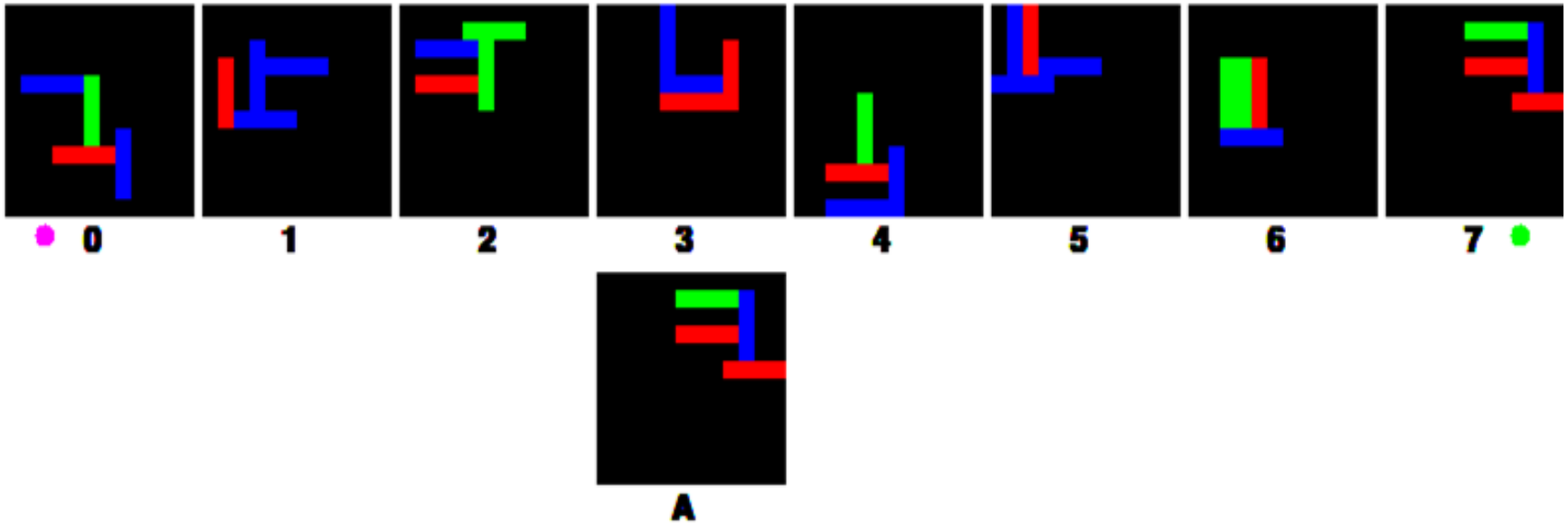
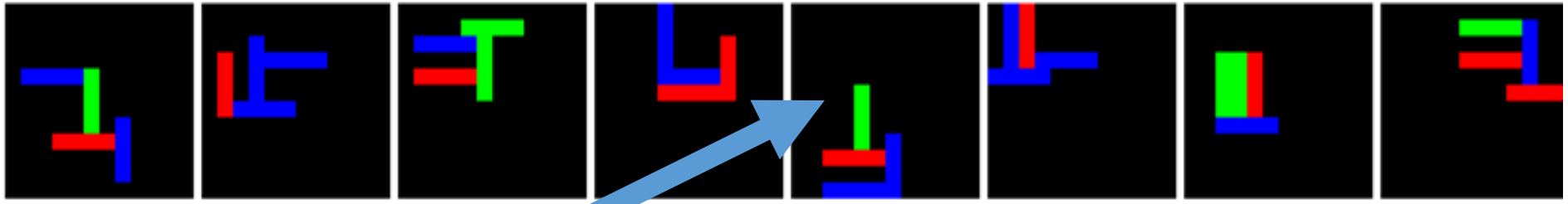
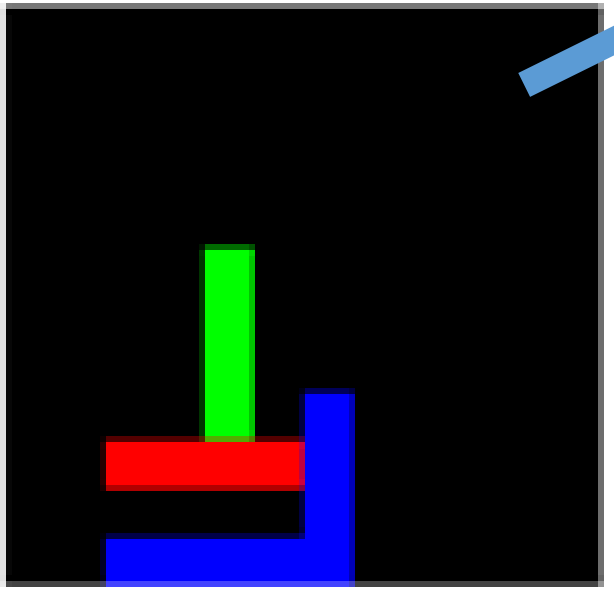


