KING'S College LONDON

University of London

Design and Implementation of a 16-bit Breadboard Computer Architecture 6CCS3PRJ - Individual Project Report

Final Project Report

Author: Luca-Dorin Anton

Supervisor: **Dr. Christian Urban**

Student ID: 1710700

Programme of Study: **MSci Computer Science**

April 23, 2020

Abstract

Computers are becoming smaller, faster, more efficient and complex. At the same time, great advancements in both hardware and software provide both developers and end-users with increasingly higher levels of abstraction from the bare hardware. Whilst this allows computer users to focus on the task at hand and ignore any implementation details of the machine they are using which might get in the way, it also means that most people, including developers, software engineers and computer scientists are viewing computers as "magic black boxes". This project focuses on specifying, designing and building a simple and understandable Turing-complete machine architecture, as well as developing the necessary software tools to operate it, bridging the conceptual gap between silicon and lines of code.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Luca-Dorin Anton

April 23, 2020

Acknowledgements

I'd like to thank my supervisor, Dr. Christian Urban, for providing great mentorship, prodiving great ideas and suggestions and greatly heling towards keeping the goals of the project acheivable.

Contents

Chapter 1

Introduction

As the demand for high speed, low power, efficient and cheap computers rose over the past three decades, manufacturers invested heavily into improving production processes, shrinking transistors, pipelining instructions, creating new aggressive branch prediction models and implementing more and more functionality into the hardware directly. Moore's prediction on the number of electronic components doubling on the same surface area every two years held up well until recently when issues of quantum tunnelling started to arise. This hasn't stopped the continuous enhancement of microprocessor and microsystems though. Now, instead of making components smaller, manufacturers are installing more processing cores onto a single computing chip package.

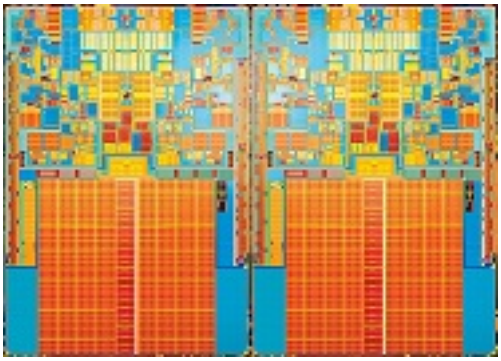


Figure 1.1: Intel quad-core 45nm CPU die

Another way of improving hardware is by implementing complex functionality, which would have been achieved traditionally through software, directly in hardware. An example of this is the implementation of the *Advanced Encryption Standard* (AES) by *Intel* directly in their lineup of CPUs through the *AES-NI* instruction set extension [29]. While the advantages for

modern society of the continuous and accelerated development of hardware cannot be doubted, there are also some worrying disadvantages. Such an advanced level of complexity in microprocessor design has been reached, that system and chip designers have started to increasingly rely on abstraction tools like *High-Level Synesthesis* to accommodate the advanced design requirements and meet user needs, as noted by Coussy in the *User Needs* chapter [8]. On the one hand, this increasing complexity of hardware poses challenges for operating system and compiler developers, who need to constantly stay up to date with the newest improvements in the hardware space and integrate them into their products, to ensure user satisfaction and sustained quality over time. One the other hand, people, including developers, are presented with no need to understand the underlying machines with which they are interacting. To quote Bruce Scheiner, a famous cryptographer and computer scientist:

> People don't understand computers. Computers are magical boxes that do things. People believe what computers tell them. [38]

As a possible solution to this general human trend towards treating computers as magical boxes", this project proposes a simple and understandable *practically implementable* machine architecture which has the same computational capabilities as a Turing machine. Some core features of the architecture are:

- 16-bit word length
- variable clock speed for live execution visualization
- single clock step function
- simplified input and output
- hardware addition and subtraction implementation

The rest of the report will go through the steps involved in specifying, designing, building and testing the machine and the software to go along with it in great detail.

1.1 Inspiration and Motivation

The main inspiration for this project is a YouTube series created by *Ben Eater* titled *Building an 8-bit breadboard computer!"* [5]. Eater's computer is itself a physical implementation of a theoretical architecture called *SAP-1*, which stands for *Simple as Possible*. There are three variants of the SAP Architecture, SAP-1, SAP-2 and SAP-3, in inreasing order of complexity, all of which have been created by Malvino and Brown in their book *Digital computer electronics* [33].

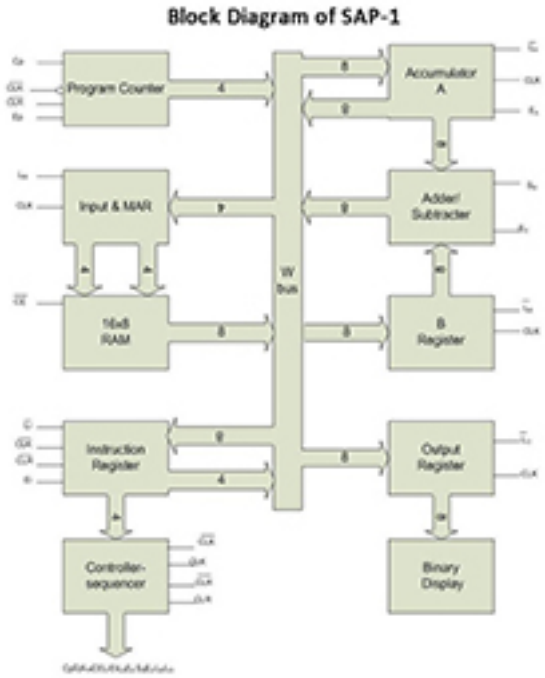


Figure 1.2: Block Diagram of the SAP-1 architecture

After going through the content, it became apparent that such a computer could serve as a great learning medium for developing a better understanding of computers, even more so with some hardware improvements and a sizable effort in writing some bespoke software for it. *This project is largely based around Eater's design.* It serves not only as the motivational source for the project, but it also provides a solid foundation for which extension, enhancement and improvement can be within the scope of a final year project.

