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 4 unsigned chars to store the tens and ones values of the clock's hours, minutes, seconds and a char ‘A’ or ‘P’ to indicate AM or PM. 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 binary 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 ARM assembly bit-wise operations, memory loads and stores (ldr/str), allocating local variables on the runtime stack and accessing them relative to the frame pointer (fp). 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).

IMPORTANT NOTE for Assembly routines:
1. Make sure you do not use registers other than r0-r3 as scratch registers in your assembly functions. Allocate local variables on the stack instead.
2. Only fp, lr are pushed to the stack.
3. Note that values in r0-r3 will not be preserved after function calls.

Grading

- README: 10 points - See README Requirements here and questions below
- 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
- 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.

NOTE: If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.
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 pi-cluster via ssh.

```bash
$ ssh cs30xyz@pi-cluster.ucsd.edu
```

Create and enter the pa2 working directory.

```bash
$ mkdir ~/pa2
$ cd ~/pa2
```

Copy the starter files from the public directory.

```bash
$ cp ~/../public/pa2StarterFiles/* ~/pa2/
```

Copy files needed from your pa1 directory.

```bash
$ cp ~/pa1/isInRange.s ~/pa2/
$ cp ~/pa1/isEven.s ~/pa2/
$ cp ~/pa1/myRemainder.s ~/pa2/
$ cp ~/pa1/printChar.s ~/pa2/
```

Starter files provided:

- pa2.h
- pa2Strings.h
- pa2Globals.c
- test.h
- testsetClock.c
- Makefile
- timeRightNow.c

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.

Example Input

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):

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

**NOTE:**

1. The output of your program **MUST** match exactly as it appears in the pa2test output. You need to pay attention to **everything** in the output, from the order of the error messages to the small things like extra newlines at the end (or beginning, or middle, or everywhere)!

2. We are not giving you any sample outputs, instead you are provided some example inputs. You are responsible for trying out all functionality of the program; the list of example inputs is not exhaustive or complete. It is important that you fully understand how the program works and you test your final solution thoroughly against the executable.
Detailed Overview

The function prototypes for the various C and Assembly functions are as follows.

C routines:

```c
int main( int argc, char * argv[] );
unsigned long interpretTime( struct tInfo * tmPtr, const char * time );
void printClock( const unsigned char clock[] ) ;
void tickClock( unsigned char clock[] ) ;
int timeRightNow( struct timeval * tp, void * p );
```

Assembly routines:

```c
void setClock( unsigned char clock[], const struct tInfo * tmPtr );
void convertTime( struct tInfo * ourTime, const struct tm * tmPtr );
unsigned char incrementClockValue( unsigned char BCDbits,
                                    const unsigned int maxValue );
long isInRange( long num, long min, long max );
long isEven( long value );
void printChar( char c );
long myRemainder( long dividend, long divisor );
```

For the Milestone, you will need to complete:

```
setClock.s  incrementClockValue.s  interpretTime.c  tickClock.c
```

Process Overview:

The following is an explanation of the main tasks of the assignment, and how the individual functions work together to form the whole program.

This program takes in 2 optional command line arguments to specify how the clock will be displayed and an optional flag to display a help message.

```
$ ./pa2 [numTicks [startTime]] | [--help]
```
Explanation of Command Line Arguments:

- **numTicks** - [optional] the number of seconds to tick the clock.
- **startTime** - [optional] starting time of the clock entered as HH:MM:SS (A|a|P|p). *If specifying start time, numTicks must also be specified as well. If startTime is not specified the start time will be the current time.
- **--help** - [optional] the help flag for displaying usage message intentionally

Printing out the clock consists of the following steps:

1. Parse command line arguments in `main()`, where `isInRange()` and `errno` will help with error checking.
   a. If there are errors, print appropriate error messages and usage message to `stderr`, then return `EXIT_FAILURE`.
   b. If there are no errors, proceed to print the clock.
2. If user inputs a start time, use `interpretTime()` to parse the specified time information into a struct.
3. If user did not input a start time, use `timeRightNow()` to get the current system time and `convertTime()` to populate our time struct with information from the system time struct.
4. After parsing the start time of the clock, use `setClock()` to set up the clock array to represent the specified value.
5. In a loop print the clock using `printClock()` and tick the clock using `tickClock()` which uses `incrementClockValue()` to add one second at a time to the clock.
6. After printing and ticking the clock (alternating) for user specified number of times or the default number of times, and exit the program by returning `EXIT_SUCCESS`. 
Milestone Functions to be Written

Listed below are the modules to be written for the milestone.

