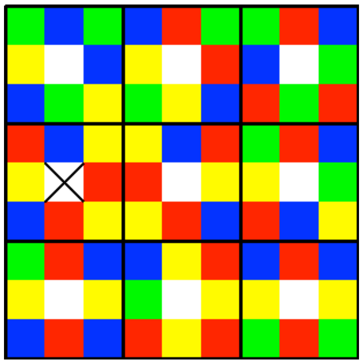
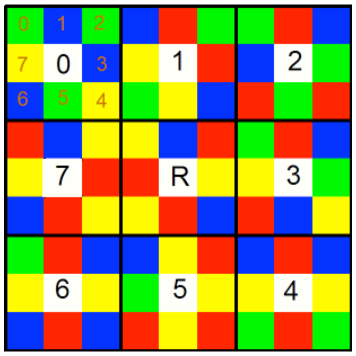
ECE 2035 Project One: Match Puzzle

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 the lower 16-bits of an unsigned integer (2 bits per element). The Reference and Candidates arrays shown below show the 2 bit values of each element, and the table shows the values of the corresponding packed unsigned int.



```
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

P1-1: In this part, you must write a C program to locate the matching puzzle in the grid. Use the shell code `P1-1-shell.c` as a starting point. Two testfiles (test1.txt and test3.txt) containing sample puzzles are included. You may create additional puzzles using the dump command in Misasim.

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`.
2. The program should report the answer using the following print statement:
   ```
   printf("The matching pattern is at position [0-7] %d\n", Position);
   ```
3. Your solution must be properly uploaded to Canvas before the scheduled due date.

P1-2: In this part, a performance implementation of the Match Puzzle program will be implemented in MIPS assembly language. Use the shell code `P1-2-shell.asm` as a starting point.

In this version, correct operation and efficient performance are evaluated. 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**.

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.

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.
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.

Implementation Evaluation:

In this project, the functional implementation of **P1-1** will be evaluated on whether the correct answer is computed. Although proper coding technique (e.g., using the proper data types, operations, control mechanisms, etc.) will be evaluated, parametric performance will not be considered.

The performance implementation of the **P1-2** program in MIPS assembly will be evaluated both in terms of correctness and performance. The correctness evaluation employs the same criterion described for the functional implementation. The performance criteria include static code size (# of instructions), dynamic instruction length (# executed instructions), and storage requirements (total number of words in registers or memory, including stack memory). All of these metrics are used to determine the quality of the overall implementation.

Once a candidate algorithm is selected, an implementation is created, debugged, tested, and tuned. 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.

One final note on design strategy: while optimizing MIPS assembly language might, at times, seem like a job best left to automated processes (i.e., compilers), it underscores a key element of engineering. Almost all engineering problems require multidimensional evaluation of alternative design approaches. Knowledge and creativity are critical to effective problem solving.

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 technique 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!