Project 1: Icon Matching

Exact Match Detection Puzzle

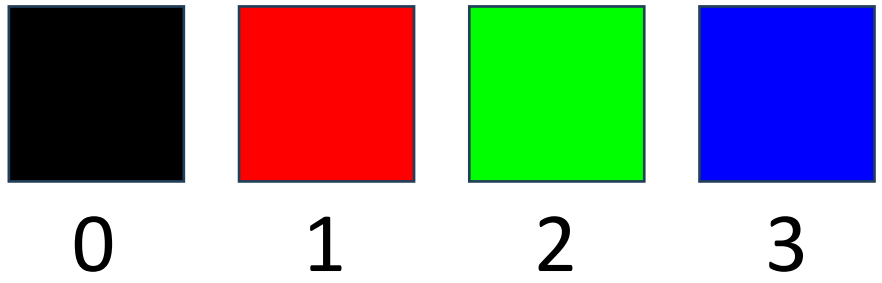




- There will always be eight candidate matches (0-7)
- One pattern must be matched (report a value 0-7)
- There will always be exactly one EXACT match among the candidates
- The parts of the pattern are not necessarily connected
- Do not make assumptions about the number or size of the parts of the pattern.
- The nonblack pixels of the pattern will not occur as a strict subset of the nonblack pixels of any candidate