1.2 Objectives

The main goal of this project is the physical implementation of a 16-bit computer system modelled after an architecture designed around the ideas of *explainability and ease of understanding* down to the transistor level. Breaking down this objective by specific hardware and software

requirements, the following can be stated: The main hardware objectives are:

1. 16 bit word length
2. appropiatley sized memory space
3. memory read/write capability
4. capability to decode and execute instructions sequentially
5. program counter alteration (jumps)
6. I/O functionality
7. hardware-implemented ability to perform basic arithmetical operations
8. simple branching
9. variable speed clock
10. single step clock function

The main software objectives are:

1. adequate microcode for the control mechanisms
2. assembly mnemonics
3. assembler package to turn assembly files into binaries
4. compiler for the WHILE-language to breadboard computer binaries

1.3 Project Structure

The report begins with an in-depth circuit specification, design and analysis literature review. These core skills lay the theoretical foundation necessary for understanding the reasoning for choices when designing the hardware. The next section is concerned with a detailed analysis of the computer built by Ben Eater [5]. Eater's computer serves as the template from which the design of the computer described in this report will originate. As such, it makes sense to analyse it carefully and classify its capabilities.

This will be followed by a specification of what the new computer should achieve in contrast to the capabilities of the existing computer.

The next chapter will cover the updated design of the computer. Important design decisions will be scrutinised and held against the main goals of the project. Both high-level, as well as in-depth design choices, will be taken into account. Main design challenges will be discussed and appropriate solutions presented.

After the design stage is complete, the following chapter will document the implementation phase. A detailed chronological breakdown of build progress will be presented. Testing will be executed in parallel with the implementation.

With the implementation phase complete, the software development phase will follow. The design and implementation of the various software tools necessary for running the breadboard computer will be documented in this chapter. With the software as well as the hardware ready, a rigorous testing phase will follow, coupled with an in-depth evaluation of the end product in comparison to the success criteria set at the start of the report.

Finally, the conclusion chapter summarises everything done so far and highlights the learning outcomes of the project and the possible continuation paths for future work.

Chapter 2

Background

Since this project is largely focused around designing and building a new hardware architecture, it is necessary to go through the existing material surrounding this topic. Hardware design can be structured in many different ways. For the purposes of this report, a structuring based on increasing abstraction levels will be used. Since the implementation objectives of the project aim to be educational in nature, it is crucial to start off with as few assumptions about the existing systems as possible. As such, the following explanations assume zero previous knowledge. Besides this, the educational outcomes largely focus on developers and computer scientists, professionals who would benefit from a better understanding of the computer but who are not necessarily familiar with the field of logic design. As such, the definitions and explanations will be kept as brief as possible, to avoid possibly superfluous levels of detail.

2.1 Computer Architecture

Computer architecture refers to organization, functionality and implementation design details of a computer system. It can be generally split into two main categories: *Instruction Set Architecture* and *Microarchitecture*.

2.1.1 Instruction Set Architecture (ISA)

The *Instruction Set* is the set of unique operations the computer is capable of performing. It is generally independent of the physical implementation of the system and it serves as an interface against which assembler, compiler and operating system designers and engineers can structure their products. ISA design will be a crucial part of this project. By building a hardware

architecture from scratch, the opportunity for many different ISA design choices will present itself. Many of those choices will be presented, implemented and analysed in this report.

2.1.2 Microarchitecture

A computing system's microarchitecture refers to concrete and detailed implementation choices for the hardware which is to implement the Instruction Set defined through the ISA. For the purposes of this project, the microarchitecture design will come first, which will be done against a general set of requirements and then the ISA design will follow based on the hardware design choices made. The ISA design will serve as a stepping stone towards the software architecture stage of the project.

2.1.3 General High-Level Architecture

Generally speaking, all computers share some high-level design features:

- *A Processor,* or *Arithmetic-Logic Unit (ALU),* which performs operations on some data
- *A Memory* or *Storage Unit,* which stores both data and instructions
- *A Control Unit,* which decodes instructions and issues control signals
- *Input/Output (I/O) devices,* to communicate with the outside world
- *A Clock Pulse Generator,* which keeps all other modules running synchronously
- *A Data Bus,* to facilitate the transfer of information between modules

Each individual component will be discussed in great detail in this report.

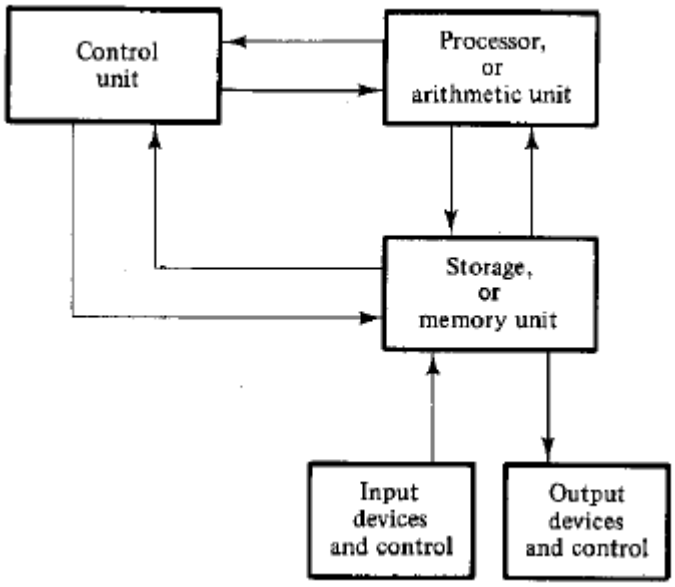


Figure 2.1: Block diagram of a digital computer, adapted from *Digital Logic and Computer Design* by M. Morris Mano [34]

Figure 2.1 is a block diagram based on the previous listing of modules. The Data Bus is represented as the double-ended arrows connecting the different components together. The clock is omitted.

