CSC258 Notes – Assembly Language

Jenci Wei

Winter 2022

Contents

1 Intro

Assembly Language $\xrightarrow{\text{converted to}}$ Machine Code $\xrightarrow{\text{extract first 6 bits}}$ Opcode $\rightarrow$

$\xrightarrow{\text{input to}}$ Control Unit $\xrightarrow{\text{produces}}$ Datapath Signals

Intro to Machine Code

- Within a processor, operations are performed by
  1. The instruction register
     - Sending instruction components to the processor
  2. The control unit
     - Based on the *opcode* value (sent from the instruction register), sending a sequence of signals to the rest of the processor
- Example: `C = A + B`
  - Assume that A is stored in `$t1`, B in `$t2`, C in `$t3`
    * Dollar sign indicates that we are referring to a register
  - Assembly language instruction:

```
add $t3, $t1, $t2
```

    * Destination comes before what we want to add
  - Machine code instruction:

```
000000 01001 01010 01011 XXXXX 100000
```

Encoding the Instruction

- Machine code instructions contain all the details about a processor operation, such as
  1. What operation is being performed (opcode)
  2. What registers are being used in this operation
  3. What other info might be needed to make this operation happen (immediate or shift values)

R-Type Instructions

- E.g. `add $t3, $t1, $t2`
- First 6 bits is 000000, indicates that this is an R-type operation
- Last 6 bits is 100000, which the ALU understands as "add"
- After the opcode, the next group of three 5 bits specify the three registers that are involved
  - The first source comes first, i.e. 01001
  - The second source comes next, i.e. 01010
  - The destination comes last, i.e. 01011
- After the register bits, the next 5 bits indicate whether we are doing a shift
  - Since we are not doing a shift, they don't matter

- Overall, we have `000000 01001 01010 01011 XXXXX 100000`

Operating on Registers

- Any operations whose inputs and outputs are all registers are called **R-type**
- In order to encode R-type instructions, we need to know the 5-bit codes used to refer to the input and output registers

Machine Code and Registers

- MIPS is **register-to-register**, i.e. almost all operations rely on register data
- MIPS provides 32 registers
  - Some have special values
    - `$0 ($zero)`: always have value 0
    - `$1 ($at)`: reserved for the assembler
    - `$28-$31 ($gp, $sp, $fp, $ra)`: memory and function support
      - `$ra` is the return address. When a function returns, it returns to this address
      - `$sp` is the stack pointer, storing where the stack is
    - `$26-$27`: reserved for OS kernel
  - Some are used ny programs as function parameters
    - `$2-$3 ($v0, $v1)`: return values
    - `$4-$7 ($a0-$a3)`: function arguments
  - Some are used by programs to store values
    - `$8-$15, $24-$25 ($t0-$t9)`: temporaries
    - `$16-$23 ($s0-$s7)`: saved temporaries
  - Three special registers `PC, HI, LO` that are not directly accessible, and not part of the 32
    - `PC` is the program counter, that stores the location of the current instruction
    - `HI` and `LO` are used in multiplication and division, and have special instructions for accessing them

I-Type Instructions

- Operates on registers, but involves a constant value as well
- "Immediate" value
- The constant is encoded in the last 16 bits of the instruction
- E.g.

```
        addi $t2, $t1, 42
```

  - Machine code instruction:

```
            001000 01001 01010 0000000000101010
```

  - Opcode is 001000, which stands for immediate add operation
  - Source register is 01001
  - Destination register is 01010

- Intermediate value is 0000000000101010

J-Type Instructions

- Jump to a location in memory encoded by the last 26 bits of the instruction
- Location is stored as a label, which is resolved when the assembly program is compiled
- E.g.

```
        j main
```

  - Machine code instruction:

```
            000010 00000000000000011000101010
```

MIPS Instruction Tupes

- R-type

opcode	rs	rt	rd	shamt	funct
6	5	5	5	5	6

- I-type

opcode	rs	rt	immediate
6	5	5	16

- J-type

opcode	address
6	26

Machine Code Details

- R-type instructions have an opcode of 000000, with a 6-bit function listed at the end
- For the "don't care" bits that we indicate as `X`, the assembly language interpreter always assign them to some value
- We could program the processor with either the machine code or an equivalent language