**setClock.s**
void setClock( unsigned char clock[], const struct tInfo * tmPtr );

This function will set the time in the `clock` array using the hours, minutes, and seconds in `tmPtr`. `tmPtr` is a struct `tInfo` that has the following members: `tm_sec`, `tm_min`, `tm_hour`, `am_pm`. (You can check `pa2.h` for the declaration). 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 the `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) AM, the following bit patterns should be stored in `clock`.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Binary</td>
<td>0001</td>
<td>0000</td>
<td>0011</td>
</tr>
</tbody>
</table>

Hint: You will need to use multiple loads and stores to access members in `tmPtr` and modify `clock`. Allocate space for local variables and parameters on the stack if you need to.

**incrementClockValue.s**
unsigned char incrementClockValue( 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`.

**interpretTime.c**
unsigned long interpretTime( struct tInfo * tInfoPtr, const char * time );

This function will parse the `time` string and populate the `tm_sec`, `tm_min`, `tm_hour` and `am_pm` members of the struct `tInfo` pointed to by `tInfoPtr`. 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 -s3 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).
  - **Note:** If the hour value in the input string is 0, it should be interpreted as 12.
- If `strtol()` did not encounter any errors, you also need to ensure that each time unit is within the appropriate range (use `isInRange()` with the constants defined in `pa2.h`). If a time unit is outside its range, set the appropriate error flag.
- After parsing the hour, min and sec values, start parsing the am/pm char following the sec value.
  - If the character is any non numeric char other than a/A/p/P, interpret as **ERR_TIME_FORMAT**.
  - If there are no more characters after HH:MM:SS, interpret as **ERR_TIME_FORMAT**.
  - If the string has the format of HH:MM:SS(a|A|p|P) followed by other chars, such as "12:15:10A3string" also interpret as **ERR_TIME_FORMAT**.

**Examples of errors:**

<table>
<thead>
<tr>
<th>time</th>
<th>flags set in the error bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>03:05:aA</td>
<td><strong>ERR_TIME_FORMAT</strong></td>
</tr>
<tr>
<td>1z0:16y20</td>
<td><strong>ERR_TIME_FORMAT</strong></td>
</tr>
<tr>
<td>13:03:05p</td>
<td><strong>ERR_TIME_RANGE</strong></td>
</tr>
<tr>
<td>12:03:05b</td>
<td><strong>ERR_TIME_FORMAT</strong></td>
</tr>
<tr>
<td>12:03:05a9</td>
<td><strong>ERR_TIME_FORMAT</strong></td>
</tr>
<tr>
<td>12:03:05</td>
<td><strong>ERR_TIME_FORMAT</strong></td>
</tr>
<tr>
<td>1g2:03:05</td>
<td><strong>ERR_TIME_NUM</strong>, <strong>ERR_TIME_FORMAT</strong></td>
</tr>
<tr>
<td>12a:66:05</td>
<td><strong>ERR_TIME_NUM</strong>, <strong>ERR_TIME_RANGE</strong>, <strong>ERR_TIME_FORMAT</strong></td>
</tr>
</tbody>
</table>

**Reasons for error:**

- **time** did not contain the correct number of colon 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 range.
- **time** did not have a valid am_pm char (a/A/p/P) immediately following a **valid** second value.
**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.

**tickClock.c**

```c
void tickClock( unsigned char clock[] );
```

This function will increment the clock by one second using `incrementClockValue()`. 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. Note that you may need to change the char of ‘A’ or ‘P’ to indicate AM or PM.

`incrementClockValue()` 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 `incrementClockValue()`.

For example:

<table>
<thead>
<tr>
<th>original clock</th>
<th>updated clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:20:30A</td>
<td>10:20:31A</td>
</tr>
<tr>
<td>05:19:59P</td>
<td>05:20:00P</td>
</tr>
<tr>
<td>09:59:59A</td>
<td>10:00:00A</td>
</tr>
<tr>
<td>11:59:59P</td>
<td>12:00:00A</td>
</tr>
</tbody>
</table>

---

**Post-Milestone Functions to be Written**

Listed below are the modules to be written after the milestone functions are complete.