2.1.4 The Processor

The processor module is tasked with executing certain operations on data. Its mode of operation is only dependent on two inputs: the data to be operated on and the operation to be applied to that data. As such, the processor does not have to hold any kind of internal state; its output will always be the same for a certain input. This makes the processor a *combinatorial circuit*, meaning that it just implements some (albeit complex) logic function and does not have any internal state or memory.

2.1.5 Memory

Memory serves the purpose of storing data and instructions and returning the stored information when requested. By nature, it is a *sequential* circuit, meaning that it has some internal state besides the logical function implementations used to communicate with the rest of the computer. Memory is organized in addresses, each address storing a word of information.

2.1.6 The Control Unit

The Control Unit oversees all other modules and ensures that everything is happening according to the present instruction. It also has the task of decoding the present instruction to correctly select the control signals which have to be issued next. There are two main design choices to be made when constructing a Control Unit. One option is to *hardwire* the logic. Whilst more efficient, this often proves to be tedious and very hard to alter. Another common and more accessible approach is the use of *microcode*. Microcode control units use some sort of *Read-Only Memory (ROM)* as a *lookup table* to decide which control signals to switch on at a given time step of a given instruction. This lookup table is microcode. As it is implemented through a ROM, it can be easily reprogrammed or swapped out for a different ROM, making the maintenance of the Control Unit much more accessible.

2.1.7 Input/Output Devices

Input devices are used to inject instructions and data into the computer. Output devices communicate calculated results back to the user. I/O devices can take many forms and usually also require the computer to implement some sort of *interrupt* system to notify the control logic that an external event is taking place. In the case of the computer built for this project, I/O devices will be abstracted as simple registers. The computer will read from and write to those registers.

2.1.8 The Clock Pulse Generator

Computers rely on a master clock to synchronise the activity of all other components. In modern microcomputers, this is usually accomplished through a *crystal osscilator* which vibrates at a predetermined frequency. There are also other ways to achieve a steadily pulsating clock signal. In the case of the computer built in this report, a pair of voltage comparators will be used.

2.1.9 The Data Bus

Given a large number of modules present in a computer, it comes off as impractical to have each module communicate with each other module directly. In this situation, the data bus presents itself as an adequate solution. All modules connect both their input and their output terminals to the bus through some guard or buffer which allows them to disconnect from the bus as needed. Then, at any given clock pulse, only one device is allowed to connect and output

to the bus, whilst any devices interested in receiving that information can connect and input from the bus. As long as only one device writes to the bus per clock cycle, the bus functions properly.

2.1.10 Other Important Components

Besides the main modules listed above, there are a few more components which are crucial to the optimal operation of a computer.

Registers

Registers are small memory units which can store only one word of memory. A computer system normally has a very limited amount of registers. In modern computers, registers have much shorter access times than memory. Besides storing data and programs, registers can have special functions, for example, input and output registers, registers tied to certain operations, flags registers and instruction registers.

Power Supply

Since the focus of this project is electrical computers, some sort of electrical power supply will be necessary. A simple solution for a power supply based on a standard ATX power supply will be presented in a later section of the report.

2.2 Implementing individual modules

With the general architecture of a computer system in place, the next design step is to create and implement designs for each individual module. This can be achieved by using circuit design theory and best practices, which are the tools" in the circuits designer's toolbox". An in-depth analysis of these can be found in the Appendix section of the report.

2.3 8-Bit Computer Architecture Implementation by Ben Eater [5]

This section concerns itself with the detailed analysis of the 8-bit computer architecture implemented by Ben Eater in his YouTube tutorial series [5]. This computer serves as the starting

design for the computer discussed in this report in the later section, as such, it presents itself as a good topic for discussion.

2.3.1 Main Features

The main features of Bean Eater's 8-bit computer on a breadboard can be broken down as follows:

1. 16 by 8 memory space (4-bit addresses, 8-bit words)
2. adjustable and manually single-steppable clock
3. two data registers: A and B
4. ALU which implements addition, subtraction and simple branching based on zero and carry conditions
5. Decimal output through three 7-segment displays
6. Microcode-based control logic using *EEPROMs* (electronically erasable and programmable read-only memory)
7. Common 8-bit data and instruction bus

2.3.2 High-Level Overview

Figure 2.2 is a high-level block diagram displaying the inner workings of the 8-bit computer built by Ben Eater[5]. The following observations can be made when observing this diagram:

1. Each module is connected directly to the common data bus. This means that every module can output information to the bus each clock cycle. After a judicious inspection of the microcode[3] it is clear that no two modules output to the bus at the same time.
2. The clock signal and the inverse clock signal are distributed throughout the computer to each module. Also, notice how there is a control signal for the clock as well. This *HLT* (Halt) signal allows the computer to halt the clock, and implicitly halt the execution, for example after finishing a calculation, to allow it to be displayed.
3. Besides the main connections through the bus, there are some additional *special connections* between certain modules. For example, the *A register* and the *B register* have

a direct connection to the *ALU*, the *MAR* (Memory address register) has a direct connection to the *RAM* (Random Access Memory) and the *IR* (Instruction Register) has a direct connection to the control logic.

4. All control signals originate from the *Control Logic* and spread out throughout the computer. Each module has at least one control signal

5. This computer is severely limited in terms of memory. While 16 bytes can be sufficient for some demonstrational trivial programs (like Factorial or Fibonacci), it is insufficient for anything else.

6. Another major limitation is the fact that it can only operate on signed integers. The architecture represents data in *big endian, two's complement integer* format. No other data format or type is supported.

2.3.3 Module Design Conventions

Ben Eater follows some essential design conventions when designing and building each of the modules for his computer. The following subsections describe those conventions.

