Programming Assignment 2 (PA2) - BCD Clock

Milestone Due:  **Wednesday, May 3 @ 11:59pm**
Final Due:  **Wednesday, May 10 @ 11:59 pm**

**Assignment Overview**
For this assignment you will build a BCD (Binary Coded Decimal) clock simulator. BCD is a way to represent decimal digits (0-9) in 4 bits (0000-1001). You will use an array of 3 unsigned chars to store the tens and ones values of the clock's hours, minutes, and seconds. The tens digit will be stored in the upper 4 bits (nibble) of each byte. The ones digit will be stored in the lower 4 bits of each byte. The BCD clock display will be similar to the BCD clock I have in my office.

The purpose of this programming assignment is to gain more experience with SPARC assembly bit-wise operations, memory loads and stores (ld/st), allocating local variables on the runtime stack (using the save instruction) and accessing them relative to the frame pointer (%fp) by converting a function you wrote in C in PA1 into assembly with an additional parameter. You will use Standard C Library routines and varied techniques to communicate C Preprocessor-derived values to your assembly routines (accessing global variables set in a C function and calling C functions from assembly).

**Grading**
- **README: 10 points** - See README File section
- **Compiling: 5 points** - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. **NO EXCEPTIONS!**
- **Style: 20 points** - See Style Requirements [here](http://cseweb.ucsd.edu/~ricko/CSE30StyleGuidlines.pdf)
- **Correctness: 65 points**
  - **Milestone (15 points)** - To be distributed across the Milestone functions (see below)
  - **Make sure you have all files tracked in Git.**
- **Extra Credit: 5 points** - View Extra Credit section for more information.
- **Wrong Language:** You will lose 10 points for each module in the wrong language, C vs. Assembly or vice versa.

**Getting Started**
Follow these steps to acquire the starter files and prepare your Git repository.

**Gathering Starter Files:**
The first step is to gather all the appropriate files for this assignment.
Connect to ieng9 via ssh (replace cs30xyz with YOUR cs30 account).

```
$ ssh cs30xyz@ieng9.ucsd.edu
```

Create and enter the pa2 working directory.
```
$ mkdir ~/pa2
$ cd ~/pa2
```

Copy the starter files from the public directory.
```
$ cp -r ~/../public/pa2StarterFiles/* ~/pa2/
```
Starter files provided:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pa2.h</td>
<td>test.h</td>
</tr>
<tr>
<td>pa2Strings.h</td>
<td>testsetBCDClock.c</td>
</tr>
<tr>
<td>pa2Globals.c</td>
<td>pa2gettimeofday.c</td>
</tr>
<tr>
<td></td>
<td>Makefile</td>
</tr>
</tbody>
</table>

Preparing Git Repository:
You are required to use Git with this and all future programming assignments. Refer to the PA0 writeup for how to set up your local git repository.

Sample Output
A sample stripped executable provided for you to try and compare your output against is available in the public directory. Note that you cannot copy it to your own directory; you can only run it using the following command (where you will also pass in the command line arguments):

```
$ ~/../public/pa2test
```

If there is a discrepancy between the sample output in this document and the pa2test output, follow the pa2test output.

The output of your program MUST match exactly as it appears in the pa2test output. You need to pay attention to the order of error messages!

Below are some brief example outputs of this program. Make sure you experiment with the public executable to further understand the program behavior. Bolded text is what you type in the terminal.

1. Command-line Parsing Errors
1.1. Too many command line arguments.
```
[cs30xyz@ieng9]:pa2$ ./pa2 -2 25:0:0 too_many
```

Usage: ./pa2 [numTicks [startTime]] | [--help]
- numTicks -- (optional) number of clock ticks to display
  Must be within the limits of [0-61]
  Default number of ticks is 7
- startTime -- (optional) starting time of the clock entered as HH:MM:SS
  HH must be within the limits of [0-23]
  MM must be within the limits of [0-59]
  SS must be within the limits of [0-59]
  Default start time is the current time

```
[cs30xyz@ieng9]:pa2$
```

1.2. Invalid number of ticks.
```
[cs30xyz@ieng9]:pa2$ ./pa2 not_an_int 10:00:00
   Error parsing: Number of clock ticks
```