Assembly Language

- Lowest-level language that a human would ever program in
- Many compilers translate their high-level program commands into assembly commands, which are then converted into machine code and used by the processor
- There are multiple types of assembly language, especially for different architectures

MIPS

- Microprocessor without Interlocked Pipeline Stages
  - A type of RISC (Reduced Instruction Set Computer) architecture
- Provides a set of simple and fast instructions
  - Compiler translates instructions into 32-bit instructions for instruction memory
  - Complex instructions (e.g. multiplication) are built ouut of simple ones by the compiler and assembler

MIPS Instructions

- Instructions are written in the format of `<instr> <parameters>`
- Each instruction is written on its own line
- All instructions are 32 bits (4 bytes) long
- Instruction addresses are measured in bytes, starting from the instruction at address 0
  - All instruction addresses are divisible by 4

Frequency of Instructions

Instruction Type	Examples	Usage	Integer Frequency	Floating Point Frequency
Arithmetic	`add, sub, addi`	Operations in assignment statements	16%	48%
Data Transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	References to data structures, such as arrays	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	Operations in assignment statements	12%	4%
Conditional Branch	`beq, bne, slt, slti, sltiu`	If statements and loops	34%	8%
Jump	`j, jr, jal`	Procedure calls, returns, and case/switch statements	2%	0%

2 ALU Operations

Arithmetic Instructions

Instruction	Opcode/Function	Syntax	Operation
`add`	100000	`$d, $s, $t`	`$d = $s + $t`
`addu`	100001	`$d, $s, $t`	`$d = $s + $t`
`addi`	001000	`$t, $s, i`	`$t = $s + SE(i)`
`addiu`	001001	`$t, $s, i`	`$t = $s + SE(i)`
`div`	011010	`$s, $t`	`lo = $s / $t; hi = $s % $t`
`divu`	011011	`$s, $t`	`lo = $s / $t; hi = $s % $t`
`mult`	011000	`$s, $t`	`hi:lo = $s * $t`
`multu`	011001	`$s, $t`	`hi:lo = $s * $t`
`sub`	100010	`$d, $s, $t`	`$d = $s - $t`
`subu`	100011	`$d, $s, $t`	`$d = $s - $t`

- `hi` and `lo` refer to the high and low bits
- `SE` is sign extend (since the ALU only takes in 32-bit numbers)
- `u` is the unsigned counterpart, overflow is handled differently
- Multiplication results in 64 bits, therefore we need two registers to store it
  - `hi` stores the first 32 bits and `lo` stores the last 32 bits
- Division results in two numbers: the quotient and the remainder
  - `lo` stores the quotient and `hi` stores the remainder

Logical Instructions

<table>
<tr><th>Instruction</th><th>Opcode/Function</th><th>Syntax</th><th>Operation</th></tr>
<tr><td><code>and</code></td><td>100100</td><td><code>$d, $s, $t</code></td><td><code>$d = $s & $t</code></td></tr>
<tr><td><code>andi</code></td><td>001100</td><td><code>$t, $s, i</code></td><td><code>$t = $s & ZE(i)</code></td></tr>
<tr><td><code>nor</code></td><td>100111</td><td><code>$d, $s, $t</code></td><td><code>$d = ~($s | $t)</code></td></tr>
<tr><td><code>or</code></td><td>100101</td><td><code>$d, $s, $t</code></td><td><code>$d = $s | $t</code></td></tr>
<tr><td><code>ori</code></td><td>001101</td><td><code>$t, $s, i</code></td><td><code>$t = $s | ZE(i)</code></td></tr>
<tr><td><code>xor</code></td><td>100110</td><td><code>$d, $s, $t</code></td><td><code>$d = $s ^$t</code></td></tr>
<tr><td><code>xori</code></td><td>001110</td><td><code>$t, $s, i</code></td><td><code>$t = $s ^ZE(i)</code></td></tr>
</table>

- `ZE` is zero extend (i.e. pad upper bits with 0 value)
- `&`: bitwise AND
- `|`: bitwise OR
- `^`: bitwise XOR

