Novosibirsk State University

Project

in the discipline of "Digital platforms"

theme of the work: The Game of Life

Made by:

Shemchuk Anna,

Dashkovsky Egor

Novosibirsk, 2025

**Introduction**

This project implements the Game of Life. All computations—such as iterating over each cell, counting neighbors, and determining the new state of a cell—are performed programmatically on the CDM-16 processor. However, the first version was originally designed for the CDM-8 processor, which was later replaced to increase performance.

We attempted several optimization strategies, including the creation and use of a set of cells to process. The idea behind this is that only live cells and their neighbors need to be processed. This optimization can provide a significant performance boost when the number of live cells is small. However, managing the set also consumes resources. It is important for us to use a set, because when adding a live cell and all its neighbors, a single cell might otherwise be processed up to 8 times. As a result, on average, if more than 12.5% of the total cells are alive, this approach may actually lead to slower execution. Although this algorithm was implemented, it was not included in the final version of the project.

Next, we will discuss the optimizations that were retained in the project and that contributed significantly to speeding up the computations.

[Link to the repository](https://github.com/ashemchuk/Logisim_Game_of_Life)

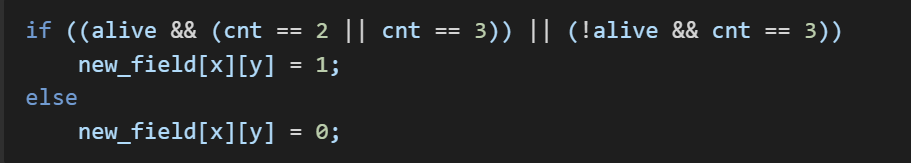
**Software**

* 1. The software implementation was developed in four stages, each being an optimized version of the previous solution:
  2. Naive implementation
  3. Encoding a cell in 4 bits + Lookup Table (LUT)
  4. Boolean function instead of LUT
  5. Packing four cells into a uint16\_t (two-byte words)

**Stage 1: Naive Implementation**

* The field is represented as a 2D array: uint8\_t field[16][16].
* Each cell’s state is stored as 0 (dead) or 1 (alive).
* A nested loop over the X and Y axes is used to iterate through all 256 cells.
* For each cell:

1. The number of live neighbors is counted by checking the eight surrounding cells in the array.
2. The following condition is checked:



1. The result is written using double buffering.

**Stage 2: Encoding a Cell in 4 Bits + Lookup Table (LUT)**

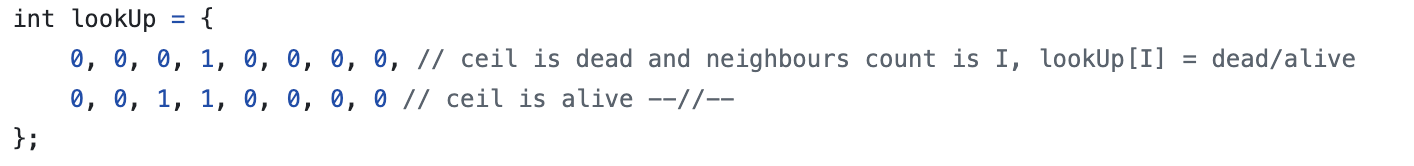
**Optimization goal:** Reduce the number of branching operations and separate memory accesses when calculating the next state.

***Cell***

One cell contains 4 bits of the form 0bvsss, where v stores alive or dead, and sss – the number of neighbors, neighbors can be from 0 to 8 inclusive, but a cell with 8(0b1000) and 0(0b0000) neighbors will be dead in the next generation, so three bits are enough to correctly encode information.

***Lookup Table (LUT)***

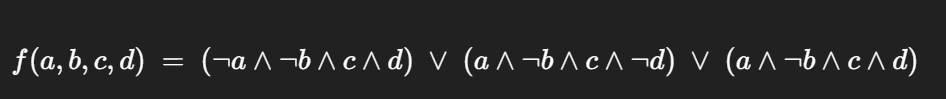
The same actions are currently being performed for each cell. The conclusion about the new state of the cell, based on the input data: the encoded state of the cell. Let's calculate the result in advance for the state of the cells and the number of neighbors, and put the result in the table. We got a look up table.

This implementation uses standard rules, so the value "1" is set for three cases: 0b0011 (dead + 3 neighbors), 0b1010 (alive + 2 neighbors), 0b1011 (alive + 3 neighbors). For all other input combinations, the value 0 is set in the truth table.

**Stage 3: Boolean Function Instead of LUT**

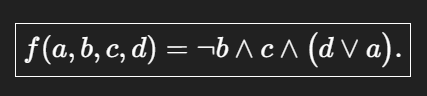
**Optimization goal:** Eliminate the overhead associated with storing, loading, and accessing the LUT by replacing it with a compact set of logical operations.

A LUT is a truth table from which you can output a Boolean function that gives the same results, but without accessing memory.