**convertTime.s**

```c
void convertTime( struct tInfo * ourTime, const struct tm * tmPtr );
```

This function will use hours, minutes, and seconds in `tmPtr` to populate the `tm_sec`, `tm_min`, `tm_hour` and `am_pm` members of the `struct tInfo` pointed to by `ourTime`. `tmPtr` is a pointer to a `struct tm` that has the following members: `tm_sec`, `tm_min`, `tm_hour` (see `man -s3 ctime` for more information). Use the offsets defined in `pa2Globals.c` to access these members. Note that `struct tm` uses 0-23 to store hours. You will need to calculate `tm_hour` (range specified in `pa2.h`) and `am_pm` members of the `struct tInfo` properly.

**printClock.c**

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

This function displays the BCD clock based on the passed-in `clock` array. For each entry that represents a number 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”).

You will print out a border made up of AM/PM (indicated by the last entry in the array), as shown in the example below (isEven() may be helpful here):

**For example:** BCD clock that displays time 08:56:37A with input clock = {0x08, 0x56, 0x37, 0x65}:

<table>
<thead>
<tr>
<th>clock[0]</th>
<th>Upper nibble: 0 → 0000₂ → . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower nibble: 8 → 1000₂ → 0 . .</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>clock[1]</th>
<th>Upper nibble: 5 → 0101₂ → . O O</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>clock[2]</th>
<th>Upper nibble: 3 → 0011₂ → . . 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower nibble: 7 → 0111₂ → . O O</td>
</tr>
</tbody>
</table>

BCD clock that displays time 10:59:59P with input clock = {0x10, 0x59, 0x59, 0x80}:

<table>
<thead>
<tr>
<th>clock[0]</th>
<th>Upper nibble: 1 → 0001₂ → . . . O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower nibble: 0 → 0000₂ → . . . .</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>clock[1]</th>
<th>Upper nibble: 5 → 0101₂ → . O O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower nibble: 9 → 1001₂ → 0 . . O</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower nibble: 9 → 1001₂ → 0 . . O</td>
</tr>
</tbody>
</table>

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.

```
main.c
int main( int argc, char *argv[] );
```

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` of size `CLOCK_ARR_SIZE` on the stack. This array contains the hours, minutes, seconds, and AM/PM 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 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 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 -s3 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 ranges 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 `timeRightNow()` (provided with the starter files) and `localtime()` (see `man -s3 ctime`) to get the current time and then use `convertTime()` to get the system’s local time. (Hint: look up `struct timeval` and `struct tm`, and how you need to make use of these structures in function calls). Look at the function description for `timeRightNow()` below for more info.
   b. If `startTime` was specified as the second command line argument, populate the `struct tm` by calling `interpretTime()`. Check the return value of `interpretTime()` 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 error message using STR_TIME_FORMAT_ERR</td>
</tr>
<tr>
<td>ERR_TIME_NUM</td>
<td>Print error message using STR_TIME_PARSE_ERR</td>
</tr>
<tr>
<td>ERR_TIME_RANGE</td>
<td>Print error message using STR_TIME_RANGE_ERR</td>
</tr>
</tbody>
</table>

If any errors occurred during the argument parsing, print usage and return **EXIT_FAILURE** (after parsing all the arguments).

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 before printing.
- Set the BCD Clock using the struct tm pointer (that points to the populated struct tm structure) and your `setClock()` function.
- For each tick in numTicks, display the BCD clock and then update the clock after each tick. (Note that if numTicks is 7, we will print the clock a total of 8 times because we want to print the clock with the initial startTime before we begin the ticks—see public executable, and make sure your output matches exactly).

**Reasons for error:**

- Incorrect number of arguments
- Error parsing numTicks or numTicks wasn’t within range.
- startTime was not formatted correctly
- Error parsing values of startTime or values of startTime were not within range.

**Return Value:**  **EXIT_SUCCESS** on success; **EXIT_FAILURE** if any errors were encountered.

---

**timenow.c**

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

This function is provided for you. **Do not make any changes to this file.**

This routine calls the function `gettimeofday()` which will populate the struct timeval passed in. See `man -s3 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:**  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()`).

