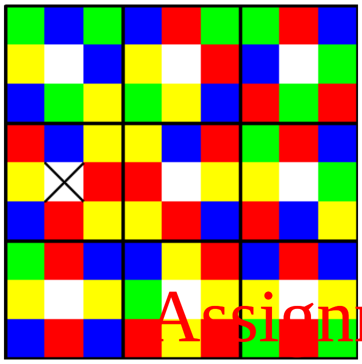
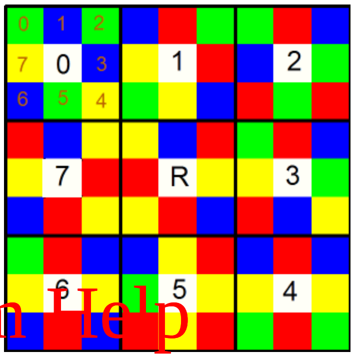
This project addresses efficient manipulation of data structures and pattern matching. The task is a classic IQ test where one is asked to match an eight color pattern to a copy that may be flipped (mirrored) and rotated. The reference puzzle is the square in the center of the board. The eight candidate puzzles wrap around the edge of the board clockwise beginning in the upper left hand corner. The color codes are as follows (0 = red, 1 = yellow, 2 = green, and 3 = blue). The reference pattern in the example below (the center 3 x 3 pattern) is yellow, blue, red, yellow, blue, red, yellow, red. The matched candidate is the pattern mark with an “X” to the left of the reference (pattern 7). Note that it is flipped and rotated. Exactly one of the eight candidates will match the reference pattern; it may be flipped (horizontally or vertically), rotated, or both. The reference and candidate color codes are each packed into a 16-bit unsigned integer.



```
Reference    [1, 3, 0, 1, 3, 0, 1, 0]


Candidates [[2, 3, 2, 3, 1, 2, 3, 1],
             [3, 0, 2, 0, 3, 1, 2, 1],
             [2, 0, 3, 2, 0, 2, 0, 3],
             [2, 0, 3, 2, 1, 3, 0, 1],
             [3, 0, 3, 1, 2, 0, 2, 1],
             [3, 1, 0, 1, 0, 1, 0, 2],
             [2, 0, 3, 1, 3, 0, 3, 1],
             [0, 3, 1, 0, 1, 0, 3, 1]]
```



packed reference and candidate values

reference	cand. 0	cand. 1	cand. 2	cand. 3	cand. 4	cand. 5	cand. 6	cand. 7
4941	31214	26403	51378	19890	25203	33863	29554	28956

Strategy: Unlike many “function only” programming tasks where a solution can be quickly envisioned and implemented, this task requires a different strategy.

1. Before writing any code, reflect on the task requirements and constraints. *Mentally* explore different approaches and algorithms, considering their potential performance and costs. The metrics of merit are **static code length**, **dynamic execution time**, and **storage requirements**. There are often trade-offs between these parameters.
2. Once a promising approach is chosen, a high level language (HLL) implementation (e.g., in C) can deepen its understanding. The HLL implementation is more flexible and convenient for exploring the solution space and should be written before constructing the assembly version where design changes are more costly and difficult to make. For P1-1, you will write a C implementation of the program.
3. Once a working C version is created, it’s time to “be the compiler” and see how it translates to MIPS assembly. This is an opportunity to see how HLL constructs are supported on a machine platform (at the ISA level). This level requires the greatest programming effort; but it also uncovers many new opportunities to increase performance and efficiency. You will write the assembly version for P1-2.

P1-1 High Level Language Implementation:

In this part, the first two steps described above are completed. It's fine to start with a simple implementation that produces an answer; this will help deepen your understanding. Then experiment with your best ideas for a better performing solution. Each hour spent exploring here will cut many hours from the assembly level coding exercise.

You should use the simple shell C program that is provided `P1-1-shell.c` to allow you to read in a puzzle. Rename the shell file to `P1-1.c` and modify it by adding your code to locate the reference pattern in the grid.

Since building example puzzles is complex, it is best to fire up Misasim, generate a puzzle, step forward until the puzzle is written in memory, and then dump memory to a file. *Your C program should print the position of the matching pattern (an integer between 0 and 7, inclusive)*. For example, for the puzzle shown above, it should print 7 using the print statement given in the shell. (A few test cases are provided; their answers are given in their filenames.)

The shell C program includes code that reads the puzzle data in from an input file. It also includes an important print statement **used for grading** (please don't change it). If you would like to add more print statements as you debug your code, please wrap them in an if statement using a `DEBUG` flag – an example is given in the shell program – so that you can suppress printing them in the code you submit by setting `DEBUG` to 0. *If your submitted code prints extraneous output, it will be marked incorrect by the autograder.*

You can modify any part of this program. Just be sure that your completed assignment can read in an arbitrary puzzle and correctly print the solution since this is how you will receive points for this part of the project.

Note: you will not be graded for your C implementation's performance. Only its accuracy and "good programming style" will be considered (e.g., organizing and commenting your code). Your MIPS implementation might not even use the same algorithm; although it's much easier for you if it does.