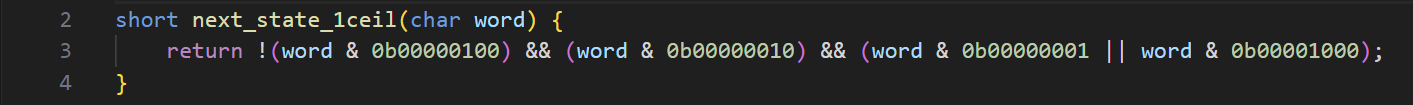
***Boolean Function:***  
It is necessary to define a Boolean function f(a, b, c, d), (where a is the state of the cell, bcd is the number of neighbors in the binary representation), which returns the value "1" exactly on the following combinations of bits: 0b0011, 0b1010, 0b1011, i.e. you can create the SDNF of this function: 

***Minimization of the Boolean Function***

The minimization resulted in a Boolean function of the following form:



In bitwise representation:



We select the necessary bits from the encoded value. It is important to note that both bitwise operations (AND, OR, NOT) and logical operations are used.

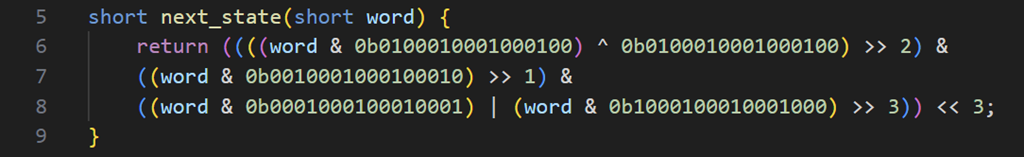
**Stage 4: Packing Four Cells into uint16\_t (Two-Byte Words)**

**Optimization goal:** Reduce the number of iterations and data transitions by processing four cells at once in one pass.

Each 16-bit word contains four 4-bit encoded cells in the format 0bvsss.

**Field:**

The field is extended with borders of dead cells (to simplify neighbor counting). The total size in blocks is 18×6 words uint16\_t.

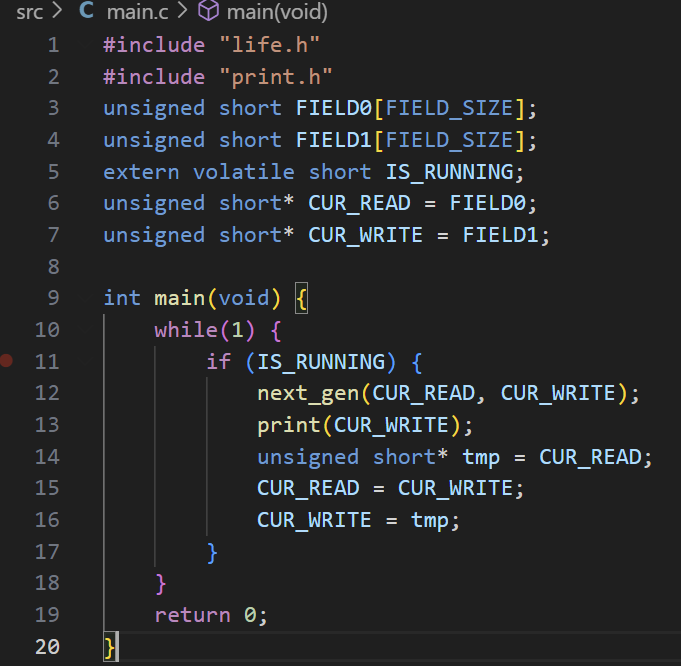
Modification of the function for 4 cells, using bitwise shifts to achieve the correct output format.

***Result:***

* Processing four cells at once per iteration instead of one.
* Minimum memory accesses and branching operations.
* The most optimized implementation among all those considered.

**Consider the project files:**  
The project consists of nine program files: six source code files and three header files. Files: **main.c**, **life.c** (life.h), **conclusion.c** (conclusion.h), **print.c** (print.h), **interrupts.c, start.asm.**

1. **main.c**



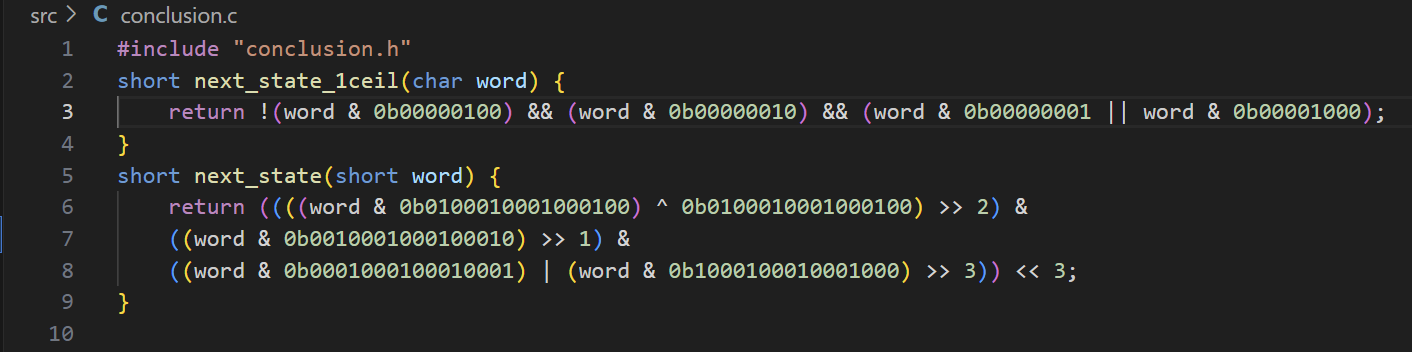
The **main.c** file runs an infinite while(1) loop. Each time the IS\_RUNNING flag is non-zero, it calculates the next generation and displays it on the screen. Double buffering is used: the first buffer is used for reading, the second for writing, then the buffers are swapped—this process repeats continuously.life.c

