x * fact(x - 1)
}
```

- Still needs base case and recursive step, as with other languages.
- Main difference (w.r.t. other language): **Maintainning register values**
  - When a recursive function calls itself in assembly, it calls `jal` back to the beginning of the program
  - What happens to the prevous value `$ra`?
  - What happens to the previous register values, when the program runs a second time?
- **Solution: the stack!**
  - Before recursive call, store the register values that you use onto the stack, and restore them when you come back to that point.
  - Don't forget to store `$ra` as one of those values, or else the program will loop forever.

Translated `fact(int x)` in Assembly

```
main:            addi $t0, $zero, 10     # Assume call on fact(10)
                 addi $sp, $sp, -4       # Push 10 onto stack
                 sw   $t0, 0($sp)        # puch 10 onto stack
                 jal  factorial          # result will be on stack
                 ...

factorial:       lw   $t0, 0($sp)        # get x from stack
                 bne  $t0, $zero, rec    # base case ?

base:            addi $t1, $zero, 1      # Base case reached, make 1
                 sw   $t1, 0($sp)        # put 1
                 jr   $ra                # return to caller

rec:             addi $t1, $t0, -1       # $t1 <- x - 1
                 addi $sp, $sp, -4       # put return value
                 sw   $ra, 0($sp)        #  onto the stack
                 addi $sp, $sp, -4       # put x - 1 onto
                 sw   $t1, 0($sp)        #   stack
                 jal  factorial          # RECURSIVE CALL
                 lw   $t2, 0($sp)        # pop return value
                 addi $sp, $sp, 4        #   from stack
                 lw   $ra, 0($sp)        # restore return
                 addi $sp, $sp, 4        #   address value
                 lw   $t0, 0($sp)        # restore x value
                 addi $sp, $sp, 4        #   for this call
                 mult $t0, $t2           # x * fact (x - 1)
                 mflo $v0                # fetch product
                 addi $sp, $sp, -4       # push n! result
                 sw   $v0, 0($sp)        #   onto stack
                 jr   $ra                # return to caller
```