Simplicity of understanding over cost and efficiency

In many situations where a simpler solution from cost or efficiency makes itself notices, Eater often chooses to go for a more pragmatical approach which focuses on the simplicity of understanding. Since his computer mostly serves as an educational tool, it makes sense to pursue solutions which are easy to understand, over solutions which might be slightly cheaper or more efficient. A good example of this is in the design of the clock module 2.3.

For the combinatorial circuit responsible for selecting a clock signal (either the automatic or the manual one) and also filtering out the clock when the *HLT* (Halt) signal is active, Eater could have opted for a circuit built out of *NAND* (Not And) gates instead of a circuit of AND, OR and inverter gates. This is because NAND gates are universal gates, which means that any combinatorial circuit can be built exclusively out of NAND gates. In this case, this would have had a net effect on cost, since the circuit could have been implemented with only two NAND ICs (integrated circuits), instead of three. The choice was made to use AND, OR and Inverter gates since the function of those gates is more intuitive and as such, the entire circuit is easier to understand.

Connection to the Bus

Eater's computer features an 8-bit common bus for both data and instructions. Most modules are tied to this bus directly. For the bus to function properly, only one device should be allowed to output to the bus at a time. Without some guards, connecting to the bus directly would mean that all modules would inadvertently drive the bus either high or low, depending on their output. The solution to this is the use of *Tri-State buffer gates.* This gates can be set to be in three states, either on or off, depending on the signal passing through them, or in a *high-impedance* state in which the two terminals of the buffer are essentially disconnected from each other. This is activated through a separate control signal. All modules which output to the bus do so through an IC (integrated circuit) containing such gates. An example of this can be seen on the A register 2.4.

2.3.4 Analysis Conclusions

Based on the previous analysis we can draw the following conclusions about Eater's approach towards implementing SAP-1:

- When faced with a choice between efficieny/cost or simplicity, simplicity should always be chosen, since the purpose of the build is academic/educational in nature.
- Each module of the computer should follow the following pattern: data inputs and outputs come and go from the common bus, while control signals come directly from the *Control Logic* module.
- Each module should execute one function and do it well. This is very similar to the Unix Programming Philosophy.

Taking all of these finding into account, the next step is to set out the specifications for the enhanced computer which is to be built in this report and then come up with designs matching those specifications.

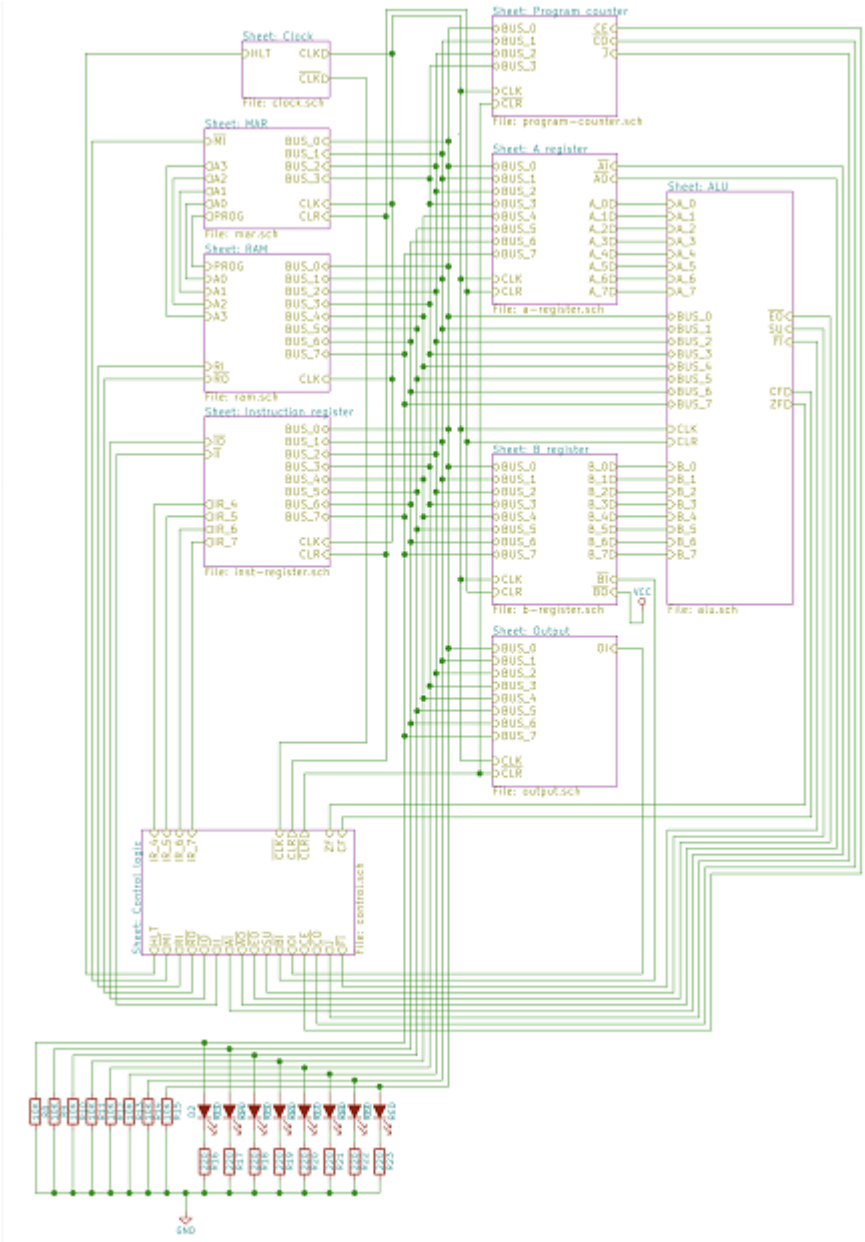


Figure 2.2: Block diagram of the 8-bit computer built by Ben Eater [4]