Shift Instructions

Instruction	Opcode/Function	Syntax	Operation
`sll`	000000	`$d, $t, a`	`$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 = $t >>> a`
`srlv`	000110	`$d, $t, $s`	`$d = $t >>> $s`

- `sr`: shift right, `sl`: shift left

- `l`: logical, `a`: arithmetic
- `v` denotes a variable number of bits, specified by `$s`
- `a` is the *shift amount*, and is stored in `shamt` when encoding the R-type machine code instructions

Data Movement Instructions

Instruction	Opcode/Function	Syntax	Operation
`mfhi`	010000	`$d`	`$d = hi`
`mflo`	010010	`$d`	`$d = lo`
`mthi`	010001	`$s`	`hi = $s`
`mtlo`	010011	`$s`	`lo = $s`

- `mf`: move from, `mt`: move to

3 Jump Instructions

Control Flow

- We have **labels** on the left side that indicate the points that the program flow might need to jump to
- References to these points are resolved at compile time

Jump Instructions

Instruction	Opcode/Function	Syntax	Operation
`j`	000010	label	`pc = (pc & 0xF0000000)|(i << 2)`
`jal`	000011	label	`$31 = pc + 4; pc = (pc & 0xF0000000)|(i << 2)`
`jalr`	001001	`$s`	`$31 = pc + 4, pc = $s`
`jr`	001000	`$s`	`pc = $s`

- `jal`: jump and link
  - Register `$31` (i.e. `$ra`) stores the address that's used when returning from a subroutime (i.e. the next instruction to run)
  - Used when jumping into a function
  - After the function returns, use `jr` to go back
- `jr` and `jalr` are jumps, but *not* J-type instructions
  - Only registers are involved, no constants are used
- `j` and `jal` are J-type instructions, thus only have 26 bits to hold an address
  - Solution 1: Trailing zeros
    * Since jump instructions load new addresses into the program counter, the values being loaded must be divisible by 4
    * The binary values of these addresses will always end in 00, and we could ignore those two zeros
    * This allows us to store an 28-bit address in 26-bits
  - Solution 2: Leading bits
    * MIPS keeps the first 4 bits of the previous program counter value
    * Those 4 bits would be the most significant 4 bits
    * We would have a full 32-bit address that could be obtained with formula

```
pc = (pc & 0xF0000000) | (i << 2)
```

Usages of Jump Instructions

- `j` is used for for-loops and while-loops
- `jal` is used when we want to call a function, and then `jr` is used when the function returns to its caller
- Cannot have nested functions since there is only one register to store the return address
  - Need the stack in order to achieve this functionality

4 Branch Instructions

Branch Instructions

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) pc += i << 2`

- Use d to produce if-statement behaviour
- I-type instruction

Branch's Immediate Value

- The *immediate value* is a 16-bit *offset* to add to the current instruction if the branch condition is satisfied, i.e. how much further in memory should the program counter go from the current location
  - Calculated as the difference between the current PC value and the address of the instruction we are branching to
  - Stored as the number of instructions (*not* the number of bytes) in order to save space (no trailing zeros)
  - i value is positive when jumping forward, and negative when jumping backward

Calculating the i-value

- Depends on the implementation (of whether the PC is incremented before or after the branch offset calculation)
- If the PC is incremented first:

```
        i = (label location - (current PC)) >> 2
```

- If the branch offset is calculated first:

```
        i = (label location - (current PC + 4)) >> 2
```

- We assume the former for this course, since MARS uses the former

Conditional Branch Terminology

- The branch is **taken** when the branch condition is met
- Otherwise the branch is **not taken**

Comparison Instructions

Instruction	Opcode/Function	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))`

- Stores a 1 in the destination register if the condition is true, and stores a 0 otherwise
- Often used in combination with branch instructions

If-Statement Implementation

- E.g. we want to implement the following code

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

- Assembly code:

```
main:  bne $t1, $t2, END # branch if (i != j)
       addi $t1, $t1, 1  # i++
END:   add $t2, $t2, $t1 # j = j + 1
```

