CS61C Midterm 1 Review

go to http://tinyurl.com/z7wf2x9
Brought to you by Alex and Steven
List of Topics:

- Number Representation (20min)
- C (1hr)
  - Basics
  - Arrays and Pointers
  - Memory management
- MIPS (1hr)
  - As a ‘programming language’
  - As numbers (instruction format)
- CALL (20)
- Q&A (5 min after each section)
Number Rep:

- Conversion (decimal - binary - hexadecimal)
- Complements (two’s, three’s, etc)
- The extreme values.
- Bit Operations
Number Rep: Conversion (decimal - binary - hexadecimal)

Using the table (Remember you have a one-page cheat sheet)

binary

Decimal

Hexadecimal

Largest power of 2 you can fit (e.g. 128 is the largest power of two to fit in 200)

Sum up Power of 2 at each bit

Power of 16

???
Complement

- Adding 1 will move you one position clockwise
- Once you reach the greatest positive number, you become the smallest negative number
- Two’s complement of 0? What about -8?
Number Rep: Two’s Complement with extremes

8-bit
Two’s Complement
Number Rep: Extremes

With a 4-bit binary string, what is:

- The largest unsigned number.
Number Rep: Extremes

With a 4-bit binary string, what is:

- The largest unsigned number. $8 + 4 + 2 + 1$ or $2^4 - 1$

The key here is to notice that a binary string of all 1’s (e.g. 1111) can be more easily represented as $2^n - 1$ where $n$ is the number of bits plus 1. (e.g. $1111_2 = 10000_2 - 1$)
## Exercise 1: Conversion

Convert the following numbers

<table>
<thead>
<tr>
<th></th>
<th>Unsigned</th>
<th>Signed</th>
<th>One’s complement</th>
<th>Two’s complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000,1000</td>
<td>0000,1111</td>
<td>0000</td>
</tr>
<tr>
<td>-1</td>
<td>N/A</td>
<td>1001</td>
<td>1110</td>
<td>1111</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Exercise 2: Bit Operation Tricks

- Check if an integer is even or odd (return 1 if the number is odd, 0 otherwise)
  
  ```
  return (x & 1);
  ```

- Get the remainder of x when we divide x by $2^n$
  
  ```
  return x & ((1<<n)-1);
  ```
C Topics

- C array and pointer.
  
  ```c
  char *a = {'a', 'b', 'c'}
  char b[] = {'a', 'b', 'c'}
  ```

- & is a function that returns the address (of something actually stored in memory)

- The size of integers/chars/pointers/structs

- malloc/calloc/realloc/free

- memory layout
  - Static: global variables, string literals, pre-defined array
  - Code: code, and read-only values (from #define or const int val;)

- *p++ vs *++p vs ++*p
Postfixal: Will always return original *p

*p++;  
*(p++);  
*(p = p + 1)

The expression returns original *p, not *(p+1)

(*p)++;  
(*p) = (*p) + 1;

The expression returns original *p, not incremented value stored at p
Prefixal: Will do incrementation first, then return

```c
*++p;
*(p = p + 1);
The expression \textit{increments} \( p \), then returns \( *p \) so that in the end, \( * (p+1) \) is returned.
```

```c
++*p;
++(*p)
(*p) = (*p) + 1;
The expression \textit{increments the value in memory it pointed to})
and then returns \( *p \) so that the incremented mem value is returned
```
Exercise: C Memory Layout

1 typedef struct bignum {
2     Int len;
3     Char *num;
4     Char description[100];
5 } bignum_t *res;
6 bignum_t *res;
7
8 int main () {
9     Bignum_t b;
10     B.num = (char *) malloc (5 * sizeof (char));
11     ...
12     }

Exercise: C Memory Layout

- In the above code,

  on line no. 6, 4 bytes has being used in static memory
  on line no. 9, 108 bytes has being used in stack memory
  on line no. 10, 5 bytes has being used in heap memory

```c
1  Typedef struct bignum{
2       Int len;
3       Char *num;
4       Char description[100];
5  }bignum_t res;
6  bignum_t *res;
7
8  int main(){
9       Bignum_t b;
10      B.num = (char *) malloc(5 * sizeof(char));
       ...  
}
```
Exercise: C code-writing -- Tokenizer

Fill in the blanks below for the function tokenize which splits up a string into proper C strings whenever it encounters a specified token (a single character).

