This project focuses on an image search problem: finding a specific face in an image or photograph. This is a problem that arises in many video surveillance applications (e.g., detecting a well-known terrorist's face in a crowd at an airline terminal) as well as in searching large collections of digital images (e.g., finding all photos you took that contain your grandmother). In this assignment, we will focus on finding George P. Burdell, whose mugshots are shown in Figure 1, in an image that contains a crowd of faces, such as the sample scene in Figure 2. The faces in the scene, including George's, may be at varying distances from the camera, so they may appear scaled. For example in Figure 3a, George is scaled two times the original size and in Figure 3b, he is scaled by a factor of three. All faces will be facing forward and upright (not rotated).

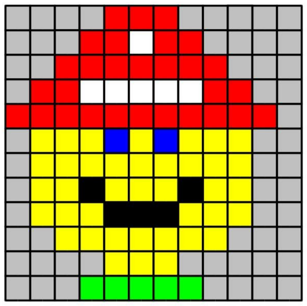


Figure 1: Mugshot of George P. Burdell
Disclaimer: Any resemblance to actual persons, roommates, or relatives is unintentional and coincidental.

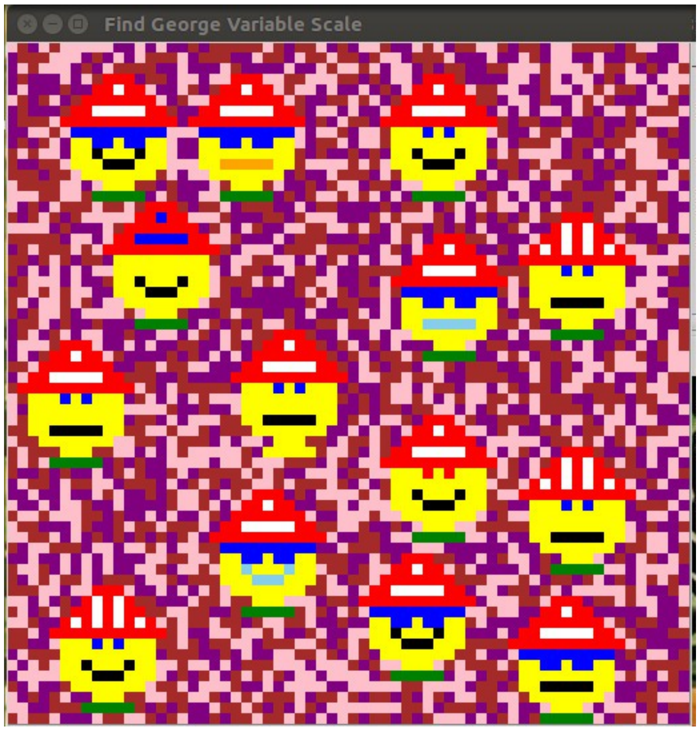


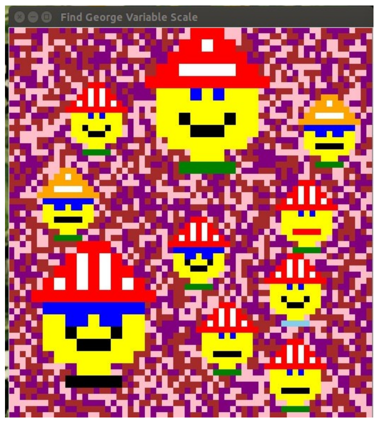
Figure 2: Sample Crowd of Faces
`(crowd-227-942.txt)`

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 functional correctness of the algorithm (i.e., getting the correct answer), since we do not want to generate frequent false alarms, or worse, miss an important match!

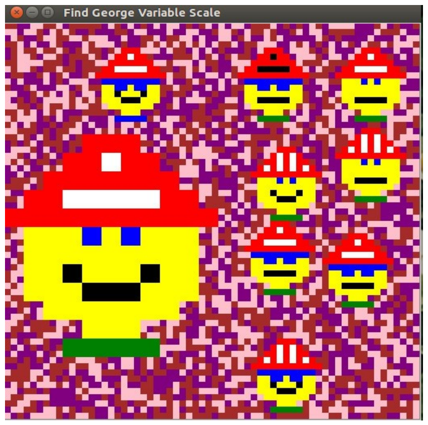
George will appear exactly once (possibly scaled) in a random spot in each crowded scene. None of the faces in the scene will overlap with each other and all faces fit completely within the boundaries of the crowd (no partial faces hanging off the edges).

