x86_64 Assembly Tic Tac Toe

> This project will **only** run on x86_64/AMD64 Linux versions.

A Tic Tac Toe version that operates on a **18 bit** grid data structure, written in assembly and C.

About

Just recently I finished reading Jeff Duntemann's "Assembly Language Step-by-Step: Programming with Linux" book, a wonderful introduction to the Intel IA32 assembly language. I figured, writing a memory optimized version of Tic Tac Toe would be a great way to practice my newly acquired skills, as well as getting introduced to the newer x86_64 assembly language.

The core of the game has been written in assembly, which exposes gcc C compatible functions functions to set, read and evaluate the 3x3 Tic Tac Toe grid. Terminal IO and grid drawing is done in C, as the compiler likely generates more optimized code than I would If I were to write those features in assembly.

Obtaining a local copy

To obtain a local copy of the repository, simply clone it using git:

```
$ git clone https://github.com/CTXz/x86_64-Assembly-Tic-Tac-Toe.git
```

Alternatively, the repository may be downloaded as a zip.

Build and run

The following software must be available to successfully build an executable:

- GCC Linux x86_64
- NASM
- make

The repository contains a makefile that offers the following targets:

Target	Description

all (aka `make` without target)	Builds the Tic Tac Toe executable.
debug	Builds the Tic Tac Toe executable with the gcc and NASM `-g` flag.
clean	Removes any object code and binary executables in the project repo.

To build the `all` target, simply run:

```
$ make
```

To build another target, simply provide it as a argument:

```
$ make TARGET
```

`all` and `debug` will produce a binary executable called `ttt` which can be executed as it follows:

```
$ ./ttt
```

The Grid

So how exactly does the grid operate on 18 bits?

A 3x3 tic-tac-toe grid consist of a total of 9 fields, where each field can maintain a total of three different states at a time:

- Empty
- Circle
- Cross

The least required amount of bits to display three different states is two. As a result, we can create a very primitive 2 bit data structure that represents a single tic-tac-toe field:

	LOW - 0	HIGH - 1

Lower Bit	Empty	Filled
Higher Bit	Circle	Cross

The following table represents all possible bit pairs:

Bit Pair	Represented Field
00	Empty
10	Empty (This state should never occur)
01	Circle
11	Cross

The low/right bit defines whether the field is empty or not.
The left/hight bit defines whether the field is filled with a cross or circle

Using a primitive data structure as such, we may create a 1 dimensional 18 bit wide tic-tac-toe grid. The lowest two bits are mapped to the top left field and the highest two bits are mapped to the bottom right field.

The following 18 bit data structure:

```
010111001111010011
```

Would translate into the following 2d grid:

```
X|  |O
X|X|
X|O|O
```

As, x86_64 CPUs are only able to allocate 8, 16, 32 and 64 bits of memory at a time, the Tic Tac Toe field is technically running on 32 bits of reserved memory, however, only 20 bits are actively used. The remaining 12 bits can theoretically be used to store additional data.

Setting and reading the grid

The code makes heavy use of "bit masks" and bitwise operations. To understand how fields are set and and read, a understanding of the **OR** and **AND** bitwise operators should be established. Further, one should understand how **bit shifts** and **rotations** work as those are frequently utilized during grid evaluation.

- To understand how a field is set, see the reference for the set_field procedure.
- To understand how the current state of a field is obtained, see the reference for the get_field procedure.
- To understand how the grid is evaluated for victory, see the eval_grid reference.

Reference

Source Files located in `src/`:

File	Description
core.asm	Core procedures written in assembly (ie. grid setting and reading)
core.h	A C interface for global procedures exposed by core.asm
main.c	Game code that utilizes functions exposed by the assembly core module

Assembly Reference:

- init_grid
- eval_grid
- set_field
- get_field

init_grid

> Source: core.asm

Description

Initializes/Empties the tic-tac-toe field by overwriting it's allocated memory with zeros.

init_grid **must** be called before any other core procedure, else the allocated memory may be filled with garbage and procedures will result in undefined behavior.

C Call

```
void init_grid();
```

C Example

```
#include "core.h"

int main()
{
    init_grid();
    // Handle grid here...
    return 0;
}
```