1. **life.c**

****

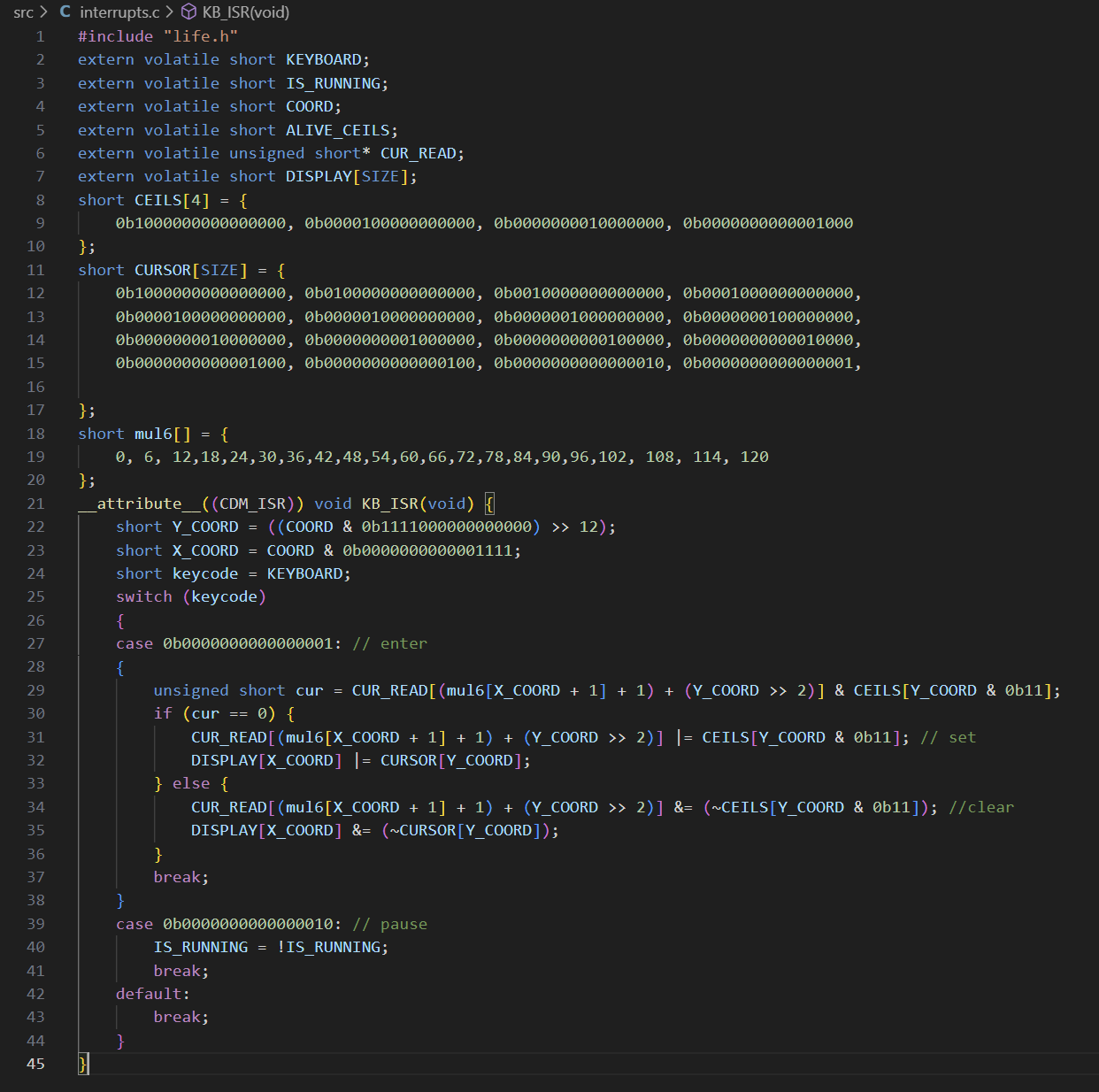
The next\_gen() function is implemented in **life.c**. Eight variables represent the presence of neighbors in the 0bX000 format for each cell. These variables are shifted so that the 1 bits are aligned to the least significant bit position for each cell, then they are summed up. Overflow is possible: for a cell with eight neighbors, if it was dead (most significant bit is zero), it becomes alive — 0b1000. However, 8 is the maximum number of neighbors, and the next state for a dead cell with 8 neighbors is the same as for a live cell with no neighbors. But if the cell was alive with 8 neighbors and we try to add its state to the neighbor count, the most significant bit of the cell overflows. This could either increase the neighbor count of the left cell or cause the bit to be lost entirely. Therefore, we don't add the current state of the cell to the neighbor count — instead, we overlay it using bitwise operations (see line 20). The next\_state() function determines the cell's state in the next generation (whether the cell is "alive" or "dead").