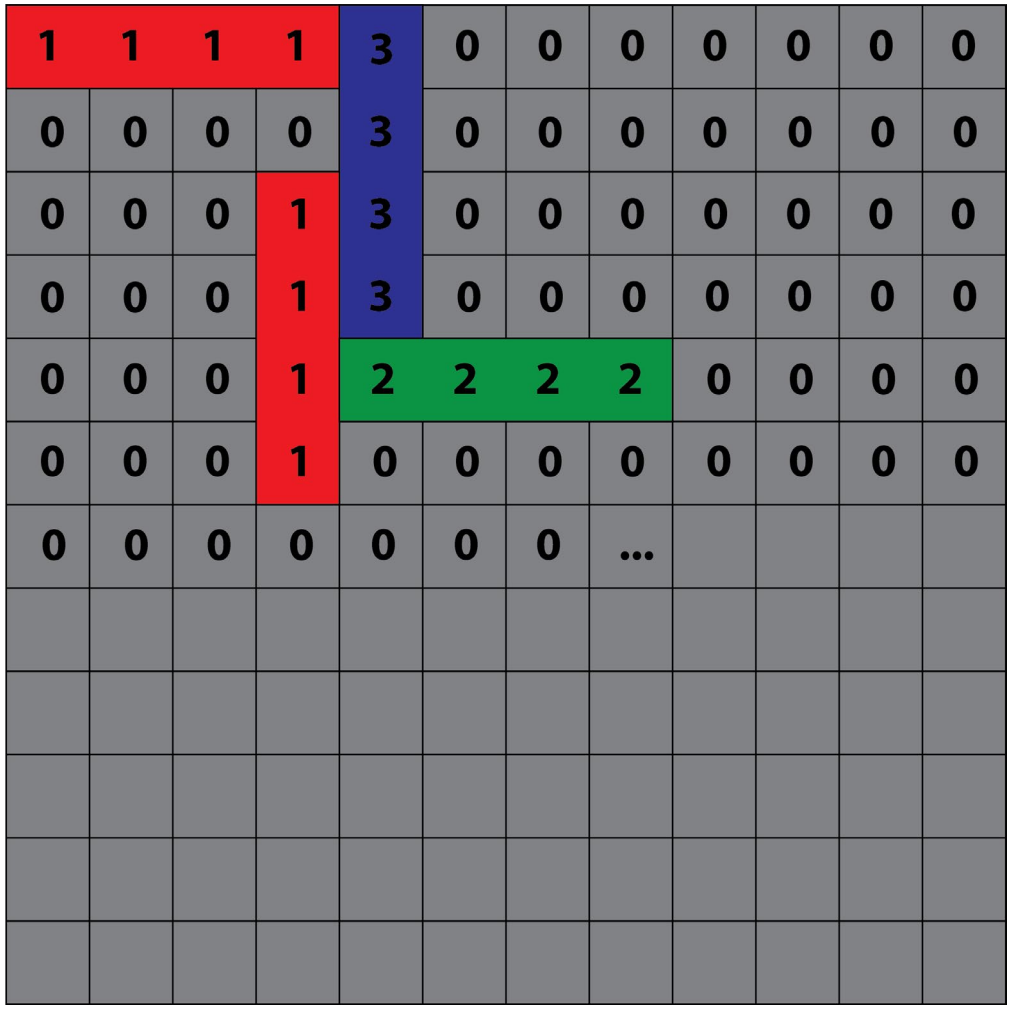
Icon and Color Palette

1	1	1	1	3	0	0	0	0	0	0	0
0	0	0	0	3	0	0	0	0	0	0	0
0	0	0	1	3	0	0	0	0	0	0	0
0	0	0	1	3	0	0	0	0	0	0	0
0	0	0	1	2	2	2	2	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	…				



- Each pixel value is 1 of 4 colors
- Each icon is 12 × 12 pixels
- Background is all black (0)
  - Shown as gray here
- Grid is NOT present in actual icons

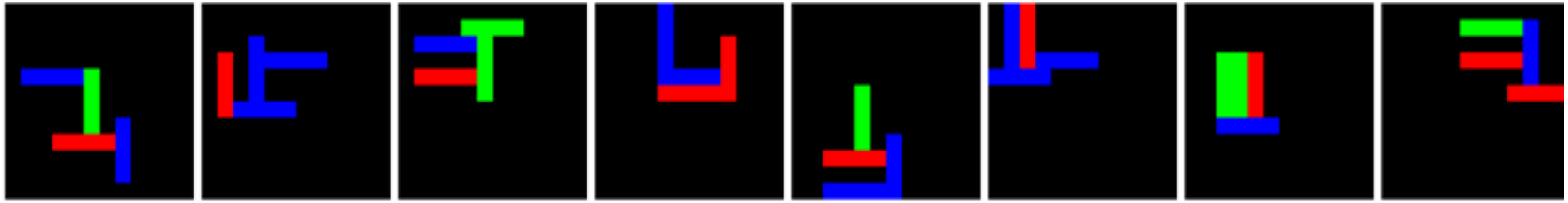
Array representation of a single icon



- Row-major
- 1 pixel value per array element
- This example is

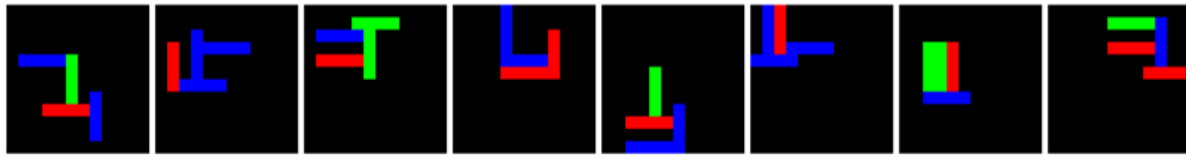
{1, 1, 1, 1, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, … }

Array representation of 8 candidate icons



- Each candidate is row-major, exactly as shown on previous slide
- The eight candidates are one after another in a large array
- The image above is misleading – it is more like a 96 row $\times$ 12 column array

Storage order of 8 candidate icons



0 1 2 3 4 5 6 7

Element 0
(Candidate 0, pixel 0)