**Inputs:**
- char * str - a NULL terminated character array
- char token - a character to split on
  
  - You may assume that the token appears at least once in the input str and the first/last character in the input string are not the token.

**Outputs:** An array of strings split by the provided token (pointers and string data on the heap). The token should never appear in an output string and your output should never contain strings of length zero.
Assume that you have access to the functions:

```c
int count_occurences(char * str, char token); // Counts number of separate words in the original array

char * strcpy (char * destination, char * source); // copies string source to buffer dest
```

Example:

```c
char * string = "Summer*time";

printf(“%s”, tokenize(string, ‘*’)[0]); // prints: Summer

printf(“%s”, tokenize(string, ‘*’)[1]); // prints: time
```
tokenize(char * str, char token) {
    char * copy = malloc(sizeof(char) * strlen(str) + 1);
    strcpy(copy, str);
    char ** string_array = malloc(sizeof(char*) * count_occurences(str, token));

    int index = 0;
    string_array[index] = copy;

    while(*copy != '\0') {
        if (*copy == token && *(copy+1) != token) {
            *copy = '\0';
            index += 1;
            string_array[index] = ++copy;
        } else if (*copy == token) {
            *copy++ = '\0';
        } else {
            *copy++;
        }
    }

    return string_array; }

Exercise: Debugging

Write a function count_az that takes an input string of lower-case letters (only ‘a’ through ‘z’) and returns an array of the number of occurrences of all letters. The returned array will be zero-indexed and the indices will correspond to their respective order in the alphabet (i.e. a = 0, b = 1, ..., z = 25). Fix all the errors; we should be able to call it like this: myAZ = count_az(str); yourAZ = count_az(str);

```c
int count_az(char *str){
    int count[26]; //create the count array
    while(*str){ //Go through the whole string
        int index = &str - 0x97; //The 97 is from the MIPS green sheet
        count[index]++; //Increment the appropriate bucket
        str++; //Go to the next character
    }
    free(str); //free the string storage
}
```
Exercise: Debugging

```c
int count_az(char *str){
    int count[26]; //create the count array
    while(*str){ //Go through the whole string
        int index = &str - 0x97; //The 97 is from the MIPS green sheet
        count[index]++; //Increment the appropriate bucket
        str++;
        //Go to the next character
    }
    free(str); //free the string storage
}
```

<table>
<thead>
<tr>
<th>Line #</th>
<th>add/change/remove</th>
<th>Edit Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>change</td>
<td>int *count_az(char *str) // because we need to return our result</td>
</tr>
<tr>
<td>2</td>
<td>change</td>
<td>int *count = (int *) malloc(sizeof(int)*26); //we can’t return things on stack</td>
</tr>
<tr>
<td>After line 2</td>
<td>add</td>
<td>for( int letter = 0; letter &lt; 26; letter ++ ) count[letter] = 0; //clear heap garbage</td>
</tr>
<tr>
<td>4</td>
<td>change</td>
<td>int index = ( (int) (*str) ) - 97;</td>
</tr>
<tr>
<td>8</td>
<td>change</td>
<td>return count;</td>
</tr>
</tbody>
</table>
Exercise: C Pointers