1. **conclusion.c**



The minimized Boolean function next\_state() is implemented in **conclusion.c**. It is used to calculate the state of a given cell in the next generation.

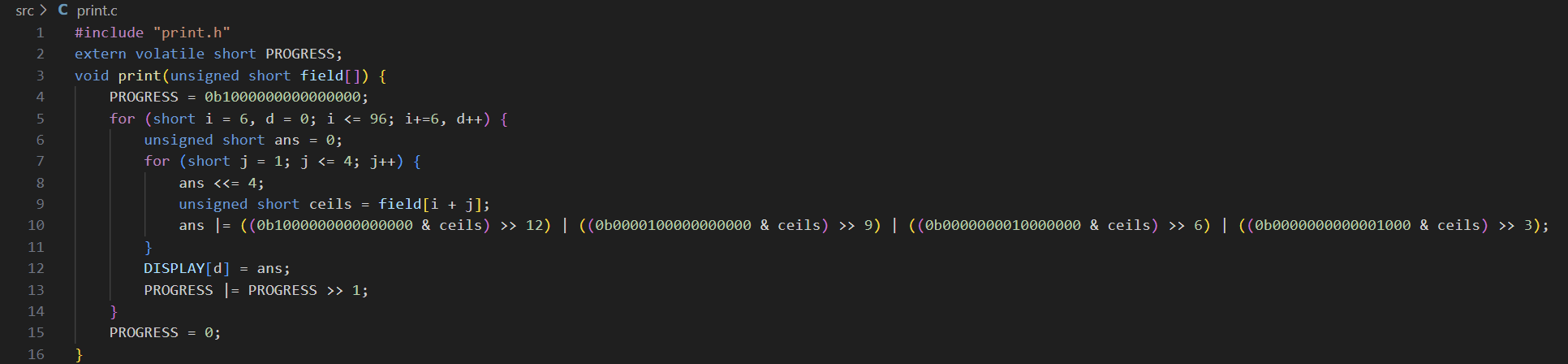
1. **interrupts.c**

****

The keyboard interrupt handler KB\_ISR is defined in **interrupts.c.**

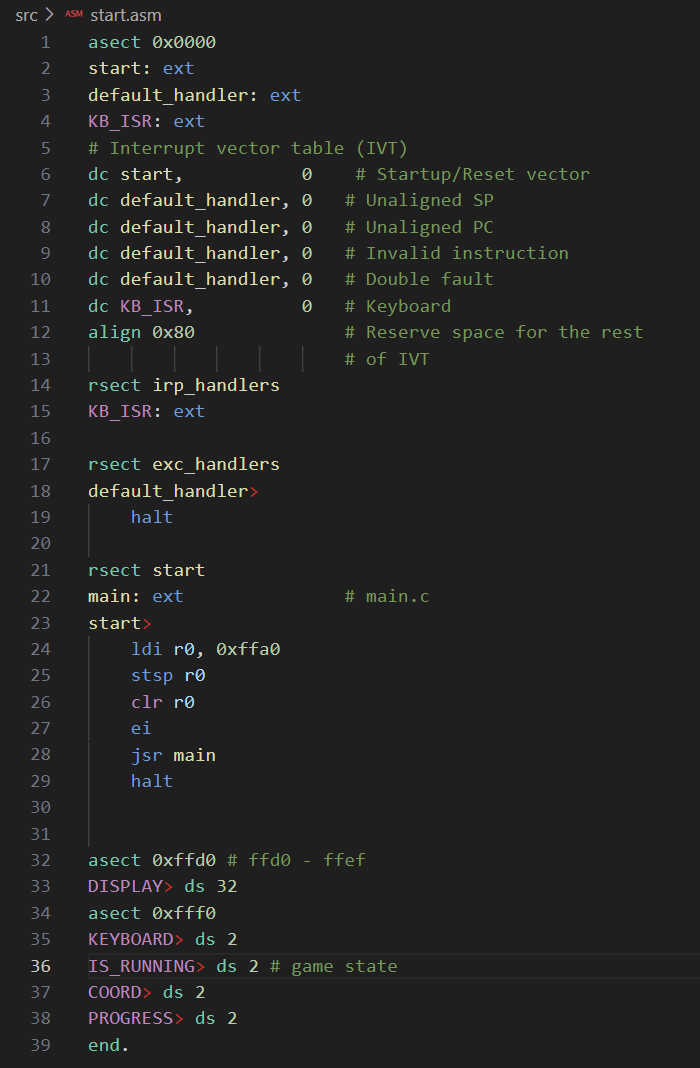
* Enter (0b0000000000000001): The current state of the cell in CUR\_READ (the buffer currently used for reading) is checked. If the cell is dead, it becomes alive, and vice versa. Since the CDM-16 processor does not support a multiplication instruction, a multiplication table mul6 is used for multiplying by 6. Multiplication by 4 is performed using a bit shift (>>2), and modulo 4 is done using a bitwise AND (& 0b11). This section also handles coordinate translation: it converts from a 2D coordinate system (x and y from 0 to 15) into the 1D array representation used in computation, which includes border dead cells and stores 4 cells per uint16\_t.
* Pause (0b0000000000000010): Toggles the IS\_RUNNING flag (start/pause simulation). The cursor is visible only in editor mode, depending on the state of this flag.