{0, … 0, 0, …, 0, … 0, … 0 }

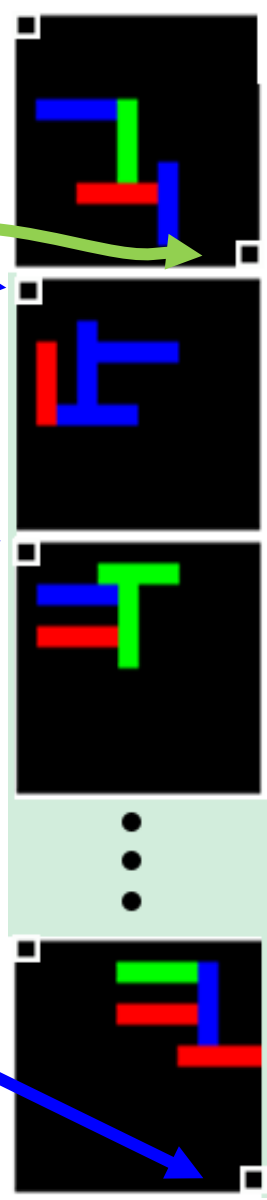
Element 143

Element 144

Element 288

Element 1008
(Candiate 7, pixel 0)

Element 1151



144 values

1152 values

P1-1: C Implementation

- Declarations:

```
#define IconSize (12 * 12)
#define NumCandidates 8
#define CandidateArraySize (IconSize * NumCandidates)
int Candidates[CandidateArraySize];
int Pattern[IconSize]
```

IconSize is the constant 144

144 × 8 = 1152

Stores 1152 ints for Candidates, followed by 144 for Pattern
(just as MIPS version will do)
It COULD HAVE been 1152 bytes, but they are ints!

- Grading:

```
// your program should use this print statement to report the answer
printf("The matching icon is %d\n", Match);
```

Shell code sets Match=42, so change that!

- Functional version: not evaluated on performance, only accuracy

P1-2: MIPS Implementation

```
.data
CandBase:    .alloc 1152
PatternBase:.alloc 144
.text
...
```

followed by SWI calls and your solution

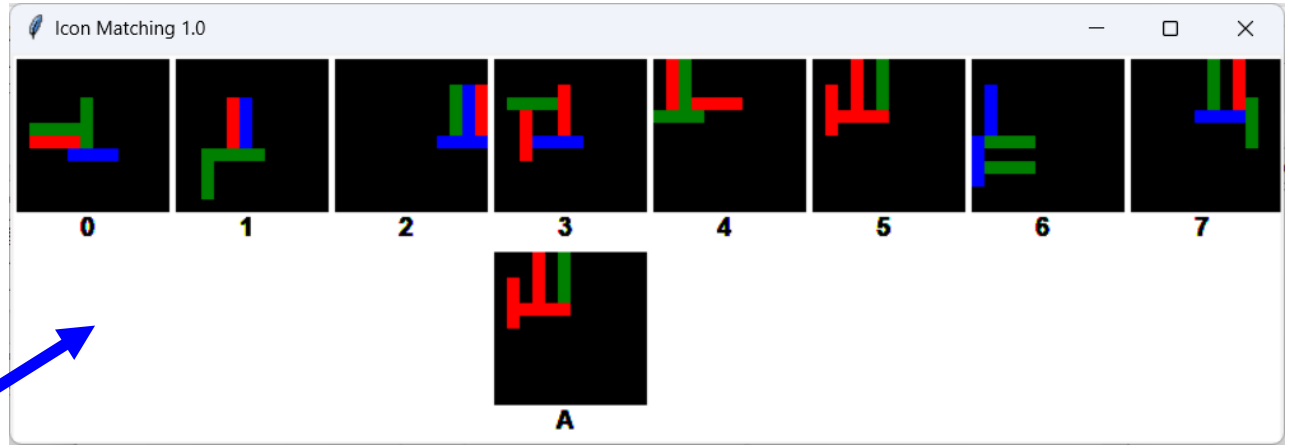
- Space for 8 12 ×12 candidates
- Space for 1 12 ×12 Pattern
- Same order as in C version