Usage: ./pa2 [numTicks [startTime]] | [--help]
- numTicks -- (optional) number of clock ticks to display
  Must be within the limits of [0-61]
  Default number of ticks is 7
- startTime -- (optional) starting time of the clock entered as HH:MM:SS
  HH must be within the limits of [0-23]
MM must be within the limits of [0-59]
SS must be within the limits of [0-59]
Default start time is the current time

1.3. Invalid number of ticks and invalid starting time.

Usage: ./pa2 [numTicks [startTime]] | [--help]

numTicks -- (optional) number of clock ticks to display
Must be within the limits of [0-61]
Default number of ticks is 7

startTime -- (optional) starting time of the clock entered as HH:MM:SS
HH must be within the limits of [0-23]
MM must be within the limits of [0-59]
SS must be within the limits of [0-59]
Default start time is the current time

1.4 Number of ticks outside limits.

Usage: ./pa2 [numTicks [startTime]] | [--help]

numTicks -- (optional) number of clock ticks to display
Must be within the limits of [0-61]
Default number of ticks is 7

startTime -- (optional) starting time of the clock entered as HH:MM:SS
HH must be within the limits of [0-23]
MM must be within the limits of [0-59]
SS must be within the limits of [0-59]
Default start time is the current time

1.5 Starting time minutes and seconds outside limits.

Usage: ./pa2 [numTicks [startTime]] | [--help]

numTicks -- (optional) number of clock ticks to display
Must be within the limits of [0-61]
Default number of ticks is 7

startTime -- (optional) starting time of the clock entered as HH:MM:SS
HH must be within the limits of [0-23]
MM must be within the limits of [0-59]
SS must be within the limits of [0-59]
Default start time is the current time
1.6 Number of ticks outside limits and starting time hours outside limits.

Usage: ./pa2 [numTicks [startTime]] | [--help]

numTicks -- (optional) number of clock ticks to display
   Must be within the limits of [0-61]
   Default number of ticks is 7

startTime -- (optional) starting time of the clock entered as HH:MM:SS
   HH must be within the limits of [0-23]
   MM must be within the limits of [0-59]
   SS must be within the limits of [0-59]
   Default start time is the current time

2. Valid Output

2.0 Help message.

Usage: ./pa2 [numTicks [startTime]] | [--help]

numTicks -- (optional) number of clock ticks to display
   Must be within the limits of [0-61]
   Default number of ticks is 7

startTime -- (optional) starting time of the clock entered as HH:MM:SS
   HH must be within the limits of [0-23]
   MM must be within the limits of [0-59]
   SS must be within the limits of [0-59]
   Default start time is the current time

2.1 No arguments, starting time defaults to the current time and number of clock ticks defaults to 7 ticks.

2.2. Number of ticks specified.

[cs30xyz@ieng9]:pa2$ ./pa2 2
Thu Apr 27 14:43:28 2017
2.3. Number of ticks and starting time specified.

```bash
[cs30xyz@ieng9]:pa2$ ./pa2 0 11:30:05
```

```
###
# . . . . . O #
# . O 0 . . . #
# . . 0 . 0 . #
# 0 . 0 . O . #
# 11 30 05 #
###
```

```
[cs30xyz@ieng9]:pa2$
```

2.4. Number of ticks and starting time specified, hour change results in ding dongs.

```bash
[cs30xyz@ieng9]:pa2$ ./pa2 2 10:59:59
```

```
###
# .. . O . O #
# .. 0 . 0 . #
# .. . . . #
# 0 . 0 0 0 #
# 10 59 59 #
###
```

```
[cs30xyz@ieng9]:pa2$
```
2.5. Number of ticks and starting time specified, hour change results in Noon! message.

[cs30xyz@ieng9]:pa2$ ./pa2 11:59:59

Noon!

[cs30xyz@ieng9]:pa2$

2.6. Number of ticks and starting time specified, hour change results in Midnight! message.

[cs30xyz@ieng9]:pa2$ ./pa2 23:59:59

Noon!

[cs30xyz@ieng9]:pa2$
Detailed Overview
The function prototypes for the various C and Assembly functions are as follows.