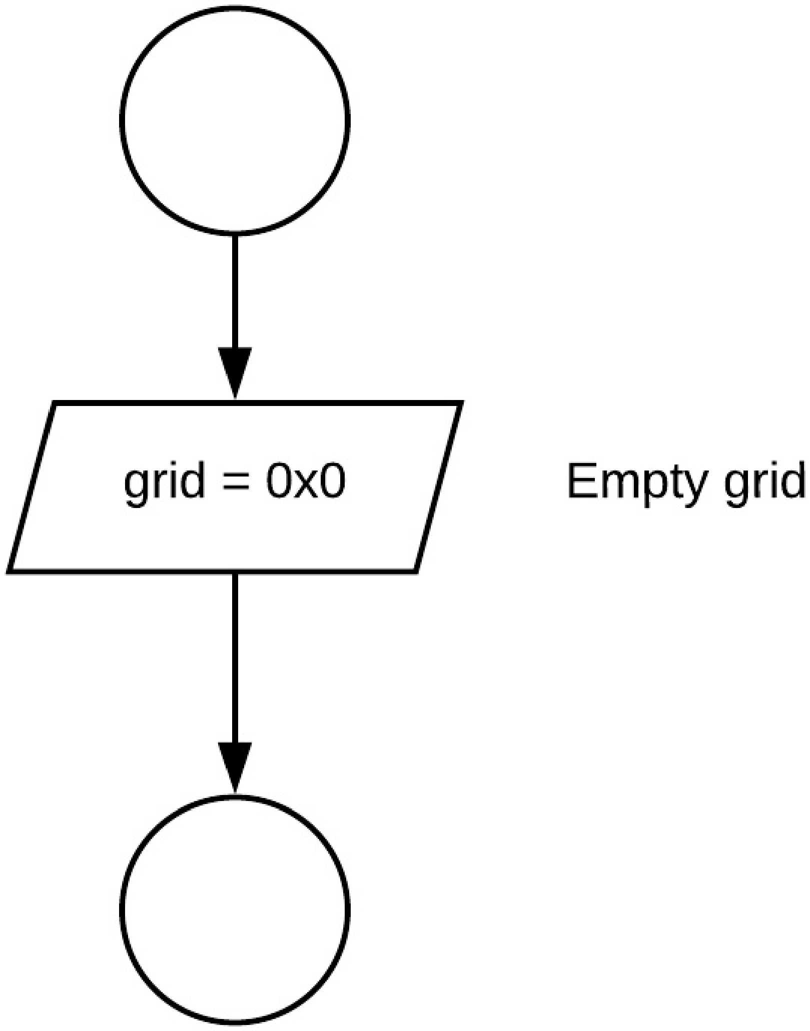
Assembly Example

```
main:
call init_grid
```

Procedure Diagram

The diagram may be viewed online here

init_grid



eval_grid

> Source: core.asm

Description

Evaluates the grid for victory.

Parameters

Register	Description
RDI	Player, where `0x0` checks for Circle and `0x1` checks for Cross

Returns

RAX/Return Value	Evaluation
`0x0`	No Victory
`0x1`	Victory

C Call

```
bool eval_grid(bool cross);
```

C Example

In the following example, we will test the grid for a X victory and announce it if player X has won.

```
if(eval_grid(true))
{
  printf("Player X has won!")
}
```