If-Else Statement Implementation

- E.g. we want to implement the following code

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

- Approach: branch on the if condition first

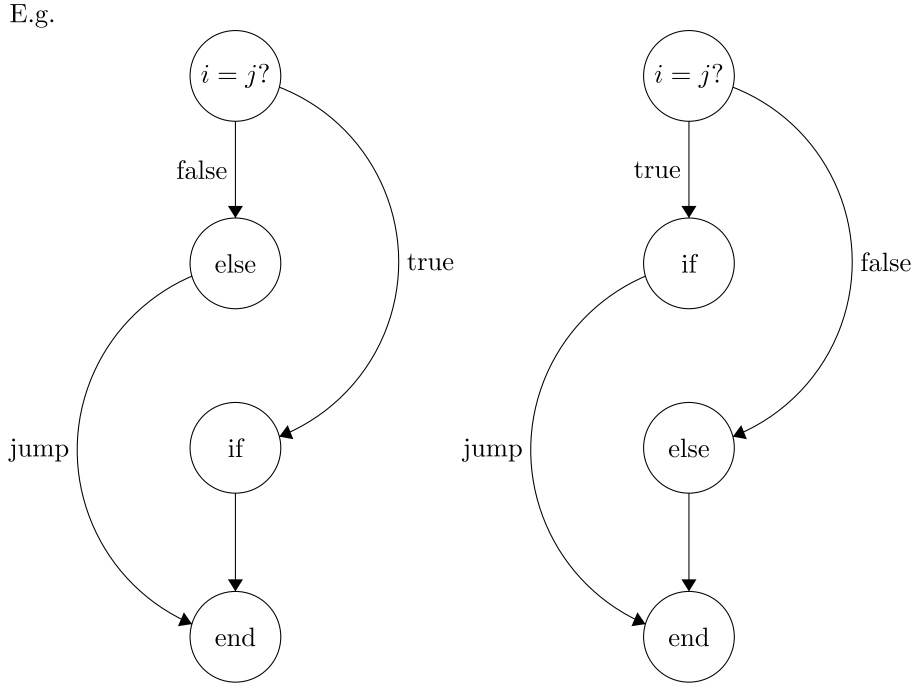
```
main:  beq $t1, $t2, IF  # branch if (i == j)
       addi $t1, $t1, -1 # i--
       j END             # jump over IF
IF:    addi $t1, $t1, 1  # i++
END:   add $t2, $t2, $t1 # j += i
```

- Approach: branch on the else condition first

```
main:  bne $t1, $t2, ELSE # branch if !(i == j)
       addi $t1, $t1, 1  # i++
       j END             # jump over ELSE
ELSE:  addi $t1, $t1, -1 # i--
END:   add $t2, $t2, $t1 # j += i
```

If Statement Flow Chart

- E.g.



- Left: branch on the if condition first
- Right: branch on the else condition first

Multiple `if` Conditions

- Need a branch statement for *each* condition
- Handling `or`
  - Branch to `if` when first condition is true
  - Branch to `if` when second condition is true
  - Execute else block
- Handling `and`
  - Branch to `else` when first condition is false
  - Branch to `else` when second condition is false
  - Execute if block

While Loops

- Use jump to make a loop
- Use branch to ensure that it is not an infinite loop
- Same structure as C

For Loops

- C structure

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

- Assembly structure

```
main:      <init>
START:     if (!<cond>) branch to END
           <for-body>
UPDATE:    <update>
           jump to START
END:
```

- Has an initialization and an update section in addition to a while loop

5 Memory Operations

Interacting With Memory

- All programs 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

```
          op $t, i($s)
```

  - `op`: operation (load or store)
  - `$t`: local data register
  - `$s`: register storing address of data value in memory
  - `i`: offset from memory address, e.g. go forward i bytes from `$s`

Loads and Stores

- **Loads** are read operations
  - We *load* (i.e. read) 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

Memory Instructions

- First letter: `l` or `s`
  - `l`: load
  - `s`: store
- Second letter: `w`, `h`, or `b`
  - `w`: word (32 bits)
  - `h`: half-word (16 bits)
  - `b`: byte (4 bits)
- Optional third letter: `u`, which stands for "unsigned"