C routines:
int main( int argc, char * argv[] );
unsigned long parseStartTime( struct tm * tmPtr, const char * time );
void displayBCDClock( const unsigned char clock[] );
void chime( const unsigned char clock[] );
void updateBCDClock( unsigned char clock[] );
void usage( const char * programName );
int pa2gettimeofday( struct timeval * tp, void * p );

Assembly routines:
void setBCDClock( const struct tm * tmPtr, unsigned char clock[] );
unsigned char updateBCDValue( unsigned char BCDbits, const unsigned int maxValue );
long withinLimits( long num, long min, long max );
void printChar( char c );

For the Milestone, you will need to complete:
parseStartTime.c    setBCDClock.s    updateBCDValue.s    updateBCDClock.c

C Functions to be Written
Listed below are the modules to be written in C.

main.c
int main( int argc, char * argv[] );
Note:
- Keep in mind that all the error strings have format specifiers, so be sure to add the appropriate arguments when printing error messages.
- Make sure you use your withinLimits() function when checking the limits of the command line arguments.
- Make sure that you free any memory that was allocated before returning from an error.

The main function will drive the rest of the program. It will create the BCD clock array, perform input checking by parsing the command-line arguments, and display the BCD clock for a specified (or default) number of ticks if no errors were encountered.

First, create a zero-filled array of unsigned char on the heap (hint: man -s3c calloc). This array contains the hours, minutes, and seconds values of the BCD clock that we will be displaying and updating throughout the program.

Our first error check will be to ensure that the user didn't enter more than the maximum number of command line arguments allowed (use the MAX_ARGS constant from pa2.h). If they did, print the usage (using your usage() function), free the memory allocated for the clock array, and return EXIT_FAILURE right away.

Now we can parse the command line arguments:

1. If the first command line argument is STR_HELP ("--help"), print usage, free the memory allocated for the clock array, and return EXIT_SUCCESS right away.

2. numTicks:
   a. If numTicks was not entered as the first command line argument, initialize it with DEF_TICK.
   b. If numTicks was specified as the first argument: convert numTicks from a string to a long using strtol() (see man -s3c strtol). Remember to set the global variable errno to 0 right before each call to strtol() to accurately identify when errors occur (see man -s2 intro).
      If numTicks was successfully converted, make sure it is within the limits defined in pa2.h.

3. startTime:
   a. If startTime was not entered as the second command line argument, populate the struct tm with the default time. To set the default startTime into a struct tm, we must use pa2gettimeofday() and localtime() to get the current time and then convert it to the system’s local time. (Hint: look up struct timeval, struct tm and how you need to make use of these structures in function calls).
   b. If startTime was specified as the second command line argument, populate the struct tm by calling parseStartTime(). Check the return value of parseStartTime() for errors in the order they appear below:

<table>
<thead>
<tr>
<th>If return value contains</th>
<th>How to Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR_TIME_FORMAT</td>
<td>Print out error message using STR_TIME_FORMAT_ERR</td>
</tr>
<tr>
<td>ERR_HR_VALUE</td>
<td>Print out error message using STR_PARSE_ERR</td>
</tr>
<tr>
<td>ERR_HR_LIMIT</td>
<td>Print out error message using STR_OUT_OF_LIMITS</td>
</tr>
<tr>
<td>ERR_MIN_VALUE</td>
<td>Print out error message using STR_PARSE_ERR</td>
</tr>
<tr>
<td>ERR_MIN_LIMITS</td>
<td>Print out error message using STR_OUT_OF_LIMITS</td>
</tr>
</tbody>
</table>
If any error occurs during the argument parsing, when finishing parsing all arguments, print usage, free the clock array allocated, and return EXIT_FAILURE.

If no errors were encountered:
- If the startTime is not specified by the user, print out current time (using the struct tm pointer that points to the populated struct tm structure) before displaying any BCD clock. 
  Hint: use asctime() to format the current time printed.
- Set the BCD Clock using the struct tm pointer (that points to the populated struct tm structure) and your setBCDClock() function.
- For each tick in numTicks, display the BCD clock, use chime() to print out any possible bell rings, and then update the clock after each tick. (Note that if numTicks is 8, we will print the clock a total of 9 times because we want to print the clock with the initial startTime before we begin the ticks--see sample output and public executable, and make sure your output matches exactly).

Before returning from the main function, don't forget to free the clock array allocated on the heap.