---

**isInRange.s**

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

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

**Reasons for error:**

- minRange is greater than or equal to maxRange → return -1

**Return Value:**  -1 if an error occurred. Otherwise, return 1 if value is in range and 0 if not in range.
**isEven.s**

```c
long isEven( long value );
```

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

**Return Value:** 0 if `value` is odd; 1 if `value` is even.

---

**myRemainder.s**

```c
long myRemainder( long dividend, long divisor );
```

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

**Return Value:** The remainder as a 32-bit signed integer.

---

**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, `testsetClock.c`. 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 test cases to all of the unit test files. You need to add as many unit tests as necessary to cover all possible use cases of each function. You will lose points if you don’t do this!** You are responsible for making sure you thoroughly test your functions. Make sure you think about boundary cases, special cases, general cases, extreme limits, error cases, etc. as appropriate for each function. The Makefile includes the rules for compiling these tests. Keep in mind that your unit tests will not build until all required files for that specific unit test have been written. These test files **will be collected with the Milestone**, so they must be complete before you submit your Milestone.

**Unit tests you need to complete:**
- `testsetClock.c`
- `testincrementClockValue.c`
- `testinterpretTime.c`
- `testtickClock.c`

**To compile:**

```
$ make testsetClock
```

**To run:**

```
$ ./testsetClock
```

(Replace “testsetClock” with the appropriate file names to compile and run the other unit tests)
# README Questions

1. How do you maintain your integrity even when you’re stressed, pressured, or tired?

2. 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 second element (int second = ?). How would you do this without using the array brackets []? (Hint: think about pointer manipulation)

3. [Vim] What is the command to search and replace all occurrences of ‘foo’ with ‘bar’? What is the command to search and replace all occurrences of ‘foo’ with ‘bar’ but ask for confirmation first, and what option should you type to confirm substitution of this match?

4. [Vim] What is the command to open another file called file.txt (in the same directory) in the current vim session by splitting the window into top and down? How about splitting into left and right? How do you move between files when in split mode?

5. [Vim] What is the command to quit without saving after you made some edits?

6. [Vim] What is the command to show tabs and what is the command to turn it off?

7. [Vim] What is the command to show line numbers?

8. [Unix] What is the command to show lines that contain “errno” in the file main.c? What is the command to show five lines above and below each line that contains “errno” in the file main.c?

9. [Unix] What is the command to change the permissions of the file file.txt such that no one can read, write or execute the file? What is the command to check for the file permissions of file.txt?

10. [Unix] What is the command to open a file file.txt in vim and go to a certain line number x? (This is a single command) [Vim] What is the command to go to a certain line number x in vim?

---

# Extra Credit

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

- **[1.5 Points]** chime.c -- print out Ding! Dong! at every hour boundary for hour number of times
- **[1.5 Points]** interpretTimeEC.c -- allows the user to enter the start time in 24 hour format
- **[2 Points]** max7.s and maxmain.s -- find the max of seven numbers and print it out

There are three components to this extra credit. Two parts (chime.c and interpretTimeEC.c) build upon your non-extra credit pa2 assignment, so do not begin this extra credit until you are completely certain that your pa2 assignment works perfectly.

First, copy your non-extra credit files to extra credit files using the following commands. Note: do not modify your non-extra credit files.

```bash
cs30xyz@pi-cluster-001:~/pa2$ cp main.c mainEC.c
(cs30xyz@pi-cluster-001:~/pa2$ cp interpretTime.c interpretTimeEC.c
```

You will be making two additions to this program. First, adding in the chime ability, described in the function description below. A small change/addition will need to be made to mainEC.c to accommodate for this feature. The second is to allow users to input the start time in 24 hour format. interpretTimeEC.c will adapt for this, converting that time into 12 hour time as stored in our struct tInfo.

The last component is a translation exercise from C to ARM assembly as described below.
Compiling:
You can compile the BCD Clock extra credit using the following command.
cs30xyz@pi-cluster-001:~/pa2$ make pa2EC

Sample Executable:
cs30xyz@pi-cluster-001:~/pa2$ ~/.public/pa2ECtest

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!”).

For example:

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

interpretTimeEC.c
unsigned long interpretTime( struct tInfo * tInfoPtr, const char * time );

This function retains much of the same functionality as the non-extra credit version, but allows for time to be entered in 24 hour format.