Load and Store Instructions

Instruction	Opcode/Function	Syntax	Operation
`lb`	100000	`$t, i($s)`	`$t = SE(MEM[$s + i]:1)`
`lbu`	100100	`$t, i($s)`	`$t = ZE(MEM[$s + i]:1)`
`lh`	100001	`$t, i($s)`	`$t = SE(MEM[$s + i]:2)`
`lhu`	100101	`$t, i($s)`	`$t = ZE(MEM[$s + i]:2)`
`lw`	100011	`$t, i($s)`	`$t = MEM[$s + i]:4`
`sb`	101000	`$t, i($s)`	`MEM[$s + i]:1 = LB ($t)`
`sh`	101001	`$t, i($s)`	`MEM[$s + i]:2 = LH ($t)`
`sw`	101011	`$t, i($s)`	`MEM[$s + i]:4 = $t`

- `b`: byte
- `h`: half word
- `w`: word

- `SE`: sign extend
- `ZE`: zero extend
- `LB`: lowest byte
- `LH`: lowest half word

Alignment Requirements

- Misaligned memory accesses result in errors
- Word accesses (i.e. addresses specified in `lw` or `sw` instruction) should be *word-aligned* (i.e. divisible by 4)
- Half-word accesses should only involve half-word aligned addresses (i.e. even addresses)
- No constraints for byte accesses
- Fetching words and half-words from invalid addresses will cause the processor to raise an address error exception
- E.g. addresses stored in the PC need to be divisible by 4

Little Endian and Big Endian

- Goal: assemble multiple bytes into a larger data type
- E.g. we have the following bytes

Address	Byte
X	Byte A
$X+1$	Byte B
$X+2$	Byte C
$X+3$	Byte D

- We have two ways of ordering them into the register
  1. Byte A Byte B Byte C Byte D
     - Same order as address of bytes
     - Called **Big Endian**
     - The *most significant byte* of the word is stored first
  2. Byte D Byte C Byte B Byte A
     - The former bytes are usually less significant than the latter bytes
     - Called **Little Endian**
     - The *least significant byte* of the word is stored first

MIPS Endianness

- MIPS processors are bi-endian, i.e. they can operate with either endian byte order
- MARS simulator uses the same endianness as the machine it is running on
- E.g. x86 CPUs are little endian

Reading from Devices

- Memory is used to communicate with outside devices, 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 OS, e.g. interacting with the user, exiting the program
- Instruction: `trap`

Memory Segments and Syntax

- Programs are divided into two main sections in memory
    1. `.data`: indicates the start of the data values section (typically at the beginning of program)
    2. `.text`: indicates the start of the program instruction section
- Within 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 the program

Labelling Data Values

- Data storage
    - At the beginning of the program, create labels for memory locations that are used to store values
    - In the form

```
            label   .type   value(s)
```

    - If we want to allocate space, then use `.space` followed by the number of bytes needed

Arrays

- 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 on an array, use i to calculate an offset distance, i.e.

```
            offset = i * sizeof(element)
```

- To operate on array elements, fetch the array values and store them in registers, operate on them, then store them back into memory

6 Functions

Functions

- A **function** creates an interface to a piece of code by defining an entry and exit point to it, alongside with the input and output parameters
- Brings forth two major considerations
  1. Jumping into and out of a function
  2. Passing values to and from a function

Defining a Function

1. Define the start of a function
   - Label the first line to provide a target address to jump to
2. Take in function arguments and return values
   - Could use registers
3. Store variables local to the function
   - Ensure that function don't clobber useful data on registers
4. Return to the calling site
   - After the last line in the function, return to the instruction after the one that did the function call

Calling and Returning From a Function

- `jal FUNCTION_LABEL` jumps to the first line of the function, which has the specified label
  - J-Type instruction taht updates register `$31 ($ra)` (i.e. the 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`
- `jr $ra` sets the PC to the address in `$ra`
  - Analogous to the return statement
  - `$ra` is set by the most recent `jal` instruction (i.e. function call)

Nested Function Call Issue

- When the nested function call modifies `$ra`, the return address of the original function call is overwritten
- Need to put `$ra` away somewhere for safe keeping if we are about to overwrite it

