Top of Pile: This project explores a programming problem that arises in many different robotics applications. In a manufacturing setting, a robotic arm may need to pick up parts from a pile and use them to assemble something such as an automobile. To plan the robotic arm's movements, an overhead camera may be positioned over a pile of parts. In this assignment, we will focus on determining which color part is on top of the pile. (This may be important in finding a particular component and uncovering it so it can be used.) We would like this task to be performed in real-time, so the computational and storage requirements must be kept to a minimum. We are also concerned with the functional correctness of the algorithms used (i.e., getting the correct answer), since picking the wrong top part could lead to extra work for the robot or worse a malfunction if the pieces get tangled.

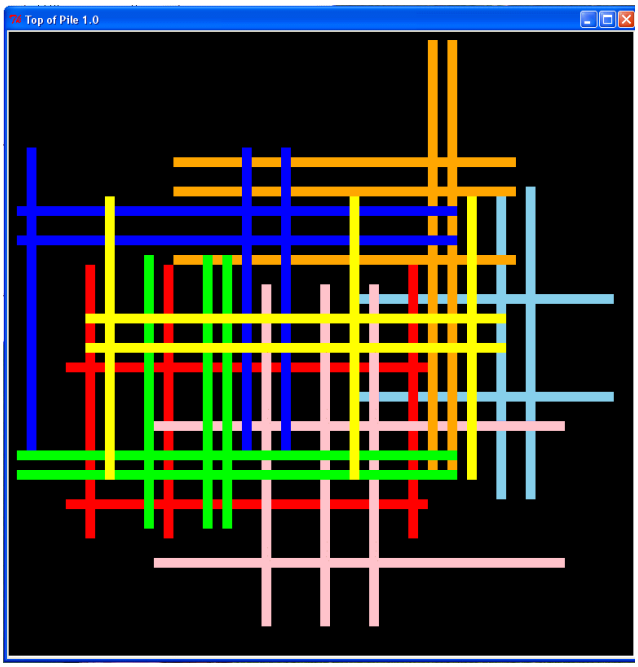

Figure 1. Pile of Parts (test case: pile1-ans6.txt)

Description: Consider a pile composed of seven parts viewed from above on a 64x64 image array of cells; each cell is one of eight colors (black=0, pink=1, red=2, green=3, blue=4, orange=5, yellow=6, skyblue=7). Each part has a unique color. There will be exactly one part that is on top of the pile. All other parts will be overlapping at least one part. Figure 1 shows an example pile image. In this example, the yellow part is on top. The answer is 6: the color yellow.

Figure 2: Color Palette

Each part is composed of horizontal and vertical lines all of the same color. Not all parts have the same number of horizontal or vertical lines. All horizontal lines in a part have the same length which ranges from 25 to 45 pixels. Similarly, all vertical lines in a part have the same length in the same range. A part's horizontal line length is not necessarily the same as its vertical line length. No parallel lines are adjacent to any other lines in the same part or in other parts. Each row (column) has at most one horizontal (vertical) line. No horizontal lines occlude (over-

lap) any other horizontal lines (i.e., no part will overlap another part along the same horizontal line). Similarly, no vertical lines occlude any other vertical lines. If a row contains a horizontal line, the row immediately above and the row immediately below that row will not contain any horizontal lines. Similarly, the columns immediately to the right and left of a column that contains a vertical line will not contain any vertical lines. Lines only touch at orthogonal intersections. No lines touch the image boundaries.

Hint: overlap can be determined by checking local constraints on a pixel and its immediate neighbors. Watch out for ambiguous "T" intersections in which you cannot tell for certain if the parts abut or overlap (e.g., in Figure 1, the skyblue part's leftmost vertical line intersects with the orange part's second horizontal line at a "T"). It will never be the case that the only point of overlap between two parts forms an ambiguous "T" intersection (e.g., the orange and skyblue parts intersect at other points that make it clear that orange is above skyblue).

The image array is provided as input to the program as a linearized array of the cells in row-column order. The first element of the array represents the color of the first cell in the first row. This is followed by the second cell in that row, etc. The last cell of the first row is followed by the first cell of the second row. This way of linearizing a two dimensional array is called *row-column mapping*. The color code (0-7) is packed in an unsigned byte integer for each cell.

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. Sometimes *back of the envelope* calculations (e.g., how many comparisons will be performed) can help illuminate the potential of an approach.

2. Once a promising approach is at hand, 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 Top of Pile 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 section, the first two steps described above will be 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 mentally-formed ideas for a better performing solution. Use any instrumentation techniques that help to assess how much work is being done. 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 linearized image array. The shell program includes a reader function `Load_Mem()` that loads the values from a text file. Rename the shell file to `P1-1.c` and modify it by adding your code.