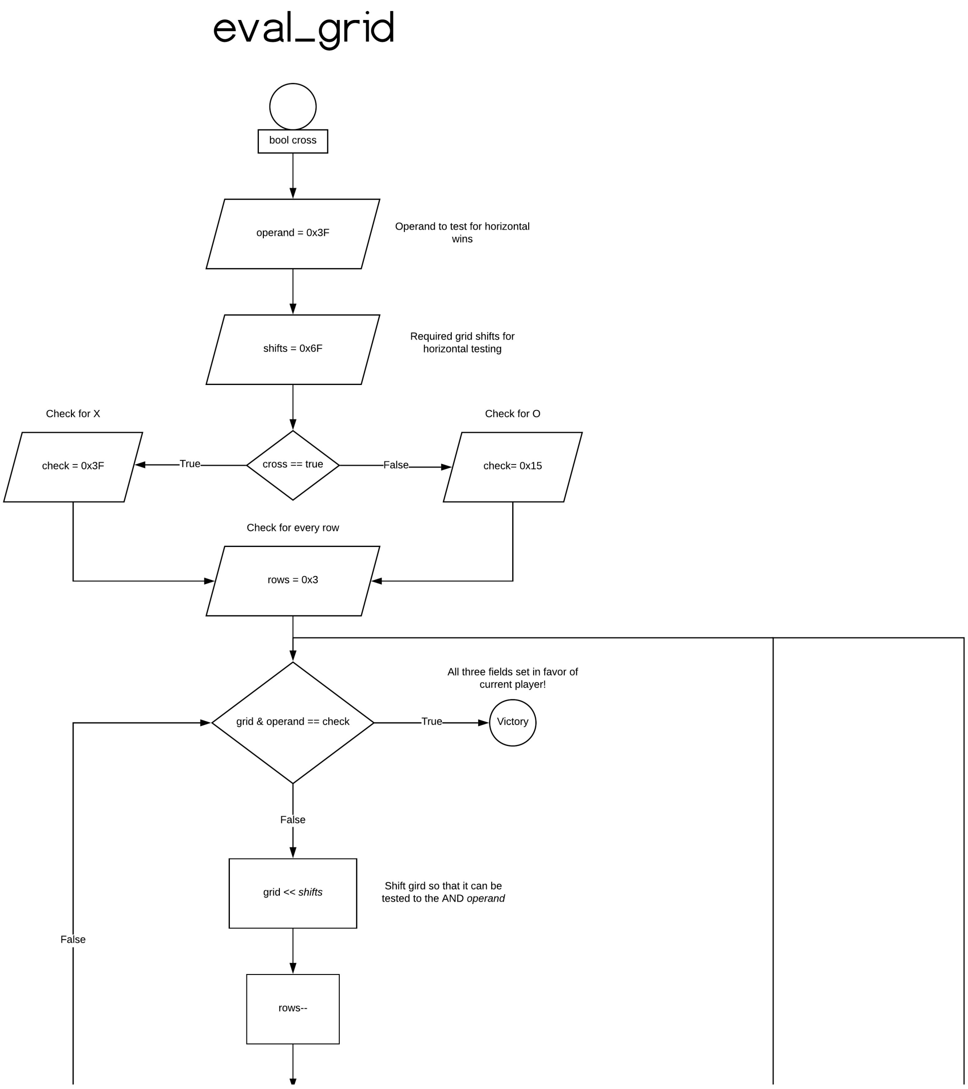
Assembly Example

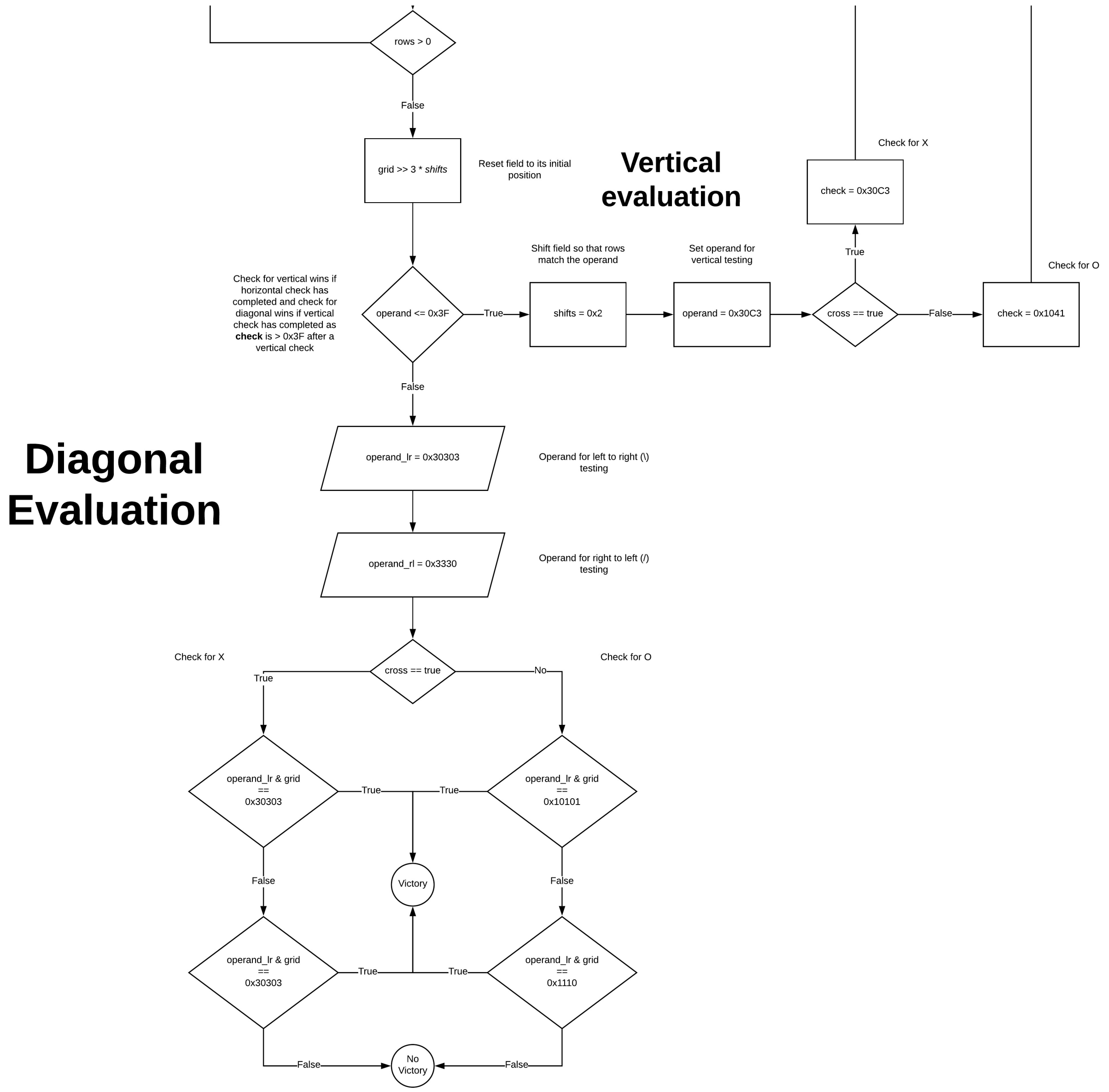
In the following example, we will test the grid for a cross victory and jump to the `win` label if player cross has won.

```
mov   RDI, 0x1    ; Check for cross victory
call  eval_grid   ; Evaluate grid for victory

test  rax, rax    ; Check for victory
jne   victory     ; Jump to victory label if player X won
```

Procedure Diagram

Full diagram may be accessed here





Procedure Overview

eval_grid evaluates the grid for a total of three different victories:

Horizontal Victory

First, the procedure checks the grid for a horizontal victory by testing each row to a bit mask of `0x3F` (which translates to `111111` in binary). If the evaluation is made for player O, the bitwise AND operation must match `0x15` to make for a horizontal victory. In contrast, if the evaluation is made for player X, the bitwise AND operation must match `0x3F` to make for a horizontal victory.

Each row is tested by rotating the grid by `0x6` to the right, thereby ensuring that the rows align with the `0x3F` bit mask.

If any of the rows match a victory, the procedure returns `0x1` on `RAX`.

In order to preserve the original state of the grid, rotating is only done to a mutable copy of the grid which is loaded into `RDX`, which can be safely discarded without affecting the actual allocated memory for the grid.

> See illustrated example bellow for a visual assistance.

Vertical Victory

Should a horizontal victory not be present, the procedure will further test for a vertical victory. Each columns is tested against `0x30C3` (which translates to `11000011000011` in binary). If the evaluation is made for player O, the bitwise AND operation must match `0x1041` to make for a vertical victory. In contrast, if the evaluation is made for player X, the bitwise AND operation must match `0x30C3` to make for a vertical victory.