7 Stack Operations

The Stack and the Stack Pointer

- The **stack** is a spot in memory used to store function values independent of the registers
- A special register stores the **stack pointer**, which points the *last element* pushed onto the top of the stack
  - For MIPS the stack pointer is `$29 ($sp)`
  - In other systems `$sp` could point to the *first empty location* on top of the stack
- We can push data (e.g. `$ra`) onto the stack, and pop data from the stack
- The stack is allocated a maximum space in memory – if it grows too large, then it could exceed its predefined size and/or overlapping with the heap
- Memory diagram

Reserved
Code (.text)
Global Variables (.data)
Heap
Unallocated
Stack
OS Code

- If stack gets to large, then there would be a *stack overflow* error
- The stack grows towards *smaller* (lower) addresses, i.e. starts near OS code and grows upward in the diagram
- Stack uses *LIFO (last-in-first-out)* order

Using the Stack

- Whenever we call a function and want to preserve values from getting overwritten (e.g. `$ra`), we store values onto the stack
- When we have nested function calls, different `$ra` values would exist in layers on the stack over time
- The stack can also be used to store
  - Function arguments
  - Function return values
  - Etc.
- Popping values off the stack

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

  - Do a load (or multiple loads as needed)
  - Deallocate space by *incrementing* the stack pointer by the appropriate number of bytes
- Pushing values to the stack

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

  - Allocate space by *decrementing* the stack pointer by the appropriate number of bytes
  - Do a store

- Any space allocated on the stack should be deallocated later on
- If we push items in a certain order, we should pop the items in the reverse order
- When pushing more than 1 item onto the stack, we can either allocate all the space in the beginning or allocate space as we go
  - Same applies for popping

Function Calling Conventions

- When we use registers to pass values to and from programs:
  - Registers 2-3 (`$v0- $v1`) are used as return values
  - Registers 4-7 (`$a0-$a3`) are used as function arguments
- If the function has up to 4 arguments, use `$a0-$a3` in that order; any additional arguments would be pushed on the stack
- Most common convention is to push *all arguments* to the stack

Function Example: `strcpy`

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

1. Initialization
   - Values that we need to store:
     - Address of `x[0]` and `y[0]`
     - Current offset value, i.e. i
     - Temporary values for the address of `x[i]` and `y[i]`
     - Current value being copied from `y[i]` to `x[i]`
   - We can fetch the locations of `x[0]` and `y[0]` from the stack
2. Main algorithm
   - We need to perform the following steps:
     - Get the location of `x[i]` and `y[i]`
     - Fetch a character from `y[i]` and store it in `x[i]`
     - Jump to the end if the character is the null character
     - Otherwise, increment i and jump to the beginning

  - At the end, push the value 1 onto the stack and return to the calling program

3. Implementation

```
strcpy:   lw    $a0, 0($sp)        # pop x address off the stack
          addi  $sp, $sp, 4
          lw    $a1, 0($sp)        # pop y address off the stack
          addi  $sp, $sp, 4
          add   $t0, $zero, $zero  # $t0 = offset i
L1:       add   $t1, $t0, $a0      # $t1 = x + i
          lb    $t2, 0($t1)        # $t2 = x[i]
          add   $t3, $t0, $a1      # $t3 = y + i
          sb    $t2, 0($t3)        # y[i] = $t2
          beq   $t2, $zero, L2     # y[i] = '\0'?
          addi  $t0, $t0, 1        # i++
          j     L1                 # loop
L2:       addi  $sp, $sp, -4       # push 1 onto the top of the stack
          addi  $t0, $zero, 1
          sw    $t0, 0($sp)
          jr    $ra                # return
```

Calling Conventions

- Besides `$ra`, other registers might be overwritten by a nested function
- Caller vs. callee
  - **Caller** is the function calling another function
  - **Callee** is the function being called
- By convention, the caller and the callee use different sets of registers
  - *Caller-saved registers*: `$t0-$t9`, i.e. temporaries
    - The caller should save those registers to the stack before calling a function
    - If the caller does not save them, there is no guarantee the contents of these registers will not be clobbered
    - Push them to the stack before calling another function and restore them immediately after
  - *Callee-saved registers*: `$s0-$s7`, i.e. saved temporaries
    - It is the responsibility of the callee to save these registers and later restrore them, if it's going to modify them
    - Push them to the stack in the first line of the function body, then restore them before returning
- The caller can assume that the `s`-registers will be untouched when control is passed back from the callee
  - Cannot assume the same for `t`-registers
- A function can be both a caller and a callee (i.e. recursion)