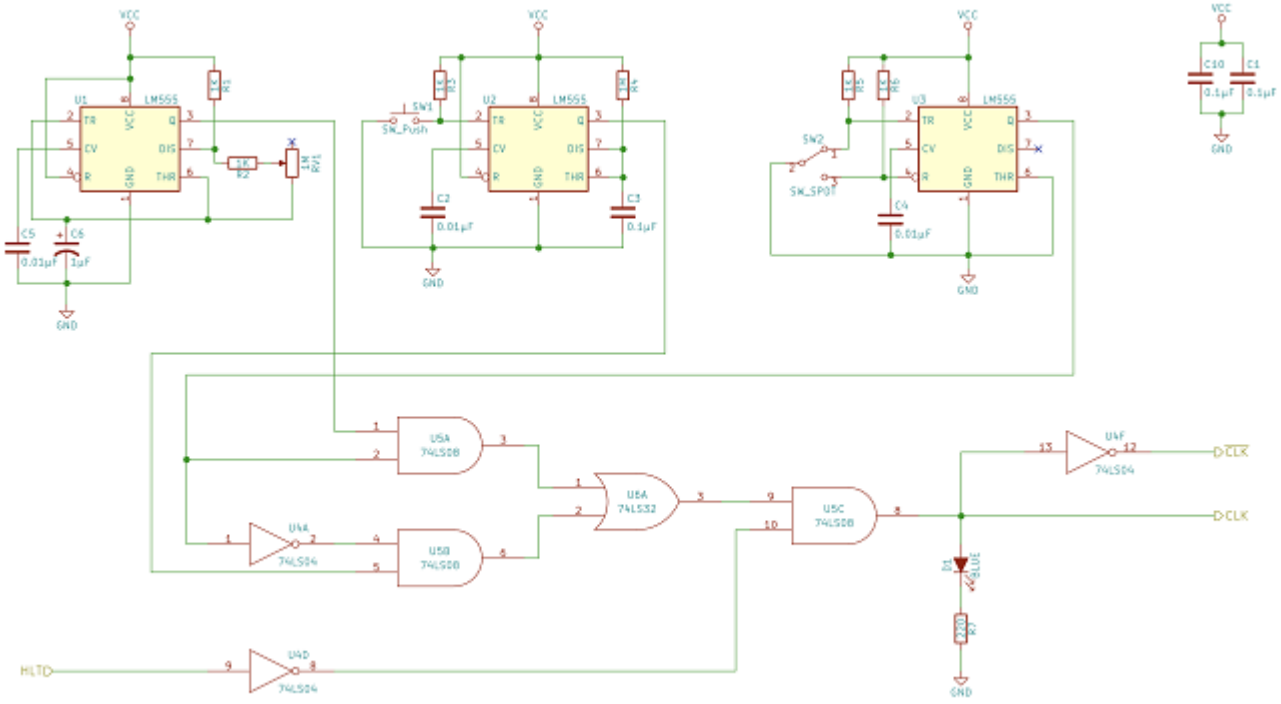


Figure 2.3: Schematic of the clock module in Bean Eater's 8-bit computer [4]

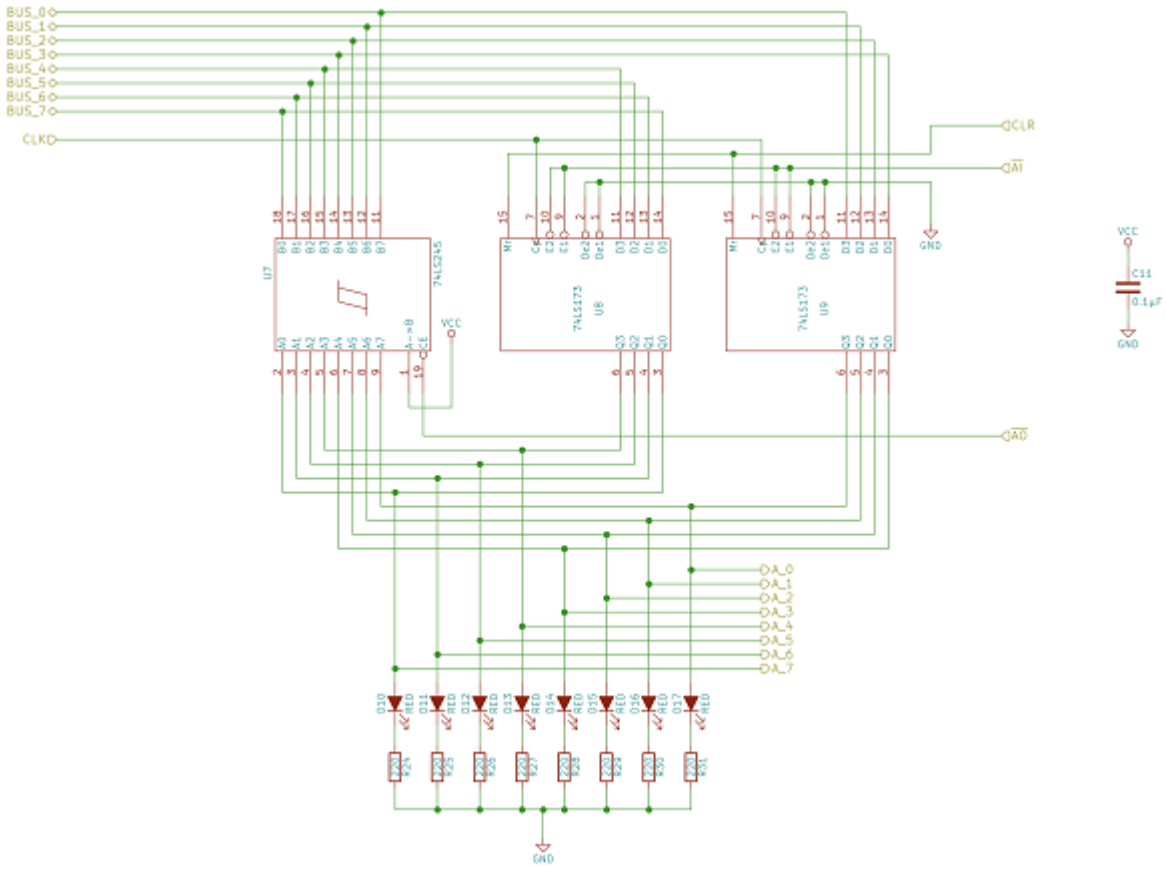


Figure 2.4: Schematic of the A register in Bean Eater's 8-bit computer [4]

Chapter 3

Hardware Specification & Design

This chapter of the report focuses on creating a comprehensive set of specifications for the computer to be built as a part of the project and then provides a detailed listing of the design choices made to satisfy the specifications provided. Since the design will be based on Bean Eater's 8-bit breadboard computer [5], but it will also include parts of SAP-2 and SAP-3 [33]. Most specification criterions will be phrased as additions and enhancements to the existing designs.

