Battleship

Magnus Hultin, Adam Johansson

Supervisor: Bertil Lindvall

Abstract

This project was made during the digital project course (EITF40) at Lund university.
The aim of this project was to construct and program the classic game of Battleship.
We also wanted to connect to another group’s game and compete with them. For this project we used a Motorola 68008 as the processor. The game would be displayed at a monochrome 128x64px display and the controller was a 16-keypad. The network would be over a serial port using SPI. The network part was never finished but the game was working.
We gained a lot of experience working and programming with microcontrollers and our understanding got a lot better. The choice of the Motorola 68008 was a big part of our learning of how the microcontroller's hardware is constructed.

Table of Contents

Introduction

For this project we chose to do the game Battleship and connect to another group and compete. We chose a Motorola 68008 as processor and had to connect the ram, rom, display and keyboard to get a working system. Because of this we could compare to the other groups Atmega 16 microcontroller. We wanted to get a deeper understanding of the design of a microcontroller and how to program one.

Specification of Requirements

For our project we will program the classic game Battleship for the processor MC6008. For our MC6008 we will need a screen, rom, serial channel, map unit and five buttons for controls. We will try to connect our game to group 3's AT16 via the serial channel to compete with them.

Hardware

Unlike modern microprocessors, the M68000 series lack internal memory and communication interfaces. The first challenge was thus to get the old 80's processor up to scratch to compete with an ATMega. For the memory we chose a 8 byte Eprom to store the coding and a 32byte Sram. A complete mapping of our finished system is available in appendix 1.

Since the only way to communicate with external devices was through the address and data buses, we needed a way to produce chip-select signals for our memories and user interfaces. This meant routing the address-bus through programmable logic gates, and in doing so effectively splitting the addressable area into different regions, making sure that when the processor tried to access a certain address only the correct unit would respond. The PRGs also had to control the external interrupts from the communication interfaces.

To be able to show the game we chose the LCD display of the type GDM-12864C. It is monochromatic, 128x64 pixel and has internal ram. The screen needs a synchronous valid memory address signal. Since the MC68008 does not supply one we had to make an external VMA generator to synchronize the address bus and processor to the enable signal. For this we used two SN74LS73 as in figure 4-2. (1)

The controller for the game is an keypad with 16 keys. It is wired to a keypad decoder that is asynchronous and triggers the interrupt level 2 in the processor.

Software

Our main program was coded using the C programming language, with some subroutines (turning on or off interrupts etc.) coded in assembler language. We wrote our code with a normal text-editor and compiled it with a provided makefile of sorts. While we were writing our program we used a development board made by our supervisor to simulate the behaviour of the M68008. With the development board we could address different components and test their functionality.

We wanted to implement a graphical user-interface, so we ended up making a library of pixelated tiles containing ships, mines, letters and other various graphical elements. These could then easily be drawn onto the screen with a single function call.

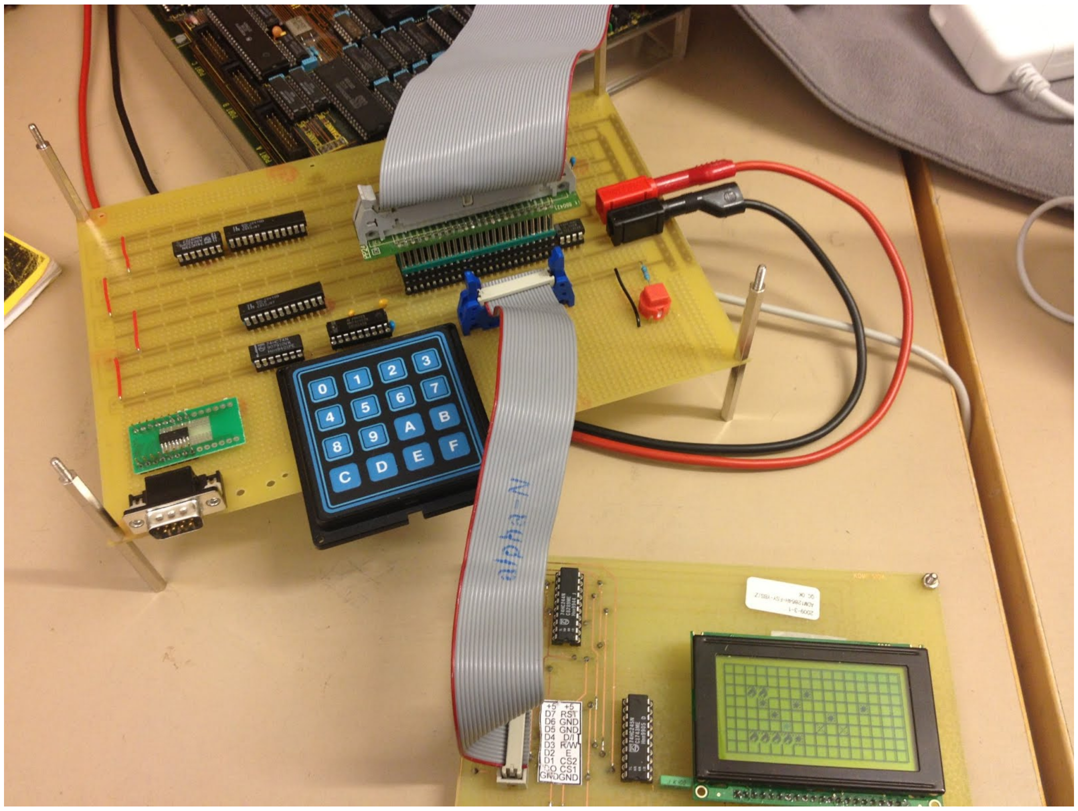
The keyboard inputs were handled by an interrupt routine that would be called whenever a key was pressed. The serial communication was thought to be handled in the same way, using a higher interrupt level, but it was never implemented.