Stack and Function Summary

1. Before calling a subroutine (i.e. helper function):
   - Push registers onto the stack to preserve their values
   - Push the input parameters onto the stack

2. At the start of a subroutine:
   - Pop the input parameters from the stack
3. At the end of a subroutine:
   - Push the return values onto the stack
4. Coming back from a subroutine call:
   - Pop the return values from the stack
   - Pop the saved register values and restore them

8 Recursion

Handling Recursive Programs

- Need base case and recursive step
- Main difference from other languages: maintaining register values
  - When a recursive function calls itself in assembly, it calls `jal` back to the beginning of the program
  - Previous value of `$ra` is overwritten
  - Previous register values are overwritten
- Use the stack for recursive programs
  - Before recursive call, store the register values that we use onto the stack, and restore them when we come back to that point
  - Must store `$ra` as one of these values

Example: Factorial

- High-level code:

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

- Assembly pseudocode
  - Pop `n` off the stack
    - Store in `$t0`
  - If `$t0 == 0`:
    1. Push 1 onto stack
    2. Return to calling program
  - If `$t0 != 0`:
    1. Calculate $n - 1$
    2. Store `$t0` and `$ra` onto stack
    3. Push $n - 1$ onto stack
    4. Call factorial
    5. ...
    6. Pop the result of factorial$(n - 1)$ off the stack, store it in `$t2`
    7. Restore `$ra` and `$t0` from stack
    8. Multiply factorial$(n - 1)$ and n
    9. Push result onto stack
    10. Return to the calling program

9 Pseudo-Instructions

Pseudo-Instructions

- Pseudo-instructions are there for the convenience of the programmer
- The assembler translates them into 1 or more real MIPS assembly instructions
  - "Real" MIPS instructions have opcodes, pseudo-instructions do not
  - The assembler often uses the `$at` register (i.e. `$1`) when mapping pseudo-instructions to MIPS instructions
- In MARS, can be looked up in the "help" section

Examples

- `la` is a pseudo-instruction that is translated to `lui` and `ori`
- `bge` checks for greater than or equal to relationships

10 Interrupts

Interrupts

- **Interrupts** take place when an external event requires a change in execution
  - E.g. arithmetic overflow, system calls (`syscall`), undefined instructions
  - Usually signalled by an external input wire, which is checked at the end of each instruction

Interrupts can be handled in two general ways

1. **Polled handling**: the processor branches to the address of interrupt handling code, which begins a sequence of instructions that check the cause of the exception
   - Branches to the handler code sections, depending on the type of exception encountered
   - MIPS uses this type of handling
2. **Vectored handling**: the processor can branch to a different address for each type of exception
   - Each exception address is separated by 1 word
   - A jump instruction is placed at each of these addresses for the handler code for that exception

Interrupt Handling

- In the case of polled interrupt handling, the processor jumps to exception handler code, based on the value in the **cause register**
- If the original program can resume afterwards, this interrupt handler returns to the program by calling `rfe` instruction (i.e. return from exception)
- Otherwise, the stack contents are dumped and execution will continue elsewhere
- Registers `$26` and `$27` are used by the exception handler

Interrupts, from highest priority to lowest

- 0 (INT): external interrupt
  - E.g. pressing the power button
- 4 (ADDRL): address error exception (load or fetch); 5 (ADDRS): address error exception (store)
  - E.g. accessing a bad address
- 6 (IBUS): bus error on instruction fetch; 7 (DBUS): bus error on data fetch
  - E.g. two things writing to the memory bus at the same time
- 8 (Syscall): Syscall exception
- 9 (BKPT): breakpoint exception
  - Pauses the program and waits for user input
- 10 (RI): reserved instruction exception; 12 (OVF): arithmetic overflow exception