When you have completed the assignment, submit the single file `P1-1.c` to Canvas. You do not need to submit data files. Although it is good practice to employ a header file (e.g., `P1-1.h`) for declarations, external variables, etc., in this project you should just include this information at the beginning of your submitted program file. In order for your solution to be properly received and graded, there are a few requirements.

1. The file must be named `P1-1.c`. (Do not worry if canvas appends a version number.)
2. Your name and the date should be included in the header comment.
3. Your submitted file should compile and execute on an arbitrary puzzle (produced from Misasim). It should contain the unmodified print statement, giving the solution.
4. Your program must compile and run with gcc under Linux. Compiler warnings will cause point deductions. If your program does not compile or enters an infinite loop, it will earn 0 correctness points.

Your solution must be properly uploaded to Canvas before the scheduled due date. The canvas P1-1 assignment has details on late penalties and policies.

P1-2 Assembly Level Implementation: In this part of the project, you will write the performance-focused assembly program that solves the Match Puzzle. A shell program (`P1-2-shell.asm`) is provided to get you started. Rename it to `P1-2.asm`.

You will call a software interrupt (582) that will generate a random Match Puzzle board. *Your solution must not change the puzzle board (do not write over the memory containing the puzzle).*

Your MIPS assembly code must compute the number of the pattern matching the reference and report it by calling another software interrupt (583).

Library Routines: There are two library routines (accessible via the `swi` instruction).

SWI 582: **Create Board**: This routine creates a puzzle board, storing the packed reference pattern in the memory address contained in $1 and storing the packed candidates in the eight words in memory beginning at the word immediately following the reference pattern. (In the shell code, the reference pattern is stored at the address labeled `Reference` and the eight candidates are stored in the eight words beginning at the address labeled `Candidates`.)

> INPUTS: $1 should contain the address of the word allocated for the reference pattern in memory.
>
> OUTPUTS: memory starting at the address in $1 contains the packed reference pattern, followed by the eight packed candidates.

SWI 583: **Highlight Candidate**: This routine highlights the selected candidate referenced by the offset stored in $3 (0, 4, 8, 12, 16, 20, 24, or 28). The selected candidate is highlighted in the display and marked for grading.

> INPUTS: $3 should contain the matching *candidate offset as a multiple of 4* (not base address).
>
> OUTPUTS: $6 contains the correct answer which you can check against what you reported in $3 to test and validate your program.

Evaluation: In this part (P1-2), correct operation and efficient performance are both important. The assessment of your submission will include functional accuracy during 100 trials and performance and efficiency. The code size, dynamic execution length, and operand storage requirements are scored empirically, relative to a baseline solution. The baseline numbers for this project are **static code size: 30 instructions, dynamic instruction length: 180 instructions (avg.), storage required: 6 words** (not including the reference word and 8 candidate words in memory and dedicated registers $0, $31). *The dynamic instruction length metric is the maximum of the baseline metric and the average dynamic instruction length of the five fastest student submissions.*

Your score will be determined through the following equation:

$$PercentCredit = 2 - \frac{Metric_{Your\ Program}}{Metric_{Baseline\ Program}}$$

Percent Credit is then used to determine the number of points for the corresponding points category. Important note: while the total score for each part can exceed 100%, especially bad performance can earn *negative credit*. The sum of the combined performance metrics scores (code size, execution length, and storage) will not be less than zero points. **Finally, the performance scores will be reduced by 10% for each incorrect trial (out of 100 trials). You cannot earn performance credit if your implementation fails ten or more of the 100 trials**.

In MIPS assembly language, small changes to the implementation can have a large impact on overall execution performance. Often trade-offs arise between static code size, dynamic execution length, and operand storage requirements. Creative approaches and a thorough understanding of the algorithm and programming mechanisms will often yield impressive results. Almost all engineering problems require multidimensional evaluation of alternative design approaches. Knowledge and creativity are critical to effective problem solving.

In order for your solution to be properly received and graded, there are a few requirements.

1. The file must be named `P1-2.asm`.
2. Your program must correctly highlight the matching candidate to the reference pattern using swi 583.
3. Your program must return to the operating system via the **`jr $31`** instruction. *Programs that include infinite loops or produce simulator warnings or errors will receive zero credit.*
4. Your solution must be properly uploaded to Canvas before the scheduled due date.

Project Grading: The project grade will be determined as follows:

part	*description*	*percent*
P1-1	**Functional C program**	25
P1-2	**Assembly program**	
	correct operation, proper commenting and style	25
	static code size	15
	dynamic execution length	25
	operand storage requirements (# registers)	10
	total	100

Honor Policy: In all programming assignments, you should design, implement, and test your own code. Any submitted assignment containing non-shell code that is not fully created and debugged by the student constitutes academic misconduct. You should not share code, debug code, or discuss its performance with anyone. Once you begin implementing your solution, you must work alone.

Good luck and happy coding!