Results

We did not meet our specification. The network communication was never implemented due to time constraints. The finished game was able to set the ships onto the playing field and then sink them using mines after giving the control over to another player. The resulting number of moves would then be printed on the screen. The network code was not programmed because the network interface was not implemented.

During initial testing, every component except the screen checked out. It turned out that unless both chip-select signals (the screen is split into two halves) could be pulled high at the same time the screen would be stuck in an idle-mode. This was impossible since our processor only could address one half at a time, but was quickly fixed by changing the on-board AND gate into a NOR.

When testing our code we ran into yet another problem with the screen. While inside our program, the screen would not accept the command to turn itself on, and the right half could not print data values higher than a certain point. This was a seemingly random behaviour since the screen performed perfectly during testing with the simulator. We never found an explanation to this problem, but it was solved by using the RAM-memory on the simulator instead of the one on our prototype.



Discussion

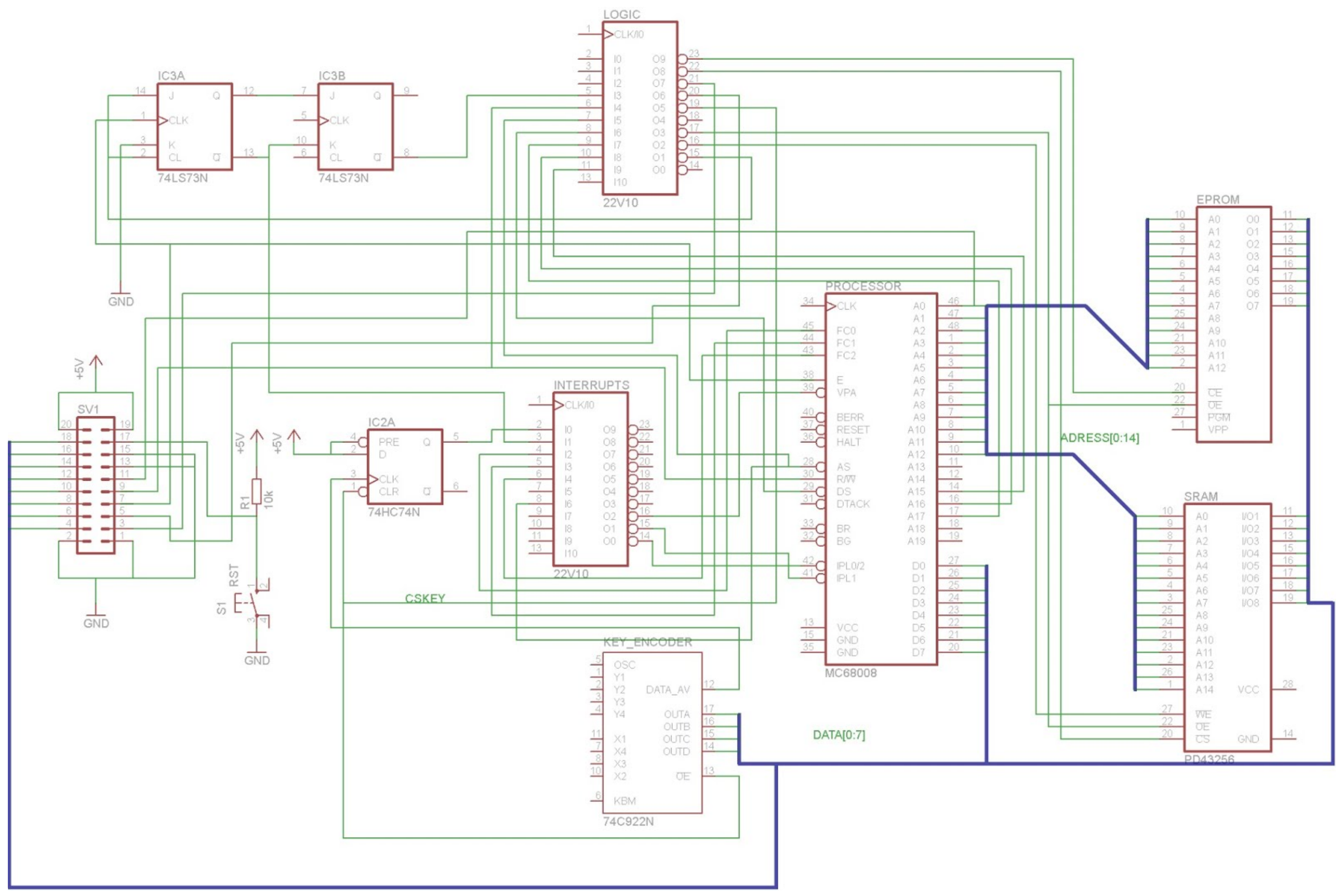
Even though the extra routing and hardware planning took a long time, it really payed off when it came to programming the processor. In our experience, the difficulty of programming modern microprocessors is getting the timing of the signals right since they use programmable pins for peripheral control. This was solved automatically by our hardware, and we could build our program like you would for a normal pc. When making simple games like Battleship or other classics, the need of a modern architecture is obsolete. In comparison, the Nintendo Game Boy system used a similar Zilog Z80 in its versions, up to the gameboy advanced, and it was by far the most commonly used handheld gaming device of its time.