Similar to horizontal testing, each column is tested by rotating the grid by `0x2` to the right, thereby ensuring that the columns align with the `0x30C3` bit mask.

If any of the columns match a victory, the procedure returns `0x1` on `RAX`.

Diagonal Victory

Once again, should a vertical victory be absent, the procedure will test for a diagonal victory. Unlike horizontal and vertical victories, no rotations are applied to the grid. Instead, the procedure evaluates for a diagonal spanning from the top left to the bottom right, and a diagonal spanning from the bottom left to the top right.

For a top left to bottom right victory, the field is tested against `0x30303` (which translates to `110000001100000011`), where the bitwise AND operation must return `0x10101` for a Circle win, and `0x30303` for a cross win.

For a bottom left to top right victory, the field is tested against `0x3330` (which translates to `11001100110000`), where the bitwise AND operation must return `0x1110` for a Circle win, and `0x3330` for a cross win.

To further explain, three examples are given, each illustrating evaluation for a different victory.

Horizontal

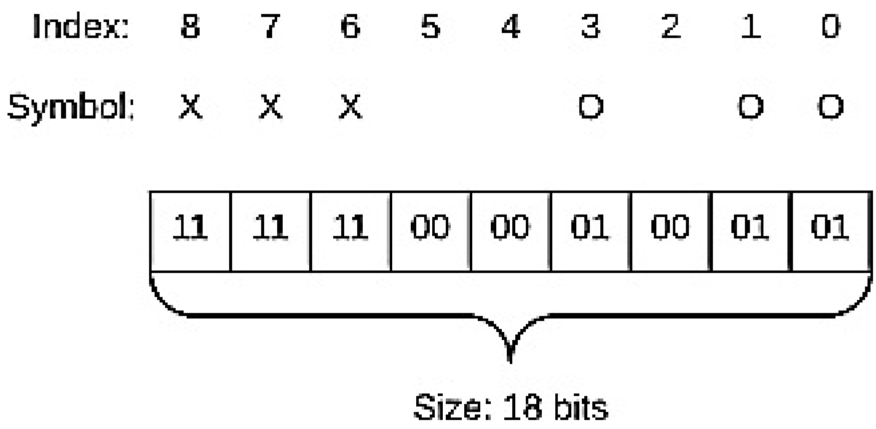
In the following example, the underlying grid is evaluated for a horizontal win for player X:

O	O	
O		
X	X	X

If we replace fields with their corresponding binary data structures, a 2d grid would manifest its information as it follows:

01	01	00
01	00	00
11	11	11

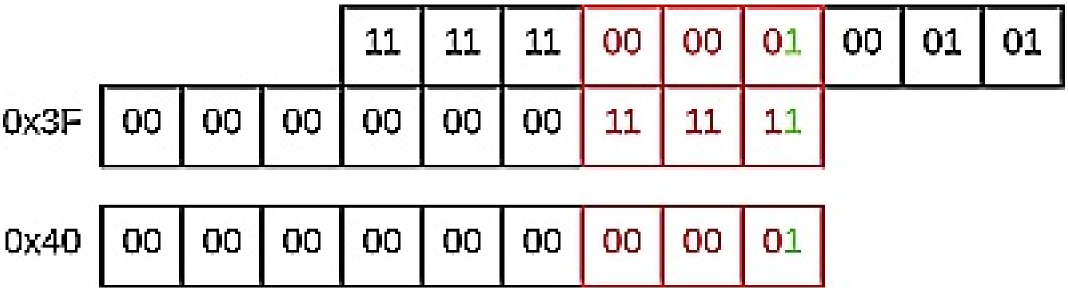
The 2d grid would translate to the following array of 18 bits in size:



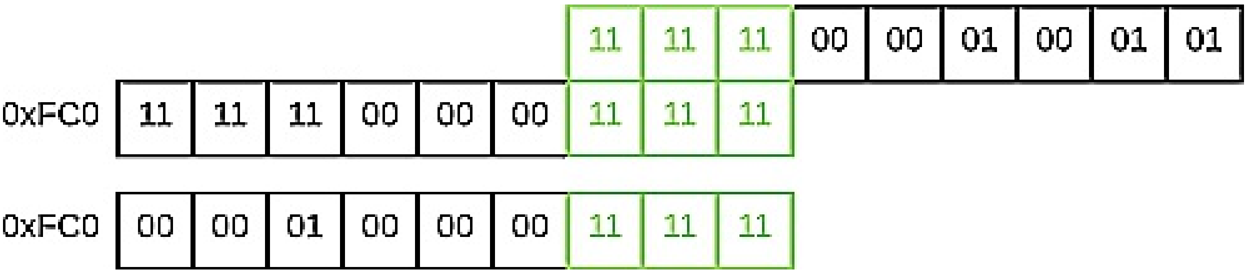
To check for victory, we begin by testing the first three fields by applying an AND operation to the 18 bit grid to value `0x3F`:

	11	11	11	00	00	01	00	01	01
0x3F	00	00	00	00	00	00	11	11	11
0x05	00	00	00	00	00	00	00	01	01

The result of the AND operation does not match `0x3F`, meaning one or more fields in this row are empty or reserved by the opposing player O. To test the next row, the grid is shifted 6 times (by `0x6`) to the right. After the field has been shifted, the grid is once again tested:



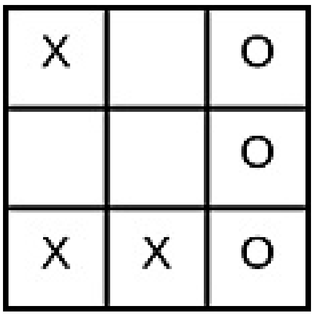
Once again, the fields mismatch. Let's shift once more, and see what happens if all three fields match:



All three fields fields match! It's a victory!