When parsing the hour, you should change the max value to be 23 (as defined in pa2.h). After checking to see if the hour is in range, you need to convert it to 12 hour time. So, if the user enters hour 13, you should set the hour to be 1 and am_pm to denote pm. Note that the user is no longer able to enter time in 12 hour format, so the conditions to check for ‘a’ or ‘p’ after seconds should be removed. See the extra credit usage statement for more information on the time format.

For example:

<table>
<thead>
<tr>
<th>User input</th>
<th>12 hour time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:25:32</td>
<td>12:25:32 AM</td>
</tr>
<tr>
<td>13:20:56</td>
<td>1:20:56 PM</td>
</tr>
<tr>
<td>12:15:20</td>
<td>12:15:20 PM</td>
</tr>
</tbody>
</table>

Return Value: Same as non extra credit version.
Max7 Extra Credit

This extra credit will give you some practice with allocating local variables on the stack and calling functions with more than four arguments in ARM. Your task is to translate the two provided C functions into assembly. This program does not take user input and only has one output as shown below; your program must match this output exactly.

```
Max of 1 2 3 4 5 6 7 is 7
```

You will have to write ARM functions to replicate the two C files (maxmain.c and max7.c) that are provided below. You will write one file maxmain.s to replicate the functionality of the main function and another file max7.s to replicate the functionality of the max7 function.

You must also allocate space and pass additional arguments beyond the first 4 (r0-r3) on the stack, with arg5 at a lower address than arg6 which is at a lower address than arg7. You will access these additional formal parameters via positive offsets from the fp in the called function. Deallocate these additional args after the function call returns.

```
No magic numbers!
```

main() is responsible for setting up the local variables, calling max7(), then setting up and executing the call to printf() to output the max number. max7() needs to take its seven parameters and return the max of the numbers.
maxmain.c
1 #include <stdio.h>
2
3 int main( void ) {
4
5   int a = 1;
6   int b = 2;
7   int c = 3;
8   int d = 4;
9   int e = 5;
10  int f = 6;
11  int g = 7;
12
13  int max;
14
15  max = max7( a, b, c, d, e, f, g );
16
17  printf( "Max of %d %d %d %d %d %d %d is %d\n", a, b, c, d, e, f, g, max );
18
19  return 0;
20
}

max7.c
1 int max7( int a, int b, int c, int d, int e, int f, int g ) {
2   int max = a;
3
4   if(b > max) max = b;
5   if(c > max) max = c;
6   if(d > max) max = d;
7   if(e > max) max = e;
8   if(f > max) max = f;
9   if(g > max) max = g;
10
11  return max;
12 }

Compilation:
To compile, use this command:

    gcc -g -o pa2max7 maxmain.s max7.s

This command compiles our code using gcc, the GNU Compiler collection. The -g option tells gcc to produce debug information for our program that allows us to debug it using gdb. The -o option is used to specify what we want our compiled executable file to be called. In this case, by using “-o pa2max7” we are telling gcc to call the executable pa2max7. We are supplying maxmain.s and max7.s to gcc as source files to be compiled. In order for compilation to succeed, one of our sources must contain a main function. In this case, that file is maxmain.s.
**Turnin Summary**

See the turnin instructions [here](#). Your file names must match the below *exactly* otherwise our Makefile will not find your files.

**Milestone Turnin:**
Due: Wednesday night, May 2 @ 11:59 pm

**Files required for the Milestone:**
- setClock.s
- incrementClockValue.s
- interpretTime.c
- tickClock.c
- testsetClock.c
- testincrementClockValue.c
- testinterpretTime.c
- testtickClock.c

**Final Turnin:**
Due: Wednesday night, May 9 @ 11:59 pm

**Files required for the Final Turn-in:**
- setClock.s
- interpretTime.c
- pa2.h
- incrementClockValue.s
- printClock.c
- pa2Strings.h
- convertTime.s
- tickClock.c
- pa2Globals.c
- isInRange.s
- timeRightNow.c
- test.h
- isEven.s
- main.c
- Makefile
- myRemainder.s
- testsetClock.c
- README
- printChar.s
- testincrementClockValue.c
- testinterpretTime.c
- testtickClock.c

**Extra Credit Files:**
- chime.c
- maxmain.s
- interpretTimeEC.c
- max7.s
- mainEC.c

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.