Since generating example pile images is complex, it is best to fire up Misasim, generate a pile, step forward until the pile is written in memory, and then dump memory to a file. Your C program should print a single integer value as the answer, which is the color code of the part on top of the pile (e.g., it should print "4" if the top part is blue.) Some test cases are provided with the file naming convention: pile*i*-ans*n*.txt, where *i* is the test case number and *n* is the answer (the color code). For example, pile1-ans6.txt shown in Figure 1 is test case number 1 whose answer is color 6 (yellow).

Be sure that your completed assignment can read in an arbitrary pile, find the color of the part that is on top, and correctly print the color code 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 (speed and storage efficiency). Only its accuracy and good programming style will be considered. Your MIPS implementation may not even use the same algorithm; although it's much easier for you if it does.

When 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`.
2. Your submitted file should compile and execute on an arbitrary pile (produced from Misasim). It should contain the following print statement (unchanged)

   "`printf("The topmost part color is: %d\n", TopColor);`" to print out a single integer which is the color code of the topmost part on the pile. The command line parameters should not be altered from those used in the shell program. Your program must compile and run with gcc under Linux.
3. Your solution must be properly uploaded to Canvas before the scheduled due date.

P1-2 Assembly Level Implementation: In this part of the project, you will write the performance-focused assembly program that solves the top of pile puzzle. A shell program (`P1-2-shell.asm`) is provided to get you started. *Your solution must not change the pile image array (do not write over the memory containing the pile).*

In this version, execution performance and cost is often equally 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: **36** instructions, dynamic instruction length: **9865** instructions (avg.), total storage required for registers, static memory, and stack memory: **13** words (not including dedicated registers $0, $31, and not including the 1024 words for the input puzzle array). *As a safety net, 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 scale the number of points for the corresponding points category. For example, if your program uses half as much storage as the baseline, then PercentCredit is 1.5 and the number of points for the storage category (see Project Grading table below) is 10 scaled by 1.5 = 10*1.5 = 15. 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 (static code size, dynamic execution length, and storage) will be capped at zero; the sum of that portion of the grade will not be less than zero points. **Finally, these performance scores will be reduced by 10% for each incorrect trial (out of 100 trials). You cannot earn points for performance metrics if your implementation fails ten or more of the 100 trials**.

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

SWI 545: Create Pile: This routine initializes memory beginning at the specified base address (e.g., `Array`). It sets each byte of the 4096 byte array to the corresponding cell's color code in the 64x64 pile array. The color codes are: black=0, pink=1, red=2, green=3, blue=4, orange=5, yellow=6, skyblue=7. INPUTS: $1 should contain the base address of the 1024 word (4096 byte) array already allocated in memory. OUTPUTS: none.

As a debugging feature, you can load in a previously dumped pile testcase by putting -1 into register $2 before you call swi 545. This will tell swi 545 to prompt for an input txt file that contains a pile.

SWI 546: Top Color: This routine allows you to specify the color of the topmost part in the pile. INPUTS: $2 should contain a number between 1 and 7, inclusive. This answer is used by an automatic grader to check the correctness of your code. OUTPUTS: $3 gives the correct answer. You can use this to validate your answer during testing. If you call swi 546 more than once in your code, only the first answer that you provide will be recorded.

SWI 547: Highlight Position: This routine allows you to specify an offset into the Pile array and it draws a white outline around the cell at that offset. Cells that have been highlighted previously in the trace are drawn with a gray outline to allow you to visualize which positions your code has visited. INPUTS: $2 should contain an offset into the Pile array. OUTPUTS: none. *This is intended to help you debug your code; be sure to remove calls to this software interrupt before you hand in your final submission, since it will contribute to your instruction count.*

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 produce and store the correct value in $2, it must call SWI 546 to report your answer, and then return to the operating system via the `jr` instruction. *Programs that include infinite loops or produce simulator warnings or errors will receive zero credit*.
3. Your solution must be properly uploaded to Canvas before the scheduled due date.

Implementation Evaluation:

In this project, the functional implementation of top of pile in C (**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 top of pile program in MIPS assembly (**P1-2**) 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 (in this example, number of registers used). 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 tradeoffs 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	Top of Pile (C code)	25
P1-2	Top of Pile (MIPS assembly)	
	correct operation, proper commenting & style	25
	static code size	15
	dynamic execution length	25
	operand storage requirements	10
	Total	*100*

All code (MIPS and C) must be documented for full credit.

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!