Vertical

In the following example, the underlying grid is evaluated for a vertical win for player O:



If we replace fields with their corresponding binary data structures, a 2d grid would manifest its information as it follows:

11	00	01
00	00	01
11	11	01

The 2d grid would translate to the following array of 18 bits in size:

Index:	8	7	6	5	4	3	2	1	0
Symbol:	X	X	O			O		X	O
	01	11	11	01	00	00	01	00	11

Size: 18 bits

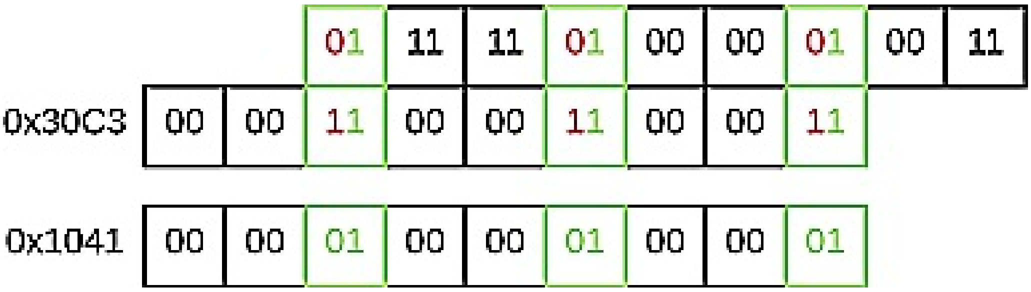
To check for victory, we begin by testing the 1st, 4th and 7th field by applying an AND operation to the 18 bit grid to value `0x30C3`:

	01	11	11	01	00	00	01	00	11
0x30C3	00	00	11	00	00	11	00	00	11
0x3003	00	00	11	00	00	00	00	00	11

The result of the AND operation does not match `0x1041` which is the value that matches a vertical O win, meaning one or more fields in this column are empty or reserved by the opposing player O. To test the next row, the grid is shifted 2 times (by `0x2`) to the right. After the field has been shifted, the grid is once again tested:

		01	11	11	01	00	00	01	00	11
0x30C3	00	00	11	00	00	11	00	00	11	
0x3000	00	00	11	00	00	00	00	00	00	

Once again, the fields mismatch. Let's shift once more, and see what happens if all three fields match:



All three fields fields match! It's a victory!

Diagonal

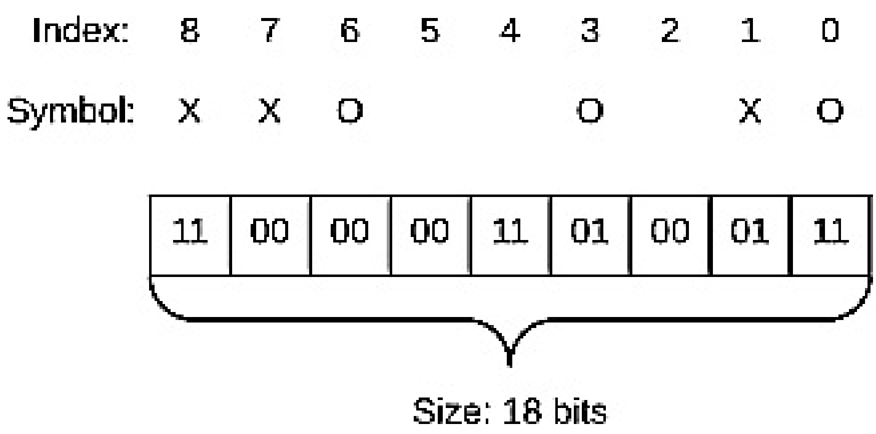
In the following example, the underlying grid is evaluated for a diagonal win for player X:

X	O	
O	X	
		X

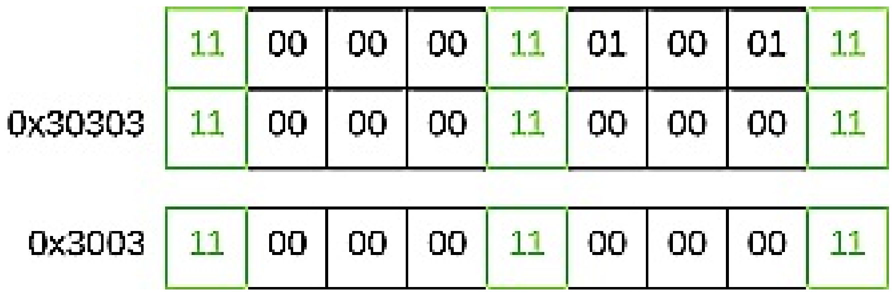
If we replace fields with their corresponding binary data structures, a 2d grid would manifest its information as it follows:

11	01	00
01	11	00
00	00	11

If we replace fields with their corresponding binary data structures, a 2d grid would manifest its information as it follows:



For diagonal wins, no shifting is done. Instead, the field is tested against two bit. One that tests for a diagonal win from the top left to the bottom right (`0x30303`) and one that checks tests for a diagonal win from the bottom left to the top right (`0x3330`). The program begins by checking for a diagonal win spanning from the top left to the bottom right, which will match our field:



All three fields fields match! It's a victory!

set_field

> Source: core.asm

Description

Sets the state of the field at position x and y.

