UC Berkeley : CS61C (Garcia & Lustig) : Midterm part 1 : 2014-10-10

_____ _____ cs61c-_____

Name (first last) *SID* *Login*

_____ _____

← *Name of person on left (or aisle)* *Name of person on right (or aisle)* →

Question 1: *Running in circles* (25 min, 18 pts)

A *nibble* is half of a byte (4 bits). You'd like to implement **`LoadNibble`** in MAL MIPS, a function that takes one **`uint32_t`** argument **`N`** and returns the **`N`**th nibble of memory in the lowest 4 bits of the return register (the other 28 bits should be 0). Note: The **`N`**th nibble immediately follows the **`N-1`**th nibble without overlapping; see box . The MIPS instruction **`srlv`** ("shift right variable") might be useful here; it operates like the **`shamt`**-based right-shift, except that its 3rd *register* argument is the variable amount to shift by.

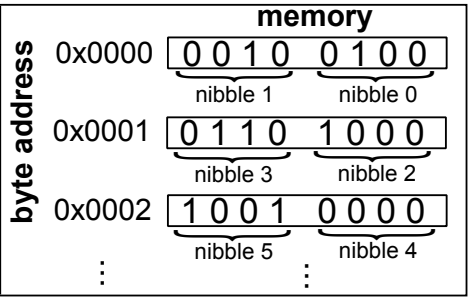
a) What fraction of all the nibbles of memory can you access? _____

b) Implement **`LoadNibble`** by filling in the blanks:

```
LoadNibble: _____ $t0  ____ ____ # figure out which byte contains that nibble
            _____ $a1 0(____)
            _____ $a0  ____ ____
sll $a0 $a0 2 # we needed this!
 gone1:     _____ ____ ____ ____
 gone2:     _____ ____ ____ ____
            jr $ra
```

for N=2, **`LoadNibble`** returns **`0b1000`**

for N=5, **`LoadNibble`** returns **`0b1001`**



c) We want to rewrite **`LoadNibble`** to make use of a helper function **`Helper`** that will take two arguments. The first is an index **`i`** from 0-1 and the second is a byte **`B`**. **`Helper`** returns the **`i`**th nibble in **`B`** placed in the lowest 4 bits of the return value (the rest 0s).

E.g., **`Helper(0, 0b01100100)`** → **`0b0100`** and **`Helper(1, 0b01100100)`** → **`0b0110`**

We decide we don't need the two MIPS instructions labeled "**`gone1`**" and "**`gone2`**". What would you replace these instructions (and the **`sll`**) with to call **`Helper`** and implement **`LoadNibble`** successfully? Write the replacement below. Follow calling conventions and complete it in the fewest lines possible.

```
_____________________________  # this line may not be necessary
_____________________________  # this line may not be necessary
_____________________________  # this line may not be necessary
______ Helper   # j works too, all other lines blank (since $ra = LoadNibble's caller)!
_____________________________  # this line may not be necessary
_____________________________  # this line may not be necessary
_____________________________  # this line may not be necessary
```

Question 2: *I can C clearly now, the rain is gone...* (25 min, 18 pts)

A) Fill in the blank to complete this function that parses a string of *octal digits* (base 8) into a **`uint64_t`**. For example, calling **`parse_octal("71")`** should return the number 57. Do not use the comma operator, nested assignment, prefix/postfix operators, or function calls. You may assume that the given number "fits" into a **`uint64_t`.** (**Hint:** The backside of the MIPS green sheet may help.)

```
uint64_t parse_octal(char *s) {
      uint64_t r = 0;
      while(*s){
            r = ____________________________________________;
            s++;
      }
      return r;
}
```

B) We have the following data *packed tightly (no padding)* into the struct **`data`**, and some more code below:

```
struct {
      int16_t a;
      char b[2+(UNKNOWN_LENGTH*4)];
      int32_t c;
      int32_t d;
} data;
```

Fill in the blanks with an equivalent expression using only the pointer **`s`**, pointer arithmetic, casting, and the function **`strlen()`**. You may **NOT** use **`UNKNOWN_LENGTH`**. Assume **`sizeof(char) = 1`**.

```
/* … Some code here that fills in data.b with the longest string possible … */
char *s  = data.b;

*( (int16_t *) ____________________________________________ ) = -1;  // data.a = -1;

*( (int32_t *) ____________________________________________ ) = -1;  // data.d = -1;
```

C) Here we have a *LR-tree*, defined as a node with two arrays of child pointers: two left children and two right children. Each node also contains a pointer to its parent node, a unique integer ID value, and a string name field. Root nodes will have a **`NULL`** parent pointer, and leaf nodes will have arrays of **`NULL`** children pointers.

```
struct lr_tree{
  char *name;
  uint64_t ID;
  struct lr_tree *left_children[2];
  struct lr_tree *right_children[2];
  struct lr_tree *parent;
};
```

Fill in the blanks to complete this function that frees a LR-tree if called with the root of the tree. You must free **ALL** data associated with this LR-tree! You might not need all of the blanks, in which case use the most minimal number of blanks possible. Do not use the comma operator, nested assignment, or prefix/postfix operators.

```
void free_lr_tree (struct lr_tree *p) {

      if (_____________________________________________________ ){
            for(size_t x = 0; x < 2; x++) {

                  _____________________________________________________;

                  _____________________________________________________;
            }

            _____________________________________________________;

            _____________________________________________________;

            _____________________________________________________;
      }
}
```