3.1 Specification Guidelines

The specifications which are about to be presented serve the purpose of adding functionality to the 8-bit breadboard computer such that its educational potential is harnessed more effectively, while at the same time avoiding over-complication and over-extensions of scope. As such, it makes sense to list some of the features which will *fall out of the scope* of this build for practical and time considerations.

- Interrupts and Interrupt handling
- Processes (The computer will only run one process, there will be no threading interface/ no operating system)
- Floating-Point operations
- Support for any kind of advanced in-hardware operations (for example encryption)
- Native support for signed integers over 16 bits

- Graphical User Interfaces (GUIs)
- Input through traditional peripheral (mouse and keyboard)

3.2 Major Architecture Changes

The computer should broadly follow the architecture o the 8-bit computer designed by Malvino and Brown [33] and built by Bean Eater [5]. The major architectural difference should be *the extension to 16-bit words*. Since the word length of the computer should be 16 bits, its bus and most of its modules should be extended to accommodate this extra capacity.

3.2.1 Operational Enhancments

Besides Addition and Subtraction, the computer should also implement *bit-shifting* (both left and righ). Bit-shifting is a crucial operation which is very often performed to increase the efficiency of certain operations (for example, multiplication and division by 2 can be expressed in binary as a left or a right shift of 1 bit)

3.2.2 I/O Enhancements

Currently, the only way to provide input to the 8-bit computer is by *manually* programming each memory address through dip-switches. This is slow, clunky and prone to errors. There should exist a mechanism to quickly program the computer, for example through an external *microcontroller* like an Arduino. Besides this, there should also exist a way for the computer to request input *while executing* from an external device like a microcontroller. Similarly, to provide persistence to the values from calculated by the computer, instead of being able to display only one value at a time, the computer should also have the ability to communicate with an existing external device like a separate microcontroller, providing it with the values it has calculated. In turn, the microcontroller a the be connected to a regular personal computer and then programmed to display those values to the screen. Besides this, the computer should have a display capable of displaying more than 4 characters and more than just numbers.

3.2.3 Stack Operations

The addition of a stack pointer would make subroutines, calls to subroutines and callbacks significantly easier. An effective stack pointer just has to have the option to increment and

decrement, in contrast to a simple program counter which just increments. With this functionality, return addresses can be pushed on and popped off of the memory stack whenever they are required.

3.2.4 Expanded Random Access Memory

Eater's implementation has a very limited address space (4 bit address, which equates to 16 addresses). While from a theoretical point of view, this is as close to a Turing Machine as a supercomputer with terrabytes of RAM (an ideal Turing Machine should have infinite memory), a more reasonable amount of memory (in the kilobytes range) would allow for far greater flexibility in software applications. For this computer, a 16K memory ($2^10 * 16$, or 10 bit address space) should be sufficent.

3.2.5 Memory Layout

Due to the added features, the memory layout of the 16-bit breadboard computer needs to be re-worked. As such, the following layout is proposed:

- From 0x000 to 0x1FF: Program Text
- From 0x200 to 0x3EF: Variables and Data
- From 0x3F0 to 0x3FF: Stack

3.3 16-bit Breadboard Computer Layout

Based on the previous specification, the computer to be built in this report should contain the following modules and have the following physical layout:

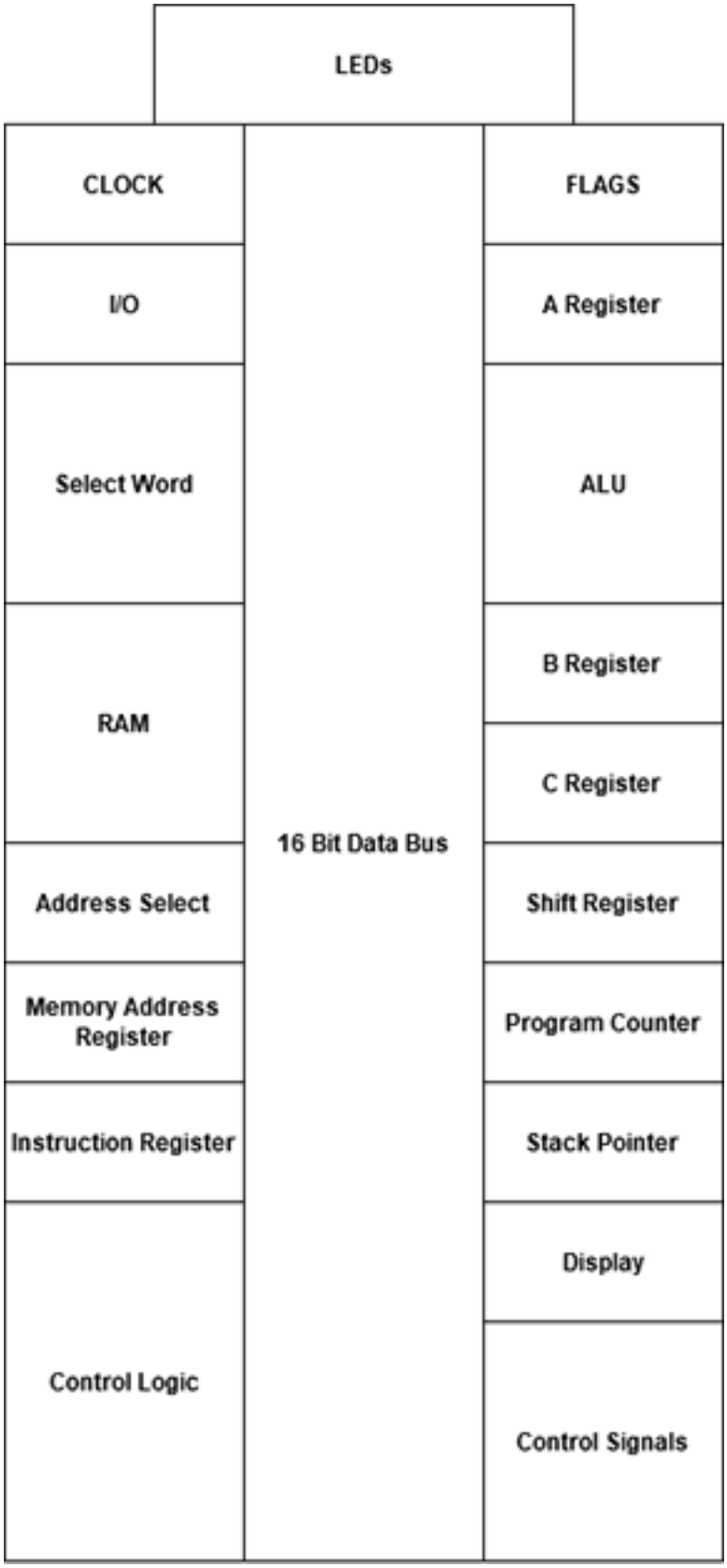


Figure 3.1: High Level Module Overview and Layout of the 16-bit Breadboard Computer

The updated computer specification contains the following modules, each of which will be followingly discussed in later sections:

1. Common Data Bus 3.3.1
2. Data Bus LEDs 3.3.2
3. Clock 3.3.3
4. A Register 3.3.4
5. B Register 3.3.5
6. Arithmetic-Logic Unit (ALU) 3.3.6
7. Flags Register 3.3.7
8. C Register 3.3.8
9. Shift Register 3.3.9
10. Random Access Memory (RAM) 3.3.10
11. Program Counter 3.3.11
12. Stack Pointer 3.3.12
13. Display 3.3.13
14. I/O 3.3.14
15. Memory Address Register 3.3.15
16. Instruction Register 3.3.16
17. Word Selector 3.3.17
18. Address Selector 3.3.18
19. Control Logic 3.3.19
20. Control Signals 3.3.20

3.3.1 Common Data Bus

The Common Data Bus acts as the communication medium of the computer. Drawing a parallel between *computer design* and *human anatomy*, the data bus acts like the circulatory systems. It ensures all other components are connected and can talk to each other. The Common Data Bus should be 16 bits wide, since the final system should have a word length of 16 bits. Since all other modules connect to the Data Bus, it makes sense to have it located centrally, between all other modules. This way, no module has to have particularly long wires to interface with the Bus.

3.3.2 Data Bus LEDs

This modules is as simple as it gets. It should be composed of just 16 LEDs, or Light Emitting Diodes, one on each Data Bus line, so that whatever gets asserted at a certain point in time on the Bus can be visible. Besides this, it can also contain pull-down resistors to ensure that, if no module asserts anything on the bus, it should default to low, or a logic 0.

3.3.3 Clock

The clock acts as the heart of the system. It provides a pulse under the form of a square wave, based on which all other components do their job. The clock signal should be distributed to all other modules. Besides providing the square wave clock pulse an inverse, or counter clock pulse should be available, as well as the ability to change the frequency and to completeley stop the square wave and replace it with a manual button press for demonstrational and debugging purposes. Finally, the clock should take in one signal as input, namely halt, or *HLT*, which should completeley stop the clock regardless of operating mode. This will allow the computer to halt execution if, for example, the program is finished.

$InputControlSignals : HLT$

$OutputControlSignals : CLK, \overline{CLK}$

3.3.4 A Register

The A register is a 16-bit memory storage module. It should implement three functions. First, when its AI (A Register In) signal goes high, it should latch in the contents of the data bus on the rising edge of the next clock pulse. The second function is to output its contents to the bus when its $\overline{AO}$ (A Register Out) signal goes low. Finally, if the $\overline{RST}$ (Reset) signal goes low, it

should clear out its contents and latch in a 0 on all bits. Additionally, the A register should have a direct 16-bit connection to the *Arithmetic Logic Unit*, or ALU 3.3.6.
$InputControlSignals : CLK, AI, \overline{AO}, \overline{RST}$ $DirectConnectionto : ALU$

3.3.5 B Register

Similarly to the A Register, the B Register should store 16 bits of data from the bus. It should have similar control signals, *BI, BO* and *RST*, which perform equivalent functions. It should also have a direct 16-bit connection to the *ALU* 3.3.6. $InputControlSignals$: $CLK, BI, \overline{BO}, \overline{RST}$ $DirectConnectionto : ALU$

3.3.6 Arithmetic Logic Unit

The Arithemtic Logic Unit, or ALU, is the module responsible with data procesing. It takes the contents from*both A and B registers* 3.3.4 3.3.5 directly and adds them up. Is the *SU* (Subtract) signal is provided, the contents of the B Register 3.3.5 will be subtracted from the contents of the A register. If the $\overline{\varepsilon O}$ (Sum Out) is taken low, the contents of the ALU will be asserted on the data bus. The ALU also has a direct connection to the Flags Register 3.3.7. Over this connection, the ALU should provide three flags:

- Parity Flag: whether the *Least Significant Bit* is 0 or 1
- Zero Flag: wheter the content of the ALU is 0
- Carry Flag: wheter the result of the operation of the ALU is cannot be expressed within 16 bits