What does the following program print if the starting address of x is \(1000_{10}\), and the starting address of array arr is \(2000_{10}\)? All addresses are byte-addressed.

```c
#include <stdio.h>

int main() {
    int *p, x;
    int arr[5] = {1000, 2000, 3000, 4000, 5000};
    int *p2;
    p = NULL;
    x = 5000;
    p = &x;
    printf("%d %d %u %u\n", x, *p, p, &x);
    p2 = arr;
    *(p2+1) = *p;
    *p = *p2 + *(p2+2);
    p2++;
    printf("%d %d %d %u\n", x, *p, *p2, p2);
    return 0;
}
```

5000 5000 1000 1000
4000 4000 5000 2004
MIPS topics

- MIPS addressing modes
- MIPS Instruction Format
- Branching (esp. Counting instructions to jump)
- Reading mysterious MIPS code
- C-MIPS translation/MIPS code-writing
MIPS: Green Sheet

http://www-inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf

- Prefixes (giga vs gibi)
- Ascii table
- How to read the MIPS ISA
## Prefixes

Bottom right corner of second page

<table>
<thead>
<tr>
<th>SIZE</th>
<th>PREFIX</th>
<th>SIZE</th>
<th>PREFIX</th>
<th>SIZE</th>
<th>PREFIX</th>
<th>SIZE</th>
<th>PREFIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^3$, $2^{10}$</td>
<td>Kilo-</td>
<td>$10^{15}$, $2^{50}$</td>
<td>Peta-</td>
<td>$10^{-3}$</td>
<td>milli-</td>
<td>$10^{-15}$</td>
<td>femto-</td>
</tr>
<tr>
<td>$10^6$, $2^{20}$</td>
<td>Mega-</td>
<td>$10^{18}$, $2^{60}$</td>
<td>Exa-</td>
<td>$10^{-6}$</td>
<td>micro-</td>
<td>$10^{-18}$</td>
<td>atto-</td>
</tr>
<tr>
<td>$10^9$, $2^{30}$</td>
<td>Giga-</td>
<td>$10^{21}$, $2^{70}$</td>
<td>Zetta-</td>
<td>$10^{-9}$</td>
<td>nano-</td>
<td>$10^{-21}$</td>
<td>zepto-</td>
</tr>
<tr>
<td>$10^{12}$, $2^{40}$</td>
<td>Tera-</td>
<td>$10^{24}$, $2^{80}$</td>
<td>Yotta-</td>
<td>$10^{-12}$</td>
<td>pico-</td>
<td>$10^{-24}$</td>
<td>yocto-</td>
</tr>
</tbody>
</table>

The symbol for each prefix is just its first letter, except μ is used for micro.
### ASCII Table

Not only does it tell you info about the instructions, but you can also find ASCII values.