1. **print.c**

****

The print() function is implemented in **print.c**. It transfers the state of the cells from the field to DISPLAY — an array of 16 uint16\_t words, each corresponding to a row on the display. This function performs the reverse conversion of coordinates and data representation format.

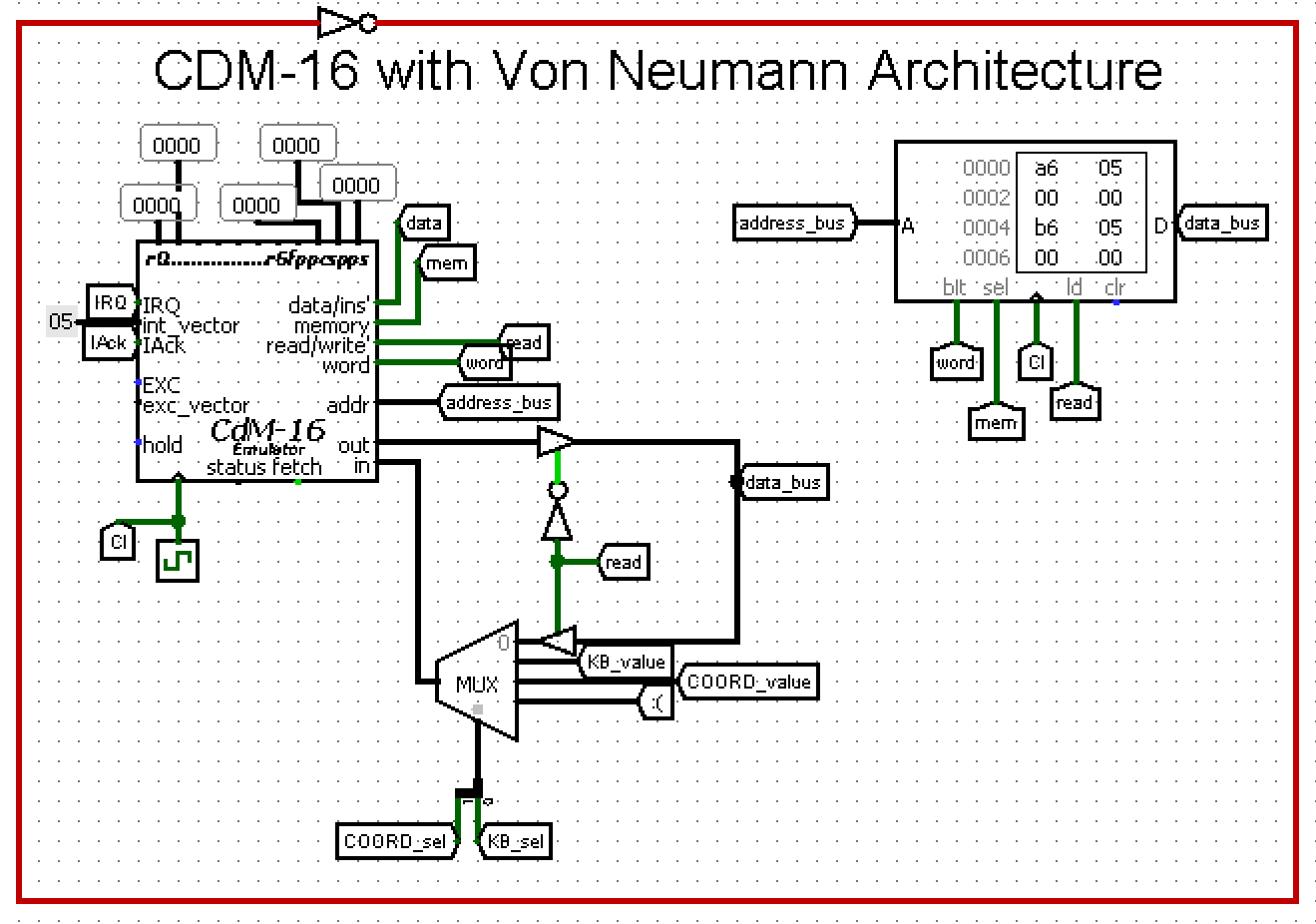
1. **start.asm**