Reasons for error:
- Wrong number of command line arguments.
- Error parsing numTicks or numTicks wasn't within limits.
- startTime was not formatted correctly
- Error parsing values of startTime or values of startTime were not within limits

Return Value: EXIT_SUCCESS on success; EXIT_FAILURE on encountering any errors

parseStartTime.c

unsigned long parseStartTime( struct tm * tmPtr, const char * time );

This function will parse the time string and populate the tm_sec, tm_min, and tm_hour members of the struct tm pointed to by tmPtr. If there are no errors in this process, you should return 0. Otherwise, the appropriate error flags/bits should be set in the unsigned long bitmap to be returned (see pa2.h for the error flags). You should not print any error messages in this function.

In order to parse the time string, you will need to:
- Copy the contents of time into a local char array (see man -s3c strncpy).
- Count the number of time separators (':' ) in the string. If the string contains the wrong number of separators (see pa2.h), set the ERR_TIME_FORMAT bit in the error bitmap and return.
- Convert each unit of time (hours, minutes, seconds) to an integer value (you should be using strtol()). From this point forward if we encounter any errors, do not return immediately, just set the appropriate error flag and continue (bit operations are required for this). Remember that strtol() will not function properly if the string you give it is not null terminated. A useful function here is strchr() (man -s3c strchr). Think about how you can use this function to ensure each string you pass into strtol() is null terminated. After each call to strtol() you will need to check errno and endptr, setting the appropriate error bits if any errors occurred. If strtol() did not encounter any errors, you also need to ensure that each time unit is within the appropriate limits (use withinLimits() with the constants defined in pa2.h). If a time unit is outside its limits, set the appropriate error flag.
Examples of errors:

<table>
<thead>
<tr>
<th>time</th>
<th>flags set in error bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:15:a2</td>
<td>ERR_SEC_VALUE</td>
</tr>
<tr>
<td>ab:cd:25</td>
<td>ERR_HR_VALUE, ERR_MIN_VALUE</td>
</tr>
<tr>
<td>85:5f:b5</td>
<td>ERR_HR_LIMITS, ERR_MIN_VALUE, ERR_SEC_VALUE</td>
</tr>
<tr>
<td>1z0:16y20</td>
<td>ERR_TIME_FORMAT</td>
</tr>
</tbody>
</table>

Reasons for error:
- time did not contain the correct number of separators.
- Error parsing the hours, minutes, or seconds from a string to a long (remember to set errno to 0 right before each call to strtol() so you can accurately identify when an error occurs).
- Hours, minutes, or seconds are outside their respective limits.

Return Value: Return an unsigned long bitmap where each bit indicates whether or not that specific error occurred, according to the error flags defined in pa2.h. If no errors occurred, this bitmap will just be 0.

displayBCDClock.c

```c
void displayBCDClock( const unsigned char clock[] );
```

This function displays the BCD clock based on the passed-in clock array. For each entry in the array (hours, minutes, seconds), the upper nibble (4 bits) represents the tens place, and the lower nibble represents the ones place of that value. To display the BCD clock, represent each bit in these nibbles by 'O' or '.':
- If the current bit is set (1): output a capital 'O' to indicate a lit LED
- If the current bit is not set (0): output a dot '.' to indicate a unlit LED

Note that the character for a lit LED is a capital O (as in “Oh my goodness, what a cool clock”), and not a 0 (as in “There is zero parking on campus”).

For example: BCD clock that displays time 21:56:37 with input clock = {0x21, 0x56, 0x37}:

```
clock[0] Upper nibble: 2 → 0010₂ → . . O .
    #. oo . . #
    # 0. . oo #
    #. 0 0 . 0 #
    # 21 56 37 #

    #. 0 0 . 0 #
    # 0 0 . 0 #
    # 56 37 #

    # 0 0 . 0 #
    # 0 . 0 #
    # 37 #
```
Note that the layout for each entry (hours, minutes, seconds) is displayed *vertically*, but you can only print out LEDs in each line from left to right (horizontally). There are many ways to implement this function. One way would be to use bit masks to extract bits from each nibble of the clock array and print out the corresponding characters to indicate lit or unlit LEDs.

All the characters you will need to print are defined as constants in `pa2.h`. Make sure you also print the border made up of '#' as shown in the example. (You might find some inspiration from PA1 for how to do that).

**Return Value:** None.

```c
chime.c
void chime( const unsigned char clock[] );
```

This function will make the clock “chime” on the hour by printing out the appropriate “chime” when the time stored in `clock` is on an hour boundary (meaning the time is exactly 1 o’clock, 2 o’clock, 3 o’clock, etc.). If the time stored in `clock` is not on an hour boundary, don’t print anything and simply return.