Like most college students, George has a limited wardrobe; he always wears a red hat and green shirt – and he's always smiling. The faces that are not George will differ from him by having different color features (e.g., an orange hat or green skin) and/or structurally (possible variations: direction of hat stripes, glasses/none, smile/frown).

Your task is to find the one face that exactly matches (possibly with scaling) George's mugshot shown in Figure 1. For example, in Figure 2, George is the face at the top and to the right.



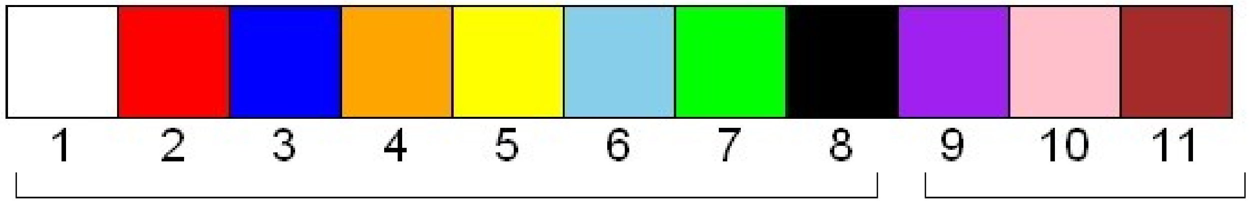
a) crowd-24-1519.txt



b) crowd-1152-3427.txt

Figure 3: More Sample Crowds

The crowd scene is a 64x64 image array of pixels; each pixel is one of eleven colors whose codes are given in the color palette shown in Figure 4. Each *non-scaled* face fits in an 12x12 region of pixels. Each face may appear in the image scaled by a factor of 2, 3, 4, or 5. For example, in Figure 3a, George is scaled by a factor of 2 and fits in a 24x24 square region; in Figure 3b, he is scaled by 3 and fits in a 36x36 region. The features of the face will always have color codes in the range 1, 2, …, 8. The background (non-face parts) of the image will always use colors whose codes are 9, 10, or 11.



Face Colors Background Colors

Figure 4: Color Palette

The image array is provided as input to the program as a linearized array of the pixels in row-column order. The first element of the array (location 0) represents the color of the first pixel in the first row. This is followed by the second pixel (location 1) in that row, etc. The last pixel of the first row (location 63) is followed by the first pixel of the second row (location 64). This way of linearizing a two dimensional array is called *row-column mapping*. The color code (1-11) is packed in an unsigned byte integer for each pixel.

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 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 Find George 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 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 image search code. Since generating example crowd images is complex, it is best to fire up Misasim, generate a crowd, step forward until the crowd is written in memory, and then dump memory to a file. Your C program should print the index of the top left corner and the index of the bottom right corner of the *square* region (bounding box) containing George. For the crowd shown in Figure 5 (below), it should print these locations as 1234 (the index of the top left pixel 18 pixels across and 19 rows down) and 2729 (the index of the bottom right pixel 41 pixels across and 42 rows down). This test case is provided as `crowd-1234-2729.txt`. (Note that the bounding box is square, so the *n* rightmost columns will contain only background pixels, where *n* is the scaling factor.)

Be sure that your completed assignment can read in an arbitrary crowd, find George, and correctly print his location (corners of his bounding box) 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., using proper data types, operations, control mechanisms, etc., and documenting your code with comments). Your C implementation does not need to use the same algorithm as the assembly program; however, it's much easier for you if it does.

When you have completed the assignment, submit the single file `P1-1.c` to the class T-square website. 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 please 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 crowd (produced from Misasim). It should contain the following print statement (unchanged):

   "`printf("George is located at: top left pixel %4d, bottom right pixel %4d.\n", TopLeft, BottomRight);`". 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 the class T-square site before the scheduled due date, **5:00pm on Wednesday, 4 October 2017. There will be a one hour grace period, after which no submitted material will be accepted.**

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

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: **???** instructions, dynamic instruction length: **???** instructions (avg.), storage required: **???** words (not including dedicated registers $0, $31, nor the 1024 words for the input puzzle array). *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:

$$\text{PercentCredit} = 2 - \frac{\text{Metric}_{\text{YourProgram}}}{\text{Metric}_{\text{BaselineProgram}}}$$

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 three library routines (accessible via the `swi` instruction).