Parameters

Register	Description
RDI	X Co-ordinates
RSI	Y Co-ordinates
RDX	State, where `0x0` = Circle and `0x1` = Cross

C Call

```
void set_field(uint8_t x, uint8_t y, bool state);
```

C Example

In the following example, the center field is set to a cross:

```
if (get_field(1, 1) == EMPTY) // EMPTY is enumerated in core.h!
{
  set_field(1, 1, true);
}
```

Assembly Example

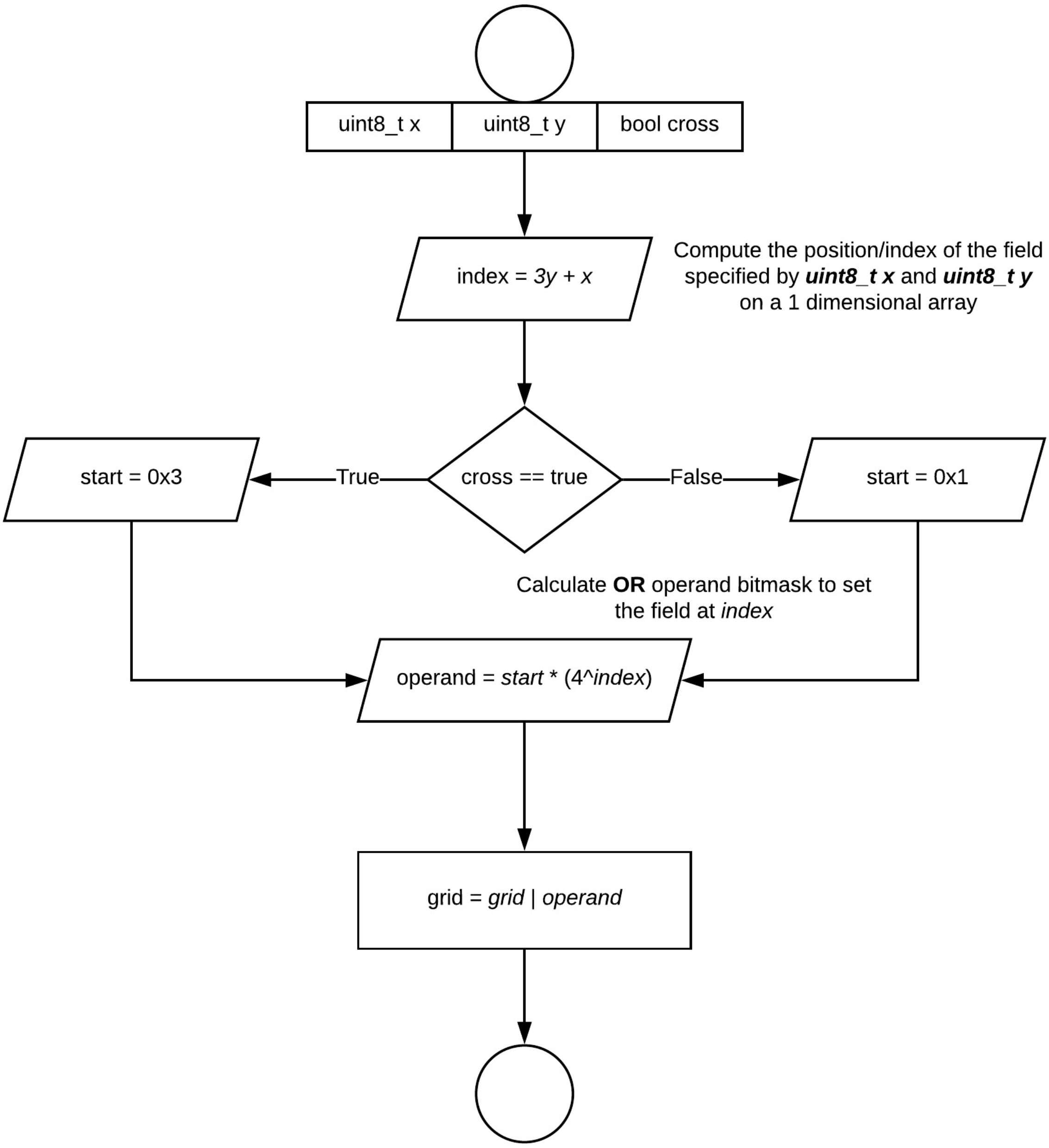
In the following example, the center field is set to a cross:

```
mov   RDI, 0x1    ; x = 1
mov   RSI, 0x1    ; y = 1
mov   RDX, 0x1    ; set state to cross
call  set_field   ; Call procedure
```

Procedure Diagram

Full diagram may be accessed here

set_field



Procedure Overview

set_field fetches the state of the field assigned to the provided x and y co-ordinates. This is achieved by first converting the x and y co-ordinates into a index, thereby obtaining the position of the specified field on a 1 dimensional field array.

The field index is calculated by the following equation:

$$index = 3y + x$$

With the index calculated, we then compute an OR operand that will set the state of the bits representing the field at the index.

The equation that computes the OR operand takes two values:

- The initial quantity, *a*, which is determined by the shape to which the field is set.