Otherwise, depending on the hour, print out an alternating pattern of “Ding!” and “Dong!” hour number of times (always starting with “Ding!”). If the time is exactly 12:00:00, print out “Noon!” instead of dings and dongs. If the time is exactly 00:00:00, print out “Midnight!” instead of dings and dongs.

**For example:**

<table>
<thead>
<tr>
<th>clock</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>04:20:30</td>
<td>Ding! Dong! Ding!</td>
</tr>
<tr>
<td>03:00:00</td>
<td>Ding! Dong!</td>
</tr>
<tr>
<td>02:00:00</td>
<td>Ding! Dong!</td>
</tr>
<tr>
<td>12:00:00</td>
<td>Noon!</td>
</tr>
<tr>
<td>00:00:00</td>
<td>Midnight!</td>
</tr>
</tbody>
</table>

**Return Value:** None.

```c
updateBCDClock.c
void updateBCDClock( unsigned char clock[] );
```

This function is used to increment `clock` by one second using `updateBCDValue()`. You should start by incrementing the ones position in seconds, then check if you need to increment the tens position in seconds, continuing this all the way to the tens position in hours.

`updateBCDValue()` only takes in 4 bits (one nibble—which corresponds to one decimal digit of `clock`) at a time. This means you will need to do some bit operations to extract the ones and tens place separately for each unit of time (see constants defined in `pa2.h`). Make sure you update `clock` after each call to `updateBCDValue()`.
For example:

<table>
<thead>
<tr>
<th>original clock</th>
<th>updated clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:20:30</td>
<td>10:20:31</td>
</tr>
<tr>
<td>05:19:59</td>
<td>05:20:00</td>
</tr>
<tr>
<td>09:59:59</td>
<td>10:00:00</td>
</tr>
<tr>
<td>23:59:59</td>
<td>00:00:00</td>
</tr>
</tbody>
</table>

**Return Value:** None (clock is updated).

---

**usage.c**

```c
void usage( const char * programName );
```

This routine outputs the usage message to `stderr` with the name of the program passed as the only argument. When printing the usage, make sure to pass in the appropriate constants for each of the format strings in `USAGE` (see `pa2.h` and `pa2Strings.h`).

**Return Value:** None.

---

**pa2gettimeofday.c**

```c
int pa2gettimeofday( struct timeval * tp, void * p );
```

This function is provided for you. This routine calls the function `gettimeofday()` which will populate the `struct timeval` passed in. See `man -s3c gettimeofday` for information on how to use this function. Because `gettimeofday()` ignores the `void * p` passed in, just pass in a null pointer when you call this function from `main()`.

**Return Value:** Return 0 to indicate success, and -1 to indicate an error (you do not need to check for errors from this function when calling it from `main()`).

---

**Assembly Functions to be Written**

Listed below are the modules to be written in Assembly.

**setBCDClock.s**

```c
void setBCDClock( const struct tm * tmPtr, unsigned char clock[] );
```

This function will set the time in the BCD `clock` array using the hours, minutes, and seconds in `tmPtr`. `tmPtr` is a `struct tm` that has the following members: `tm_sec`, `tm_min`, `tm_hour` (see `man -s3c ctime` for more information). Use the offsets defined in `pa2Globals.c` to access these members.

You will need to set each digit in `clock` separately (make sure you understand BCD notation). This means for each byte in `clock`, you'll need to set the tens digit of that time value in the upper nibble, and the ones digit of that time value in the lower nibble. You will need to use bit operations to do this.

**For example:** If the time is 10:38:52 (HH:MM:SS), the following bit patterns should be stored in `clock`. 

```c
00001010
00000110
00000101
00000111
```
Hint: Because we are directly modifying `clock`, you will need to use multiple loads and stores.

**Return Value:** None (`clock` is updated).

### `updateBCDValue.s`

```c
unsigned char updateBCDValue( unsigned char BCDbits,
                                  const unsigned int maxValue );
```

This function will increment the 4 BCD bits (nibble) passed in as `BCDbits`. If the incremented value is larger than the `maxValue`, return 0 to indicate that the next nibble should be incremented. Otherwise, return the incremented value.