<table>
<thead>
<tr>
<th>ASCII Symbols</th>
<th>Binary</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>ASCII Character</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>ASCII Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>00 0000</td>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td>64</td>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>00 0001</td>
<td>1</td>
<td>1</td>
<td>SOH</td>
<td>65</td>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>00 0010</td>
<td>2</td>
<td>2</td>
<td>STX</td>
<td>66</td>
<td>42</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>00 0011</td>
<td>3</td>
<td>3</td>
<td>ETX</td>
<td>67</td>
<td>43</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>00 0100</td>
<td>4</td>
<td>4</td>
<td>EOT</td>
<td>68</td>
<td>44</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>00 0101</td>
<td>5</td>
<td>5</td>
<td>ENQ</td>
<td>69</td>
<td>45</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>00 0110</td>
<td>6</td>
<td>6</td>
<td>ACK</td>
<td>70</td>
<td>46</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>00 0111</td>
<td>7</td>
<td>7</td>
<td>BEL</td>
<td>71</td>
<td>47</td>
<td>3</td>
</tr>
<tr>
<td>H</td>
<td>00 1000</td>
<td>8</td>
<td>8</td>
<td>BS</td>
<td>72</td>
<td>48</td>
<td>3</td>
</tr>
<tr>
<td>I</td>
<td>00 1001</td>
<td>9</td>
<td>9</td>
<td>HT</td>
<td>73</td>
<td>49</td>
<td>3</td>
</tr>
<tr>
<td>J</td>
<td>00 1010</td>
<td>10</td>
<td>a</td>
<td>LF</td>
<td>74</td>
<td>4a</td>
<td>3</td>
</tr>
<tr>
<td>K</td>
<td>00 1011</td>
<td>11</td>
<td>b</td>
<td>VT</td>
<td>75</td>
<td>4b</td>
<td>3</td>
</tr>
<tr>
<td>L</td>
<td>00 1100</td>
<td>12</td>
<td>c</td>
<td>FF</td>
<td>76</td>
<td>4c</td>
<td>3</td>
</tr>
<tr>
<td>M</td>
<td>00 1101</td>
<td>13</td>
<td>d</td>
<td>CR</td>
<td>77</td>
<td>4d</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>00 1110</td>
<td>14</td>
<td>e</td>
<td>SO</td>
<td>78</td>
<td>4e</td>
<td>3</td>
</tr>
<tr>
<td>O</td>
<td>00 1111</td>
<td>15</td>
<td>f</td>
<td>SI</td>
<td>79</td>
<td>4f</td>
<td>3</td>
</tr>
<tr>
<td>P</td>
<td>01 0000</td>
<td>16</td>
<td>10</td>
<td>DEL</td>
<td>80</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>Q</td>
<td>01 0001</td>
<td>17</td>
<td>11</td>
<td>DC1</td>
<td>81</td>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>R</td>
<td>01 0010</td>
<td>18</td>
<td>12</td>
<td>DC2</td>
<td>82</td>
<td>52</td>
<td>3</td>
</tr>
<tr>
<td>S</td>
<td>01 0011</td>
<td>19</td>
<td>13</td>
<td>DC3</td>
<td>83</td>
<td>53</td>
<td>3</td>
</tr>
<tr>
<td>T</td>
<td>01 0100</td>
<td>20</td>
<td>14</td>
<td>DC4</td>
<td>84</td>
<td>54</td>
<td>3</td>
</tr>
<tr>
<td>U</td>
<td>01 0101</td>
<td>21</td>
<td>15</td>
<td>NAK</td>
<td>85</td>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>V</td>
<td>01 0110</td>
<td>22</td>
<td>16</td>
<td>SYN</td>
<td>86</td>
<td>56</td>
<td>3</td>
</tr>
<tr>
<td>W</td>
<td>01 0111</td>
<td>23</td>
<td>17</td>
<td>ETB</td>
<td>87</td>
<td>57</td>
<td>3</td>
</tr>
<tr>
<td>X</td>
<td>01 1000</td>
<td>24</td>
<td>18</td>
<td>CAN</td>
<td>88</td>
<td>58</td>
<td>3</td>
</tr>
<tr>
<td>Y</td>
<td>01 1001</td>
<td>25</td>
<td>19</td>
<td>EIM</td>
<td>89</td>
<td>59</td>
<td>3</td>
</tr>
</tbody>
</table>

### MIPS Opcodes

- **add.f**
- **sub.f**
- **mul.f**
- **div.f**
- **sllv**
- **srlv**
- **sra**
- **srev**
- **addi**
- **addiu**
- **jalr**
- **slt**
- **slti**
- **sltiu**
- **andi**
- **ori**
- **xori**
- **lui**
- **mfhi**
- **mthi**
- **mflo**
- **mtlo**
- **mult**
- **multu**
## How to read MIPS

### General Tips:

- **opcode/funct** only applies to **R-types**, otherwise is only the opcode.
- Observe RTL if you are unsure of which registers do what.