****

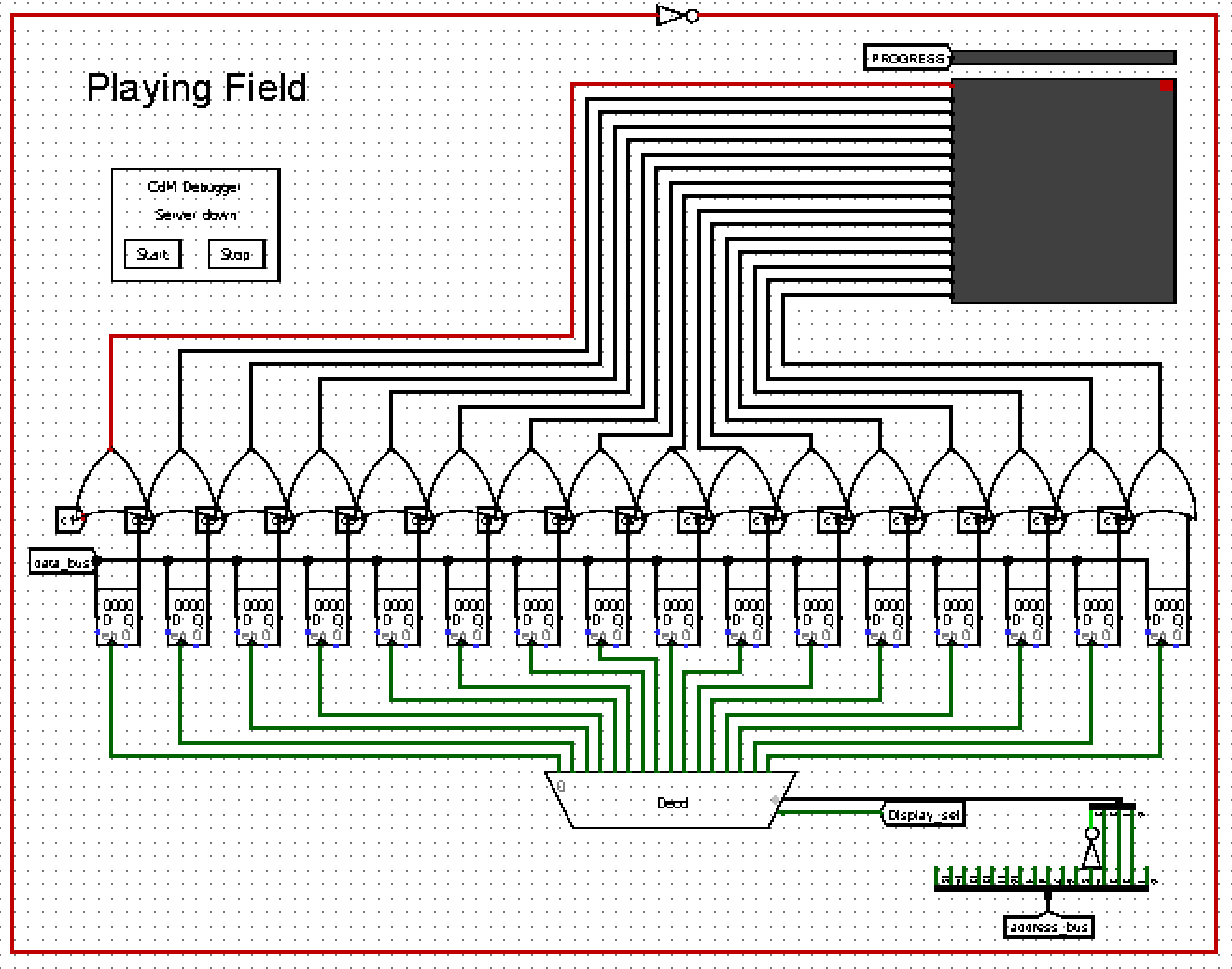
**start.asm** sets up the interrupt table, initializes SP, enables interrupts, declares service labels, and calls the main function.

**Hardware**

The game is implemented on the **CDM-16 with Von Neumann Architecture**.



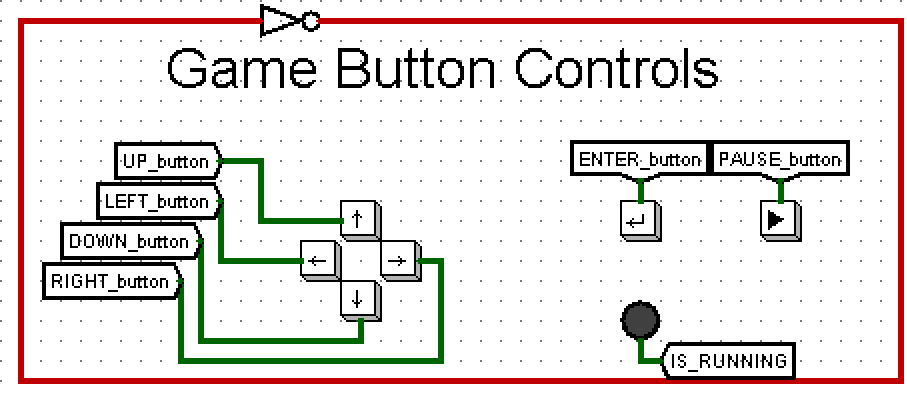
The game field is displayed using the schematic **Playing Field**.