P1-2: Software Interrupts

1. SWI 584: Create Puzzle

INPUTS: $1 = base address

OUTPUTS: none - memory populated with image array



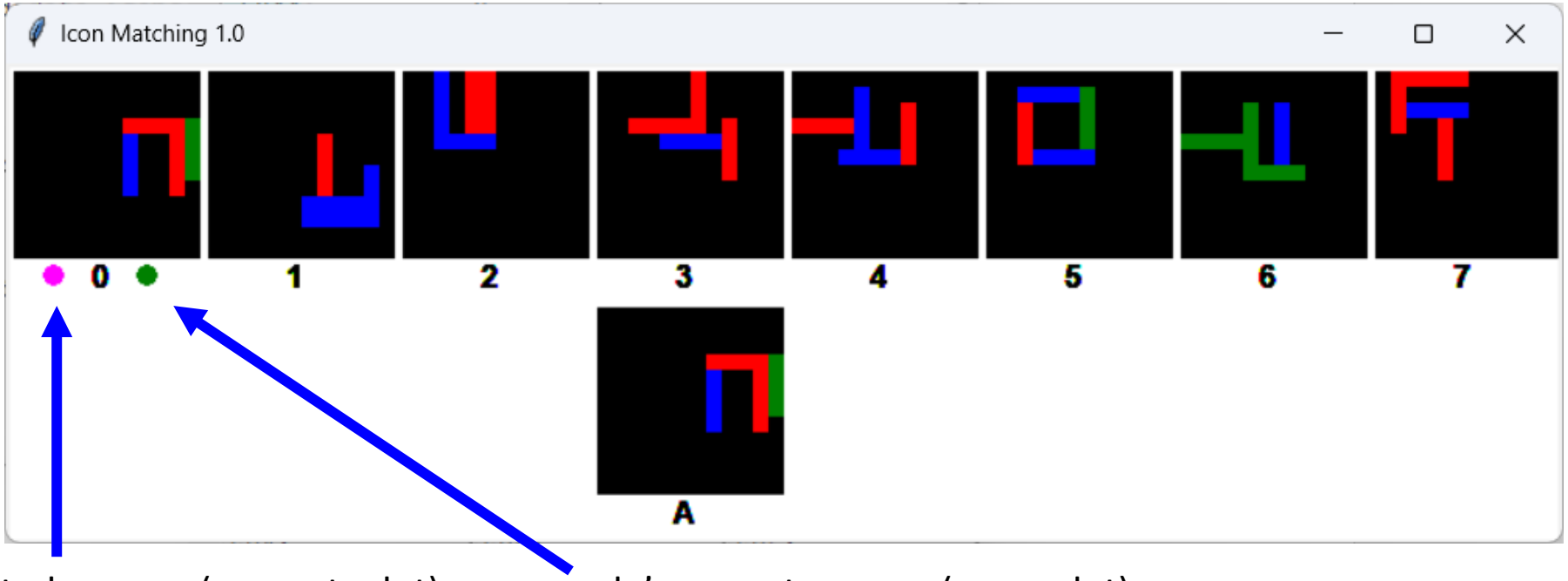

```
.data
CandBase:     .alloc 1152
PatternBase:  .alloc 144

.text
IconMatch:   addi   $1, $0, CandBase   # point to base of Candidates
             swi    584                # generate puzzle icons
```

P1-2: Software Interrupts

2. SWI 544: Match Ref: This routine allows you to report the number of the candidate icon that matches the pattern icon.

INPUTS: $2 a number between 0 and 7, inclusive (candidate #)

OUTPUTS: $3 gives the correct answer from oracle



submitted answer (magenta dot)

oracle's correct answer (green dot)