### CORE INSTRUCTION SET

<table>
<thead>
<tr>
<th>NAME, MNEMONIC</th>
<th>FORMAT</th>
<th>OPERATION (in Verilog)</th>
<th>OPCODE / FUNCT (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>add</td>
<td>( R[rd] = R[rs] + R[rt] )</td>
<td>( 0 / 20_{hex} )</td>
</tr>
<tr>
<td>Add Immediate</td>
<td>addi</td>
<td>( R[rt] = R[rs] + \text{SignExtImm} )</td>
<td>( 8_{hex} )</td>
</tr>
<tr>
<td>Add Imm. Unsigned</td>
<td>addiu</td>
<td>( R[rt] = R[rs] + \text{SignExtImm} )</td>
<td>( 9_{hex} )</td>
</tr>
<tr>
<td>Add Unsigned</td>
<td>addu</td>
<td>( R[rd] = R[rs] + R[rt] )</td>
<td>( 0 / 21_{hex} )</td>
</tr>
<tr>
<td>And</td>
<td>and</td>
<td>( R[rd] = R[rs] &amp; R[rt] )</td>
<td>( 0 / 24_{hex} )</td>
</tr>
<tr>
<td>And Immediate</td>
<td>andi</td>
<td>( R[rt] = R[rs] &amp; \text{ZeroExtImm} )</td>
<td>( c_{hex} )</td>
</tr>
<tr>
<td>Branch On Equal</td>
<td>beq</td>
<td>( \text{if}(R[rs]==R[rt]) ) ( PC=PC+4+\text{BranchAddr} )</td>
<td>( 4_{hex} )</td>
</tr>
<tr>
<td>Branch On Not Equal</td>
<td>bne</td>
<td>( \text{if}(R[rs]!=R[rt]) ) ( PC=PC+4+\text{BranchAddr} )</td>
<td>( 5_{hex} )</td>
</tr>
<tr>
<td>Jump</td>
<td>j</td>
<td>( PC=\text{JumpAddr} )</td>
<td>( 2_{hex} )</td>
</tr>
<tr>
<td>Jump And Link</td>
<td>jal</td>
<td>( R[31]=PC+8;PC=\text{JumpAddr} )</td>
<td>( 3_{hex} )</td>
</tr>
<tr>
<td>Jump Register</td>
<td>jr</td>
<td>( PC=R[rs] )</td>
<td>( 0 / 08_{hex} )</td>
</tr>
<tr>
<td>Load Byte Unsigned</td>
<td>lbu</td>
<td>( R[rt]={24'b0,M[R[rs] + \text{SignExtImm}][7:0]} )</td>
<td>( 24_{hex} )</td>
</tr>
<tr>
<td>Load Halfword Unsigned</td>
<td>lhu</td>
<td>( R[rt]={16'b0,M[R[rs] + \text{SignExtImm}][15:0]} )</td>
<td>( 25_{hex} )</td>
</tr>
<tr>
<td>Load Linked</td>
<td>ll</td>
<td>( R[rt] = M[R[rs]] + \text{SignExtImm} )</td>
<td>( 30_{hex} )</td>
</tr>
<tr>
<td>Load Upper Imm.</td>
<td>lui</td>
<td>( R[rt] = {\text{imm}, 16'b0} )</td>
<td>( f_{hex} )</td>
</tr>
<tr>
<td>Load Word</td>
<td>lw</td>
<td>( R[rt] = M[R[rs]] + \text{SignExtImm} )</td>
<td>( 23_{hex} )</td>
</tr>
</tbody>
</table>
MIPS topics: Addressing modes

- Branching vs Jump (PC-relative vs Pseudodirect)

0x20000004 Label1: `beq $4, $0, Label2`

Q1: Can you write this instruction?

0x20000008 add $4, $4, $0

Q2: What about this one?

0x2000000c j Label1

Q3: And this one?

0x20000010 Label2: `slt $2,$3,5`

...
Exercise: instruction translation.