For example:

<table>
<thead>
<tr>
<th>BCDbits</th>
<th>maxValue</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

**Return Value:** Return 0 if incremented `BCDbits > maxValue`. Otherwise, return incremented `BCDbits`.

### `withinLimits.s`

```c
long withinLimits( long num, long min, long max );
```

This is just your `withinLimits()` from PA1 (copy it over from your `~/pa1` directory).

**Reasons for error:**

- `min` is greater than `max` → return `-1`

**Return Value:** If any errors occur, return `-1`. Otherwise, return 1 to represent true, 0 to represent false.

### `printChar.s`

```c
void printChar( char c );
```

This is just your `printChar()` from PA1 (copy it over from your `~/pa1` directory).

**Return Value:** None.

### Unit Testing

You are provided with only one basic unit test file for the Milestone function, `setBCDClock()`. This file only has minimal test cases and is only meant to give you an idea of how to write your own unit test files. **You must write unit test files for each of the Milestone functions, as well as add several of your own thorough**
**Unit tests you need to complete:**

- testparseStartTime.c
- testsetBCDClock.c
- testupdateBCDValue.c
- testupdateBCDClock.c

**To compile:**

```
$ make testsetBCDClock
```

**To run:**

```
$ ./testsetBCDClock
```

(Replace "testsetBCDClock" with the appropriate file names to compile and run the other unit tests)

**README File**

Your README file for this and all assignments should contain:

- High level description of what your program does.
- How to compile it (be more specific than: just typing “make”--i.e., what directory should you be in?, where should the source files be?, etc.).
- How to run it (give an example).
- An example of normal output and where that normal output goes (stdout or a file or ???).
- An example of abnormal/error output and where that error output goes (stderr usually).
- How you tested your program (what test values you used to test normal and error states) and showing your tests covered all parts of your code (test coverage). (Be more specific than diff’ing your output with the solution output--i.e., what are some specific test cases you tried?, what different types of cases did you test?, etc.)
- Anything else that you would want/need to communicate with someone who has not read the assignment write-up but may want to compile and run your program.
- Answers to questions (if there are any).

**Questions to Answer in the README**

1. Why are professional engineers expected to act with integrity?
2. What happens when you select text and then middle click in the vim editor when in insert/input mode?
3. What is a .vimrc file, and how do you create/edit them?
4. What is the command to cut a full line of text to the clipboard in vim? How do you paste it? (Both the questions refer to using the keyboard, not using the mouse).
5. How do you search for a string in vim?
6. How do you change the permissions on a file? Let’s say we want to give read permission to the group (without changing the existing permissions for user and world). Specify the command to do this.
7. What does realloc() do? (Hint: read the manpage: man -s3c realloc)
   What function would realloc() have the same functionality with, if you pass in a NULL pointer as the first parameter of realloc()? 
8. What does sizeof(char*) return? What does sizeof(char) return?
9. Assume in some C code you have the following array:
   ```c
   int arr[] = {10, 20, 30, 40};
   ```
And you want to get the value of the third element (\texttt{int third = ?}). How would you do this without using the array brackets \texttt{[]}? (Hint: think about pointer manipulation.)

10. Give the line of C code to allocate an array of 12 shorts on the stack. Give the line of C code to allocate an array of 12 shorts on the heap.

**Extra Credit**

There are 5 points total for extra credit on this assignment.

- **Early turnin:** [2 Points] 48 hours before regular due date and time
  [1 Point] 24 hours before regular due date and time
  (it's one or the other, not both)

- **[3 Points Total, 0.5 for each nop]** Eliminating nops in the sample assembly file.

**Getting Started**

Copy over the following files from the public directory.

```
$ cp ~/../public/pa2EC.c ~/pa2
$ cp ~/../public/pa2NOPS.s ~/pa2
```

**Overview**

You will be removing up to six nops in \texttt{nops.s} to perform assembly optimization--do NOT modify any other files. Be sure to move the instruction you intend to fill the delay slot with into where the nop instruction was, do NOT just remove the nop. A reference executable will not be provided, so you should use the non-optimized version of \texttt{pa2NOPS.s} to test against your optimized version.