P1-2: Software Interrupts

3. SWI 585: Mark Icon Pixel:

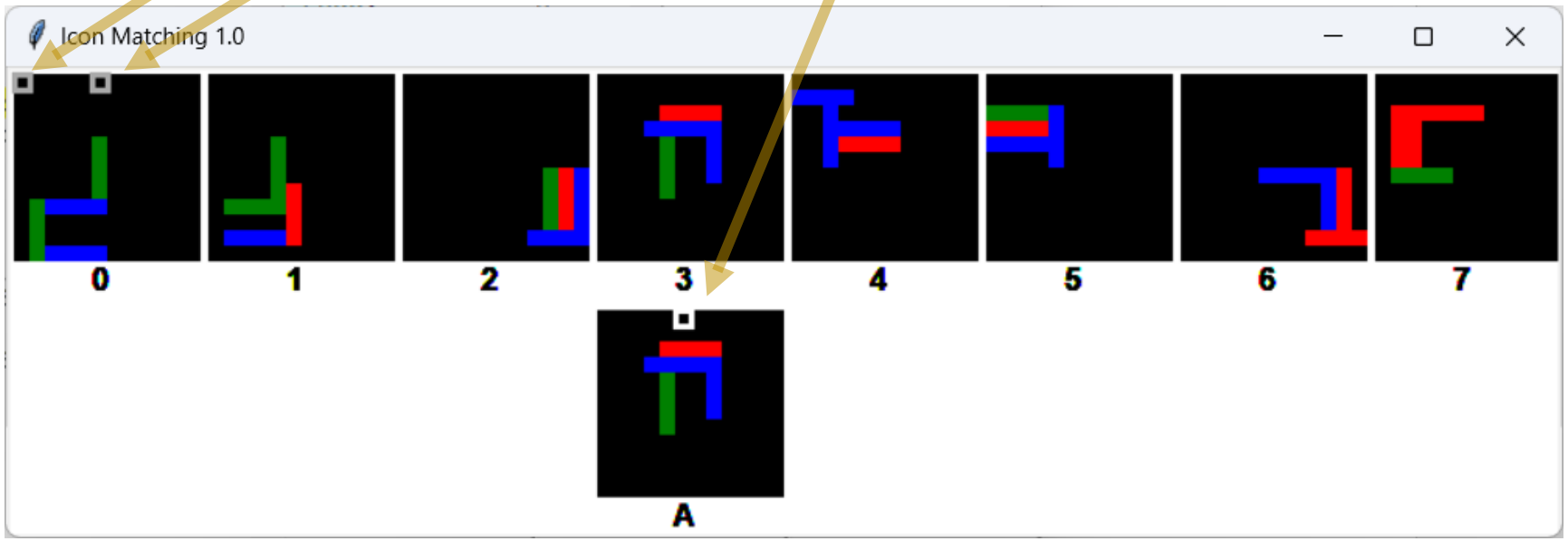
INPUTS: $2 should contain an address of a pixel. The address should be within the 1296 word region allocated for Candidates and the Pattern

Cells that have been highlighted previously in the trace are drawn with a gray outline.

OUTPUTS: none.

```
addi    $2, $0, CandBase # 0th icon, 0th cell
swi     585
addi    $2, $2, 20       # 0th icon, cell 5
swi     585
addi    $2, $0, PatternBase
addi    $2, $2, 20       # pattern, cell 5
swi     585
```