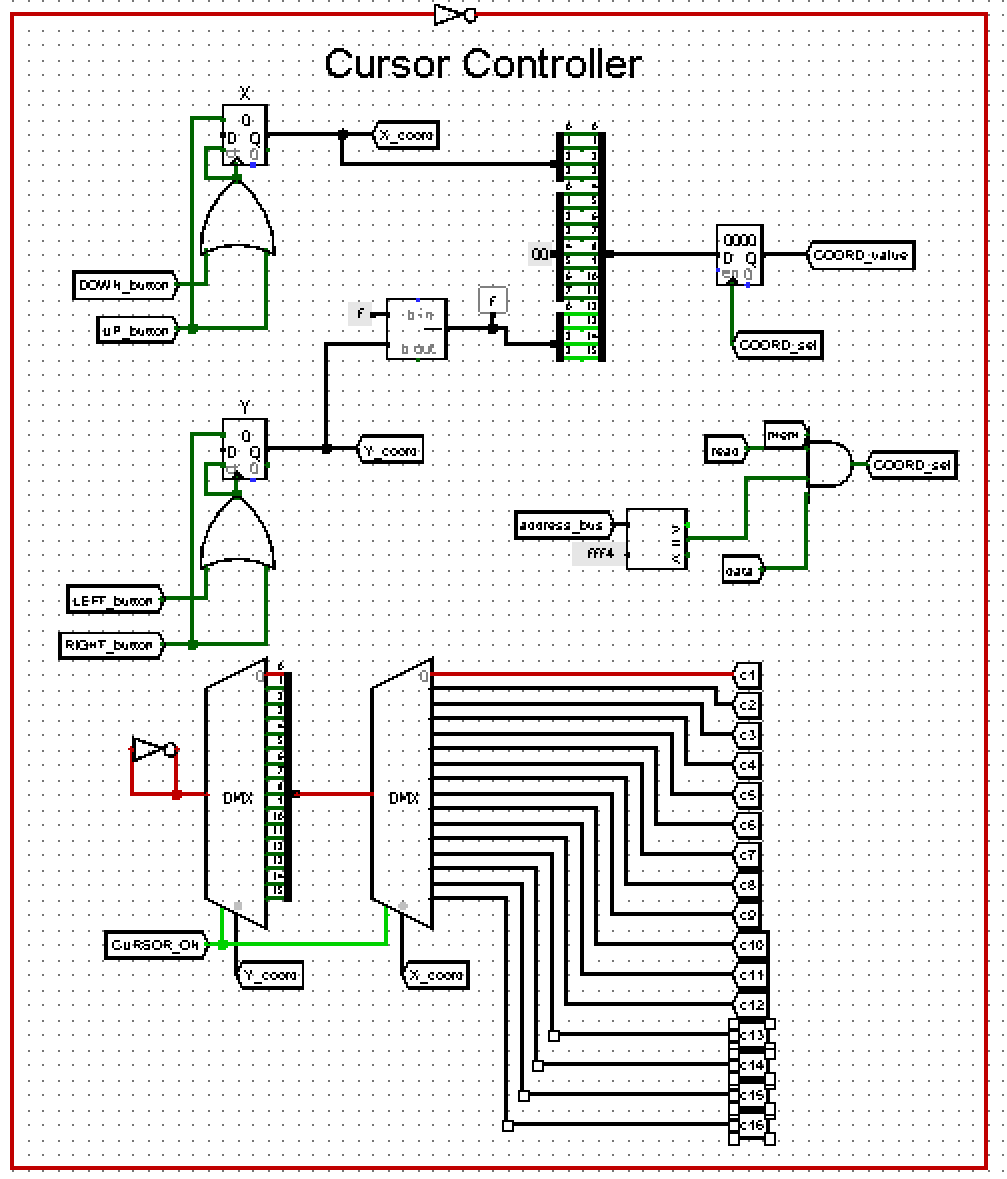
The **Game Buttons Controls** schematic handles cursor movement using the buttons: UP\_button, LEFT\_button, DOWN\_button, and RIGHT\_button. These do not interact with the processor.  
The ENTER\_button and PAUSE\_button do interact with the processor.

* The ENTER\_button toggles the state of the cell currently under the cursor.
* The PAUSE\_button starts or pauses the simulation.

The simulation stops only after completing the current cycle of computation and display.  
During the pause, once the field has been rendered, editing is possible.