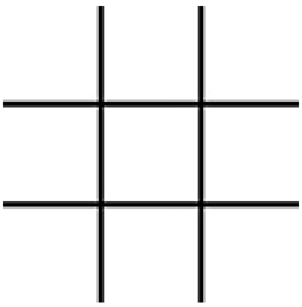
- The index *i*, which has already been computed from the x and y co-ordinates.

  The initial quantity *a* is set to `0x1` for circles (`cross = false`) and `0x3` for cross (`cross = true`).

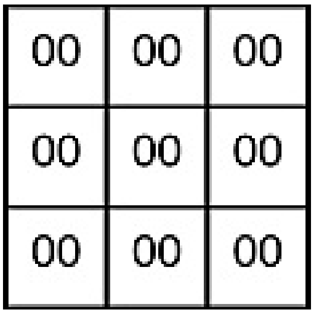
$$operand = a \cdot 4^i$$

With the operand returned, a bitwise OR instruction is applied against the grid, which sets the field to the desired state.

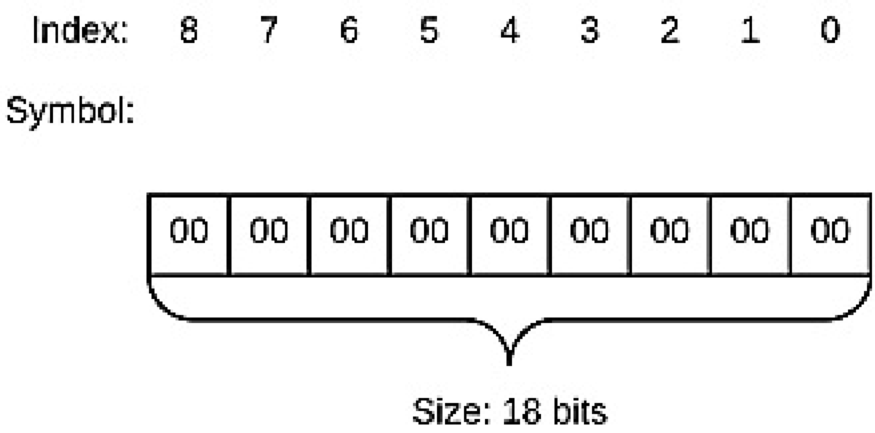
To further explain, the following example is guided by an illustration:
In the following grid, we will set the top left field to a circle:



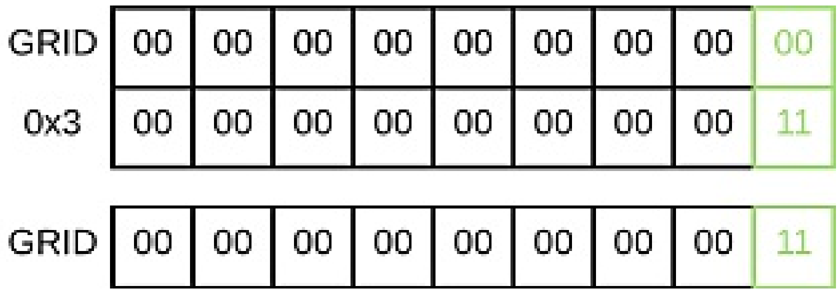
If we replace fields with their corresponding binary data structures, a 2d grid would manifest its information as it follows:



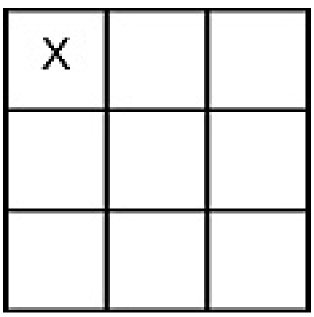
The 2d grid would translate to the following 18 bit data structure:



To set the state of the top left field, an bitwise OR operation is applied, where `0x3` is the bit mask to set the top left field of the grid:



The data structure would now yield the following 2d grid:



get_field

> Source: core.asm

Description

Obtains the state of the field at position x and y.

Parameters

Register	Description
RDI	X Co-ordinates
RSI	Y Co-ordinates

Returns

RAX/Return Value	Field State
0x0	Cross
0x1	Circle
0x2	Empty

C Call

```
uint8_t get_field(uint8_t x, uint8_t y);
```

C Example

In the following example, we will test the middle field and print a message describing its current state:

```
switch(get_field(1, 1)) // Co-ordinates start at 0!
{
  case 0x0 :
    printf("Field set to X\n");
    break;

  case 0x1 :
    printf("Field set to O\n");
    break;

  default:
    printf("Field is empty!\n");
}
```