0x00000000     add $t0 $a0 $0  <=> 0x00804020
0x00000004     add $t1 $a1 $0  <=> __________
0x00000008 LabelA: add $t2 $0 $0  <=> 0x00005020
0x0000000c LabelB: beq $t0 $0 END  <=> __________
0x00000010 LabelC: addi $t0 $t0 -1  <=> __________
0x00000014     lw $t3 0($t1)  <=> __________
0x00000018     __________  <=> 0x014b5020
0x0000001c     addi $t1 $t1 4  <=> 0x21290004
0x00000020     __________  <=> 0x08000003
0x00000024 END: add $v0 $t2 $0  <=> 0x01201020

0x0c:

```
0001 00 01 000 0 0000 0000 0000 0000 0101
```
Exercise: instruction translation.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Corresponding Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>add $t0 $a0 $0</td>
<td>0x00804020</td>
</tr>
<tr>
<td>0x00000004</td>
<td>add $t1 $a1 $0</td>
<td>___</td>
</tr>
<tr>
<td>0x00000008</td>
<td>LabelA: add $t2 $0 $0</td>
<td>0x00005020</td>
</tr>
<tr>
<td>0x0000000c</td>
<td>LabelB: beq $t0 $0 END</td>
<td>___</td>
</tr>
<tr>
<td>0x00000010</td>
<td>LabelC: addi $t0 $t0 -1</td>
<td>___</td>
</tr>
<tr>
<td>0x00000014</td>
<td>lw $t3 0($t1)</td>
<td>___</td>
</tr>
<tr>
<td>0x00000018</td>
<td>___</td>
<td>0x014b5020</td>
</tr>
<tr>
<td>0x0000001c</td>
<td>addi $t1 $t1 4</td>
<td>0x21290004</td>
</tr>
<tr>
<td>0x00000020</td>
<td>___</td>
<td>0x08000003</td>
</tr>
<tr>
<td>0x00000024</td>
<td>END: add $v0 $t2 $0</td>
<td>0x01201020</td>
</tr>
</tbody>
</table>

0x20: 0x08000003

<table>
<thead>
<tr>
<th>0b</th>
<th>0 0 0 0</th>
<th>1 0 0 0</th>
<th>0 0 0 0</th>
<th>0 0 0 0</th>
<th>0 0 0 0</th>
<th>0 0 0 0</th>
<th>0 0 0 0</th>
<th>0 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>LabelB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS: Instruction Format

Why is:

- the rs, rt, rd field 5 bits?
- the shamt field is 5 bits?
- the immediate field 16 bits?

If we ignore the fact that each register is 32 bits, what changes are necessary if we want to:

- Increase the number of registers to 64?
- Have 64-bit registers instead of 32?
MIPS topics: Reading mysterious MIPS code

● Practice, Practice, Practice!
  ○ There are common patterns for a MIPS function (e.g. If you want to have a constant, use `addiu` to load that constant into a register so you can use it. `addiu` is also used to increment the loop index.)
  ○ Shifting a register by 2 typically turns (integer) index into bytes. (You will have to know that shifting left by 1 byte means multiplying the number by 2)

● Find a way to track your registers (their types and purposes)

● Memory accessing: using load and store instructions to access memory (basically whenever you hear the word array.)
  ○ The difference between `lw`, `lb` and `sw`, `sb"
MIPSterious exercise:

# $a0 - > array, $a1 -> length of array

Mystery:    add $v0, $0, $0

Label:   slti $t0, $a1, 2
          bne $t0, $0, Done  # exit if fewer than 2 elements remaining
          lw $t0, 0($a0)
          lw $t1, 4($a0)  ⇐ Load two consecutive elements in the array
          slt $t2, $t1,$t0  ⇐ compare the two elements, set t2 to 1 if the the first is larger
          add $v0, $v0,$t2  ⇐ put the results into v0
          subi $a1, $a1,1
          addi $a0, $a0,4  ⇐ incrementing the pointer, decrement the counter
          j    Label

Done:      beq $v0, $0, Return1
            addi $v0, $0, 0
            jr    $ra