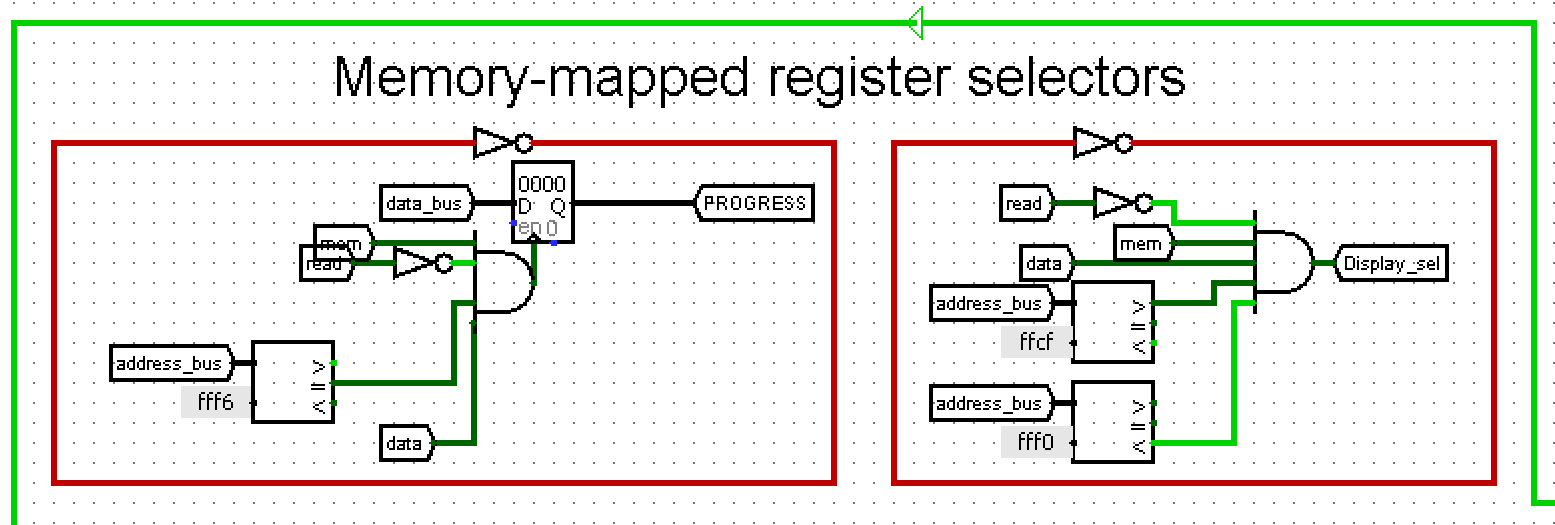


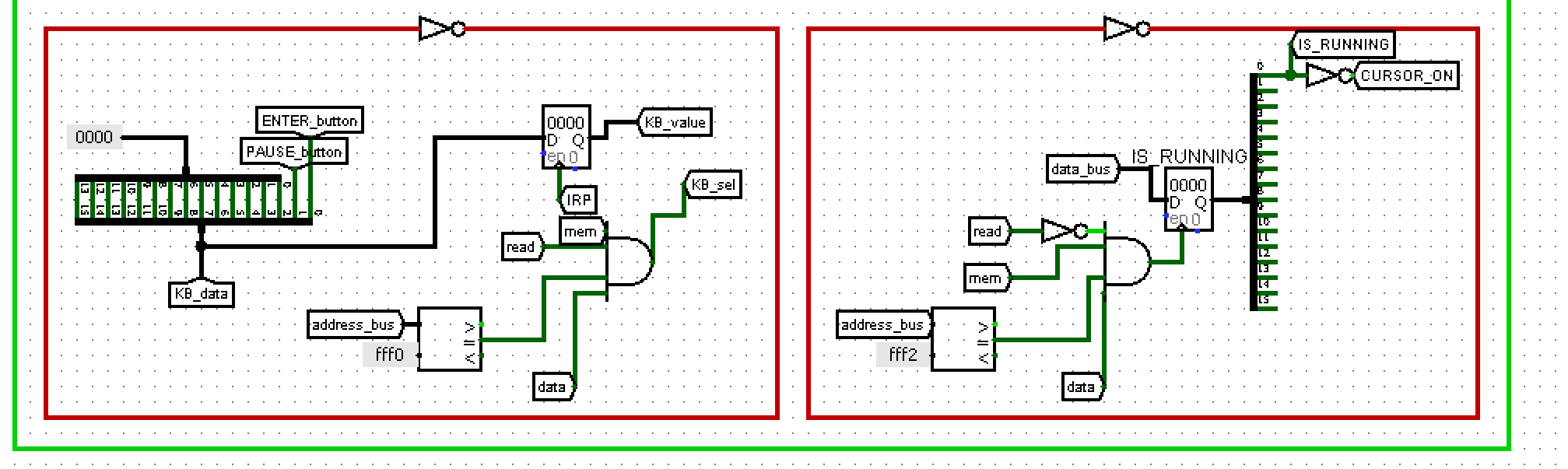
The **Cursor Controller** schematic is responsible for managing the cursor’s position and its display.  
Position control is implemented using two counters (for the X and Y coordinates), whose values are updated using the UP\_button, LEFT\_button, DOWN\_button, and RIGHT\_button. The cursor is displayed by sending an error signal through two demultiplexers.



The **Memory-mapped Register Selectors** schematic manages access to special input/output registers mapped to specific memory addresses.

* The **top-left block** updates the PROGRESS register, which is used to visualize the progress of rendering and field computation.
* The **top-right block** triggers output to a specific row of the display.
* The **bottom-left block** reads the value from the keyboard (KB\_value) when the ENTER or PAUSE buttons are pressed.
* The **bottom-right block** controls cursor display based on the state of mem[IS\_RUNNING].

****

****

The **Interrupt Controller** schematic is responsible for detecting keyboard events and initiating an interrupt request (IRQ) when the user presses a key.