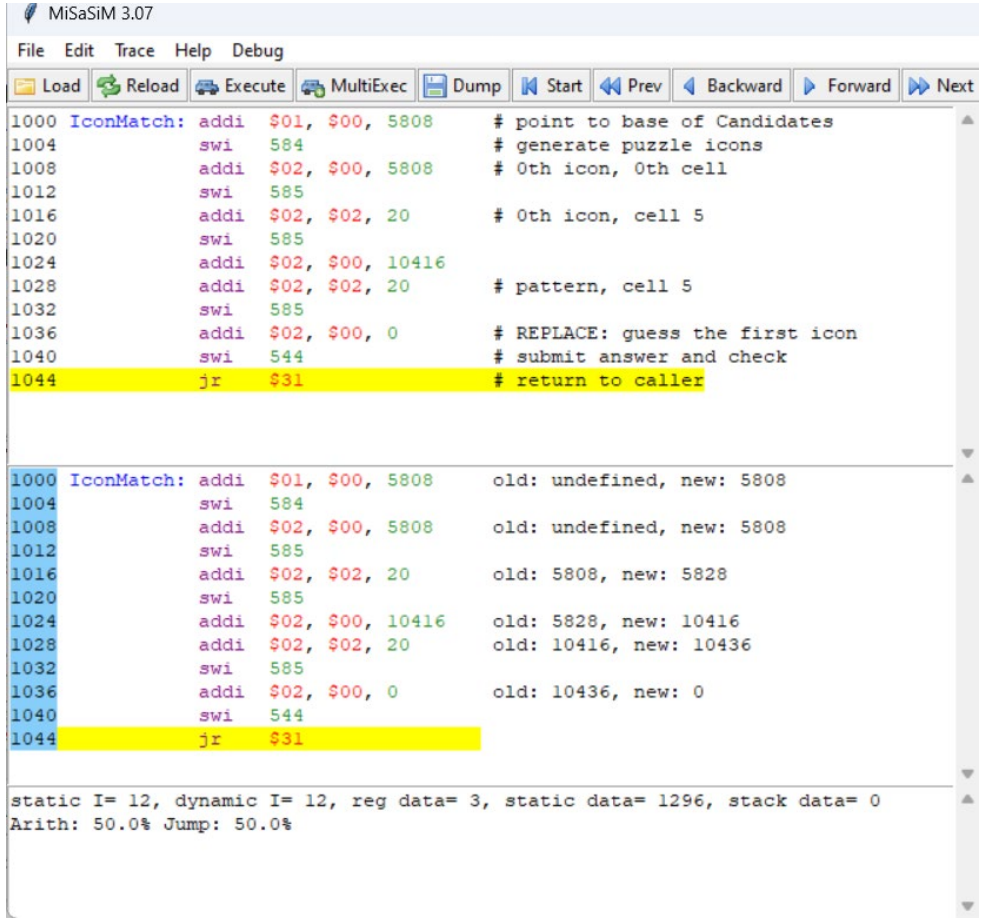
For debugging only



P1-2: Accuracy *and Efficiency* Evaluated

```
static I= 12, dynamic I= 12, reg data= 3, static data= 1296, stack data= 0
Arith: 50.0% Jump: 50.0%
```

- Statistics for shell code above
- Very efficient (but tests show not very accurate)



P1-2: Efficiency Evaluation

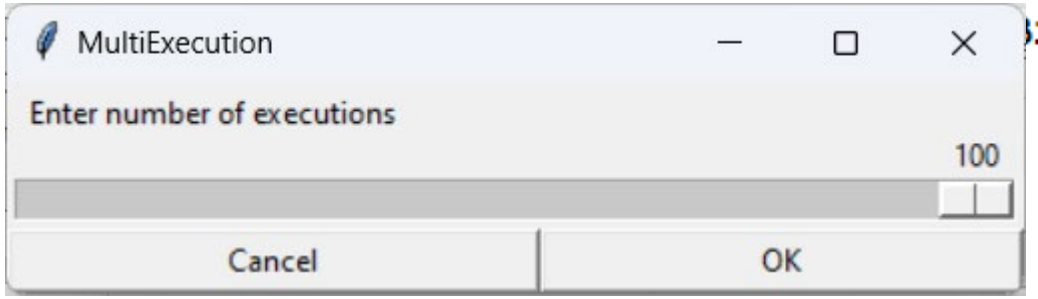
- Your Goal: Beat the baseline

  static code size: <u>38</u> instructions, dynamic instruction length: <u>350</u> instructions (avg.), total register and memory storage: <u>11</u> words (not including dedicated registers $0, $31, or the 1296 words for the input puzzle array)

- You should run MultiExecute to get average DI, storage over multiple runs

```
static I= 30, dynamic I= 327, reg data= 7, static data= 1024, stack data= 0
Arith: 34.3% Jump: 0.9% Load: 32.4% Branch: 32.4%
```

```
static I= 30, dynamic I= 1179, reg data= 7, static data= 1024, stack data= 0
                                                  ranch: 33.1%
```



We'll run 100 trials (same trials for all students).

Score

	Part	Description	Points
Accuracy Metrics	P1-1	Icon Matching (C code)	25
	P1-2	Icon Matching (MIPS code)	
		checkpoints, correct operation, proper commenting/style	25
Performance Metrics		static code size	15
		dynamic execution length	25
		storage requirements (registers, memory)	10
		Total	100

For Accuracy Metrics:

- Points reduced by 10% for each failed trial
- 0 points if 10 or more trials fail AND (for P1-2) no points for any performance metrics

Score

	Part	Description	Points
Accuracy Metrics	P1-1	Icon Matching (C code)	25
	P1-2	Icon Matching (MIPS code)	
Performance Metrics		checkpoints, correct operation, proper commenting/style	25
		static code size	15
		dynamic execution length	25
		storage requirements (registers, memory)	10
		Total	100

For each P1-2 Performance Metric:

Points = PercentCredit x (Performance Metric Points)
where

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

Baseline Metrics:
static code size: <u>38</u> instructions,
dynamic inst length: <u>350</u> instructions (avg.),
total register & memory storage: <u>11</u> words

E.g., if your program has a 315 avg dynamic execution length,
Points for dynamic: (2-315/350) x 25 = 1.1 x 25 = 27.5 (out of 25)
If your program uses 12 words, storage points: (2-12/10) x 10 = 8 (out of 10)

P1-2 swi 584 Easter Egg

```
addi $1, $0, CandBase
addi $2, $0, -1
swi  584
```

- Ability to load in a previously dumped puzzle
- Debugging feature
- **Be sure to comment out this change to $2 before submitting your code** so that the autograder is not prompted for a test file.

Honor Code

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.*