Return1:   addi $v0, $0, 1
            jr    $ra

This mysterious MIPS code checks if the array is non-descending. Returns 1 for yes, 0 otherwise.
MIPS topics: C-MIPS translation/MIPS code-writing

- When making a function call, use jal instead of j because otherwise you will lose the way back (esp. Important for recursive functions.)
- To prepare the arguments, put them into $a0 - $a3. To return a result, put them in $v0 or $v1
- When you are confused, at least write the prologue and epilogue.
  - Only save onto the stack the saved registers that you will change. You do not need to save the temporary registers.
  - And remember, you decrement the stack pointer to open a stack and increment it to close it.
Exercise: C-MIPS

The following C code recursively sums the elements in an array of length n.

```c
int32_t sum_arr(int32_t *arr, size_t n) {
    if (n) {
        return sum_arr(arr + 1, n - 1) + arr[0];
    }
    return 0;
}
```

Translate it into MIPS! No pseudoinstructions. Assume *arr is stored in $a0 and n is stored in $a1
```c
int32_t sum_arr(int32_t *arr, size_t n) {
    if (n) {
        return sum_arr(arr + 1, n - 1) + arr[0];
    } return 0;
}
```

```asm
sum_arr: bne $a1, $zero non_zero
   addu $v0, $0, $0
   jr $ra
non_zero:
   addiu $sp, $sp, -8
   sw $s0, 4($sp)
   sw $ra, 0($sp)
   jal sum_arr # Make the recursive call
   lw $t0, 0($s0)
   addu $v0, $v0, $t0 # Then add arr[0] to the result
   lw $s0, 4($sp)
   lw $ra, 0($sp)
   addiu $sp, $sp, 8
   addiu $s0, $a0, 0 # Store arr into $s0
   addiu $a0, $a0, 4
   addiu $a1, $a1, -1
   jr $ra
```
```
Assume (for simplification) that main returns a value to its caller (Unix) through standard MIPS procedure calling conventions. We wish to see how long our command-line inputs arguments are:

—$a0 holds the number of words after unix%
  (count_argument_characters is considered a word)
—$a1 holds the address to a pointer array of the argument’s words
CALL Topics

Steps to Starting a Program:

1) Compiler
2) Assembler
3) Linker
4) Loader
Interpretation vs Compilation.

True or False

- Interpreted language is faster  
  False; Compiled language is usually faster due to compiler optimization

- Interpreted language is more verbose  
  Compiled language is usually more verbose. E.g. JAVA

- Interpreted language is platform dependent.  
  False. Compiler is platform dependent. E.g. we have C-MIPS compiler as well as C-x86 compiler.
CALL me maybe?

- When is the symbol table and relocation table created?
  
  During the assembler stage

- True or False: The system will empty all the registers in the loading stage so the program can start without garbage
  
  False. The system will empty all the registers except the stack pointer, since the system must put the address of the top of the stack into $sp.
• By the time you reach the linker stage, will you see instructions like mul? How about mult?

No, because mul is a MAL, which will be replaced by TAL by the stage of linker. You will see mult however since it’s one of MIPS TAL instruction

• And how about addiu $t1,$t2, 0x10000? And what addiu $t1, $t2, 0x1000

You will not see addiu $t1,$t2, 0x10000 because 0x10000 will not fit into the 16-bit immediate field. Multiple instructions have to be used to finish this task. The latter is perfectly fine
When is immediate field in branching instructions (beq, bne) decided?

During the assembler stage

When is the 26 bits in j-type instructions decided?

During the linker stage
Two’s Complement: Proof

~x is flipping all bits of x

Two’s complement states that \(-x = \sim x + 1\)

Property we know:
\[x + \sim x = 0b111...\]
Which is always -1
\[x + \sim x = -1\]
\[x + \sim x +1 = 0\]
\[\sim x + 1 = -x\]
\[-x = \sim x + 1\]