**Compiling**

You can compile the extra credit program using the following command.

```
$ gcc -o pa2EC pa2EC.c pa2NOPS.s
```

**Sample Output**

```
[cs30xyz@ieng9]:pa2$ ./pa2EC
 10  82  166  175  212  296  305  318  328  336
 350  366  372  374  438  452  481  489  489  569
 576  615  633  649  653  675  720  729  727  764
 771  771  851  858  862  918  960  1036  1098
1118 1120 1168 1216 1270 1282 1298 1306 1344 1375
1376 1400 1404 1440 1541 1568 1580 1683 1700 1714
1734 1739 1761 1782 1811 1817 1821 1826 1867
1936 1989 2030 2035 2067 2092 2136 2206 2218 2229
2303 2328 2330 2337 2352 2385 2395 2423 2460 2481
2495 2527 2568 2571 2574 2575 2594 2617 2643 2664
2667 2740 2745 2748 2795 2820 2824 2844 2845 2846
2850 2901 2934 2979 3015 3017 3043 3055 3058 3078
3127 3141 3157 3160 3172 3179 3183 3212 3247 3280
3314 3335 3335 3384 3420 3461 3464 3492 3550 3555
3556 3590 3590 3608 3653 3671 3685 3718 3741 3820
3825 3880 3921 3936 3985 4028 4034 4072 4088 4148
4231 4237 4258 4262 4263 4344 4394 4397 4440 4499
4543 4555 4570 4591 4595 4633 4662 4702 4724 4726
4727 4758 4783 4802 4804 4806 4834 4836 4877 4906
4963 4973 4996 5019 5053 5077 5095 5104 5126 5161
5166 5191 5204 5269 5286 5354 5371 5377 5412 5426
5431 5432 5449 5489 5493 5501 5520 5534 5564 5576
```
Milestone Turn-in Instructions
Milestone Turn-in - due Wednesday night, May 3 @ 11:59 pm [15 points of Correctness Section]
Before final and complete turnin of your assignment, you are required to turnin several modules for the
Milestone check. Each module must pass all of our unit tests in order to receive full credit.

Files required for the Milestone:
parseStartTime.c  setBCDClock.s  updateBCDValue.s  updateBCDClock.c

A working Makefile with all the appropriate targets and any required header files must be turned in as well. All
Makefile test cases for the milestone functions must compile successfully via the commands make test***.
You do not need to turn in your README with the milestone.

In order for your files to be graded for the Milestone Check, you must use the milestone specific turnin script.
$ cse30_pa2milestone_turnin

To verify your turn-in:
$ cse30verify pa2milestone

Result = 12304
Final Turn-in Instructions

Final Turn-in - due Wednesday night, May 10 @ 11:59 pm

Once you have checked your output, compiled, executed your code, finished your README file (see above), and double-checked your style, you are ready to turn it in. Use the following names *exactly* otherwise our Makefile will not find your files.

Files required for the Final Turn-in:

<table>
<thead>
<tr>
<th>chime.c</th>
<th>printChar.s</th>
<th>test.h</th>
<th>pa2.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>displayBCDClock.c</td>
<td>setBCDClock.s</td>
<td>testparseStartTime.c</td>
<td>pa2Strings.h</td>
</tr>
<tr>
<td>main.c</td>
<td>updateBCDValue.s</td>
<td>testsetBCDClock.c</td>
<td>pa2Globals.c</td>
</tr>
<tr>
<td>parseStartTime.c</td>
<td>withinLimits.s</td>
<td>testupdateBCDClock.c</td>
<td>Makefile</td>
</tr>
<tr>
<td>pa2gettimeofday.c</td>
<td></td>
<td>testupdateBCDValue.c</td>
<td>README</td>
</tr>
<tr>
<td>updateBCDClock.c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>usage.c</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extra Credit Files:

<table>
<thead>
<tr>
<th>pa2EC.c</th>
<th>pa2NOPS.s</th>
</tr>
</thead>
</table>

How to Turn in an Assignment

Before turning in, run `make clean` and then `make` to double check for any compiler errors/warnings. Then use the following turnin script to submit your full assignment before the due date as follows:

```
$ cse30turnin pa2
```

To verify your turn-in:

```
$ cse30verify pa2
```

Up until the due date, you can re-submit your assignment via the scripts above. Note, if you turned in the assignment early for extra credit and then turned it in again later (after the extra credit cutoff), you will no longer receive early turn-in credit.

Failure to follow the procedures outlined here will result in your assignment not being collected properly and will result in a loss of points. **Late assignments WILL NOT be accepted.**

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.