Given more time a lot of improvements could be made, the most obvious the implementation of a communication interface for playing with other players. Other improvements could be evolving the game to include power-ups, moving boats, animations etc. The code is thought to be written in a way that it is easy to add new things into the game.

Reference

(1). Motorola . april 1985. *MC68008 8-/32-bit microprocessor with 8-bit data bus*. 1. edition.

Appendix 1



LOGIC
22V10
IC3A
74LS73N
IC3B
74LS73N
GND
PROCESSOR
MC68008
EPROM
SRAM
PD43256
ADRESS[0:14]
INTERRUPTS
22V10
IC2A
74HC74N
SV1
+5V
R1
10k
RST
S1
CSKEY
KEY_ENCODER
74C922N
DATA[0:7]

Appendix 2

Pseudo code for our executing program:

```
/* BATTLESHIP */

void wait(const int nbr) {
        /* waits */
}

int readCommand(const short int side) {
        /* used for checking if the screen is busy */
}

void sendCommand(const short int instruction, const short int side) {
        /* sends instructions to the screen */
}

uchar loadScreenData(const short int side) {
        /* loads the data shown on the screen */
}

void sendScreenData(const short int instruction, const short int side) {
        /* send data to bee shown to the screen */
}

void initDisp() {
        /* setups the screen */
}

void setGrid(const char c, const short x, const char y) {
        /* fills a grid at coordinates (x, y) with character c */
}


void emptyGrid() {
        /* clears the grid */
}

void update() {
        /* draws the entire grid to the screen */
}

int checkBorder(char c) {
        /* checks that the entire boat is on printable area */
}
```

```
int isBoat(const short int x, const short int y){
        /* checks if a the square is a boat-part */
}

int checkBoats(char c) {
        /* checks if the vicinity is free of boats so that boats cant be place on
        top of each other */
}

void addBoat(char c) {
                /* runs the check routines and correctly places boats into the grid */
}

void addMine() {
        /* checks if the grid square is a boat or mine and if hit checks if the boat is
sunk */
}

void resetVars() {
        /* resets global variables */
}

void game_on() {
        /* here the entire game routine is added */
}

exp2()
{
        /* listens for key presses */
}

exp5()
        /* can be used for SPI communication */
}

main()
{
        /* initiates variables and runs setup routines and game_on */
}
```