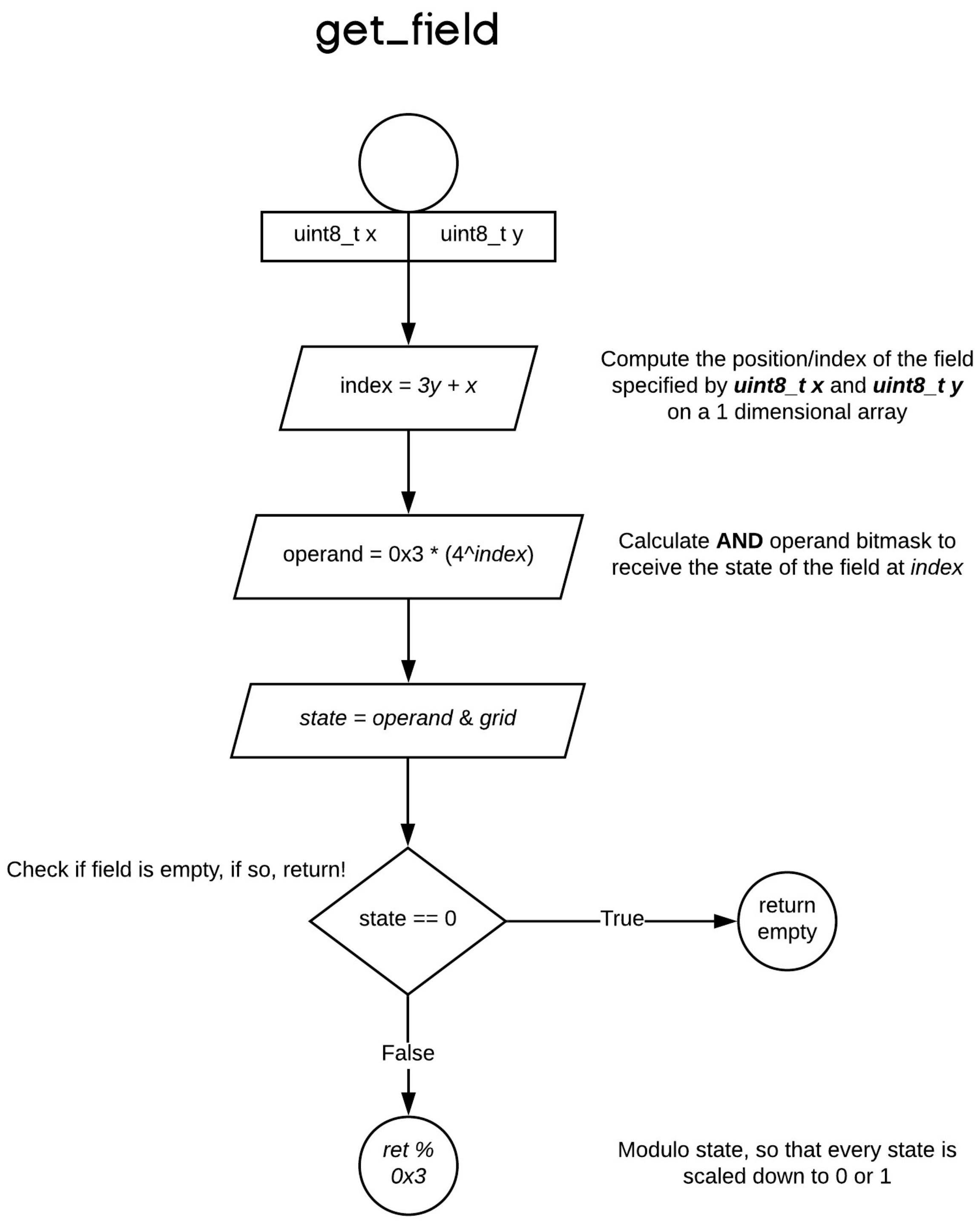
Assembly Example

The following example tests the middle field and jumps to `print_cross` if the field is set to a cross, `print_circle` if the field is set to a circle and `print_empty` if the field is empty:

```
mov   rdi, 0x1      ; x = 1
mov   rcx, 0x1      ; y = 1
call  get_field     ; call procedure

or    rax, rax      ; if return is 0
je    print_cross  ; Jump to print_cross label

and   rax, 0x1      ; if return is 1
je    print_circle ; Jump to print_circle label

jmp   print_empty  ; else return is 2
...

print_cross:
...

print_circle:
...
```

Procedure Diagram

Full diagram may be accessed here



Procedure Overview

get_field fetches the state of the field assigned to the provided x and y co-ordinates. This is achieved by first converting the x and y co-ordinates into a index, thereby obtaining the position of the specified field on a 1 dimensional field array.

The field index is calculated by the following equation:

$$index = 3y + x$$

With the index calculated, we then compute an AND operand that will fetch the state of the bits representing the field at the index. The operand is computed with the following equation:

$$operand = 0x3 * 4^{i}$$

Where `i` is the field index.

After the operand has been tested, the resulting value is checked for 0. Should this be the case, the field is empty, and 0 is returned. Should the operation not return a value of zero, it is scaled/narrowed down by applying a modulo of 0x3 to it, which will result in:

- 0 - For Circle
- 1 - For Cross

As a circle field is always even and a cross field is always odd.

To further explain, the following example is guided by an illustration:
In the following grid, we will test the middle field for its state:

X		
X	O	
X	O	

If we replace fields with their corresponding binary data structures, a 2d grid would manifest its information as it follows:

11	00	00
11	01	00
11	01	00

The 2d grid would translate to the following 18 bit data structure:

Index:	8	7	6	5	4	3	2	1	0
Symbol:		O	X		O	X			X
	00	01	11	00	01	11	00	00	11

$\underbrace{\hspace{10em}}_{\text{Size: 18 bits}}$

To obtain the state of the middle field, an AND operation is applied, where `0x300` is the bit mask to read the center field of the grid:

GRID	00	01	11	00	01	11	00	00	11
0x300	00	00	00	00	11	00	00	00	00

0x200	00	00	00	00	01	00	00	00	00

Further, the result of the AND operation is applied to a modulo of 3, so that values are narrowed/scaled down to 0 or 1, where 0 represents even numbers and thereby returns a X and 1 represents odd numbers, thereby returning O.

```
0x200 % 0x3 = 0x1
```

The result is 1, meaning that the center field holds a circle!