SWI 592: Create Scaled Crowd: 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 pixel's color code in the 64x64 crowd image array. The color codes are defined in the palette in Figure 3. INPUTS: $1 should contain the base address of the 1024 word (4096 byte) array already allocated in memory. OUTPUTS: none.

SWI 552: Highlight Position: This routine allows you to specify an offset into the crowd array and it draws a white outline around the pixel at that offset. pixels that have been highlighted previously in the trace are drawn with a gray outline to allow you to visually trace which positions your code has visited. INPUTS: $2 should contain an offset into the Crowd array (an integer in the range 0 to 4095, inclusive). OUTPUTS: none. *This is intended to help you debug your code; be sure to remove calls to this software interrupt before handing in your final submission, since it will contribute to your instruction count.*

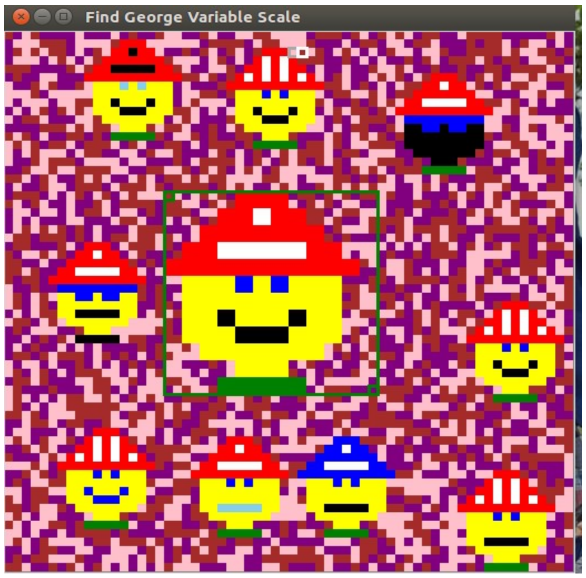


Figure 5: `Crowd` (`crowd-1234-2729.txt`) with Correct Answer:

`George is located at: top left pixel 1234, bottom right pixel 2729.`

SWI 593: Locate Scaled George: This routine allows you to specify the position of the top left and bottom right corner pixels of George's *square* bounding box to indicate the location where George has been found.

INPUTS: $2 should contain two packed numbers: in the upper 16 bits, the top left corner pixel location and in the lower 16 bits, the bottom right corner pixel location. Each location should be a number between 0 and 4095, 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 593 more than once in your code, only the first answer that you provide will be recorded. The visualization will draw a yellow box around the location you provide and a larger yellow box spanning the corner locations you provide. It will also draw a green box at the location of the correct answer and a larger green box showing George's actual location. If your answer is correct, the green boxes should completely cover your yellow boxes. For example, in Figure 5, the correct answer is `0x4D20AA9` (top left: 1234 = `0x4D2`; bottom right: 2729 = `0xAA9`) and when swi 593 is called with `0x4D20AA9` in $2, the green boxes appear as shown in Figure 5.

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 values in memory in the statically allocated memory.

3. Your program must call SWI 593 to report an answer and return to the operating system via the `jr` instruction. *Programs that include infinite loops or produce simulator warnings or errors will receive zero credit*.

4. Your solution must be properly uploaded to the submission site before the scheduled due date, **5:00pm on Friday, 13 October 2017**. **There will be a one hour grace period, after which no submitted material will be accepted.**

Implementation Evaluation:

In this project, the functional implementation in C (**P1-1**) will be evaluated based 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, performance will not be considered.

The performance implementation of the 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*	*due date*	*percent*
P1-1	Find George (C code)	Wed., 4 Oct. 2015	25
P1-2	Find George (MIPS assembly)	Fri., 13 Oct. 2015	
	correct operation, technique & 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.

You should design, implement, and test your own code. Any submitted project containing code not fully created and debugged by the student constitutes academic misconduct.