$InputControlSignals : \varepsilon O, SU$ $DirectConnectionto : ARegister, Bregister, FlagsRegister$

3.3.7 Flags Register

The Flags Register is essential towards ensuring the Turing completeness of the computer being built. Based on the state of the flags, the computer can make branch decisions, which reflect the fact that Turing machines can make selective decisions based on the symbol it has just read. Essentially, the flags register serves the purpose to latch in the three flags provided by the ALU 3.3.6: the Parity Flag, the Zero Flag and the Carry Flag. When the *FI* (Flags In) signal is taken high, on the next clock pulse it should latch in the contents of those flags. Besides this, the Flags Register should clear its contetns when the $\overline{RST}$ (Reset) signal is taken low. There

is a direct connection between the Flags Register and the Control Logic module, as the flags play a role in deciding what to do next.

$InputControlSignals : CLK, FI$ $DirectConnectionto : ALU$

3.3.8 C Register

The C Register serves as a general purpose 16-bit register. It has signals similar to the other two registers, A 3.3.4 and B 3.3.5. When *CI* (C Register In) goes high, on the next clock pulse the register should latch in the data word asserted on the bus in its storage. IF $\overline{CO}$ (C Register Out) goes low, the register should assert its contents on the data bus. IF $\overline{RST}$ (Reset) goes low, it should clear its contents and latch in onyl zeroes. There should be no direct connection between the C register and any other registers.

$InputControlSignals : CLK, CI, \overline{CO}, \overline{RST}$

3.3.9 Shift Register

The Shift Register is the other module of the 16-bit breadboard computer capable of conducting data processing tasks. First and foremost, it should act as a normal register, so it should latch in the bus contents on the next clock pulse if the *SI* (Shift Register In) signal goes high, asserts its contents to the bus if $\overline{SO}$ goes low, and clears it contents if $\overline{RST}$ goes low. Besides this, if *SFL* (Shift Left) goes high, on the next clock pulse it should shift its contents to the left by one bit and insert a 0 at the least significant bit of the data word. Similarly, if *SFR* (Shift Right) goes high, on the next clock pulse it should shift its contents one bit to the right and insert a 0 at the most significand bit of the data word.

$InputControlSignals : CLK, SI, \overline{SO}, SFL, SFR, \overline{RST}$

3.3.10 Random Access Memory (RAM)

Random Access Memory, or RAM, is the equivalent of the tape on a Turing machine. It can be read from and written to, and it should be large enough to accommodate algorithms, data and variables; all at the same time. Memory is organised in addresses. Each address should store one 16-bit word of memory. A 10-bit address is deemed to be sufficent, so the computer should have a $2^10 * 16 = 16K$ memory space. The RAM module should also have a direct connection to the *Address Selector* 3.3.18 and to the the *Word Selector* 3.3.17. The Address Selector serves a 10-bit address to the memory, while the word selector serves a 16-bit data word to the RAM. If the *RI* (RAM In) signal goes high, on the next clock pulse the RAM Module should latch

in the data word served by the word selector at the address provided by the address selector. If the RO signal goes low, the RAM module should assert the data word stored at the address provided by the address selector on the data bus. Additionally, based on two control signals originating from toggle switches, $PROG$ and $ARDUINO$, the signal which governs the RAM writes can be changed. If $PROG$ is low, that means the computer is in run mode, and the RI signal controls writes. If $PROG$ is high, then the computer is in programming mode, and the write signal is chosen based on the $ARDUINO$ signal. If $ARDUINO$ is low, then the writes are manually controlled using a simple push button. If it is high, then an external *Arduino Mega* [13] controls the writes to memory using a signal called $ARDUINO_WRITE$.

$InputControlSignals : CLK, RI, RO, PROG, ARDUINO, ARDUINO_WRITE$

$DirectConnectionto : WordSelector, AddressSelector, ArduinoMega$

3.3.11 Program Counter

The Program Counter serves the purpose of keeping track of the current address which should be executed. As such, it should be a 10-bit register, to ensure coverage of the entire address space of the computer. If the CE (Counter Enable) signal goes high, on the next clock cycle the program counter should count up one in binary from the value it has currently latched in its storage and then latch this new value in. If the $\overline{JMP}$ (Jump) signal goes low, on the next clock pulse the program counter should latch in whatever value is asserted on the 10 loweest bits of the bus in its register. If $\overline{CNT_O}$ (Counter Out) goes low, the value latched in the program counte should be asserted on the data bus. If $\overline{RST}$ (Reset) goes low, the program counter should clear out its contents and latch in zeroes on all bits.

$InputControlSignals : CLK, CE, \overline{JMP}, \overline{CNT_O}, \overline{RST}$

3.3.12 Stack Pointer

Tha stack pointer should essentially be a register holding a 10-bit address, but accepting only a small subsection of the address space (from 0x3F0 to 0x3F). This means that the stack should be 16 addresses tall. If the $\overline{ST_I}$ (Stack Increment) signal goes low, on the next clock pulse, the stack pointer should increment by one. If $\overline{ST_D}$ (Stack Decrement) goes low, the stack pointer should decrement by one on the next clock pulse. On $\overline{ST_J}$ (Stack Jump) going low, on the next clock pulse the stack pointer should latch in the address asserted on the bus in its register, assuming it is a valid stack address. If $\overline{ST_O}$ goes low, the stack pointer should assert the address it has stored out on the bus. Finally, if $\overline{RST}$ (Reset) goes low, the stak pointer should reset to

zero.

$InputControlSignals : CLK, \overline{ST_I}, \overline{ST_D}, \overline{ST_J}, \overline{ST_O}, \overline{RST}$

3.3.13 Display

The Display module is the main way through which the computer can interface with the user. It should have the ability to display both alfanumeric characters and digits and it should feature at least 2 lines of 16 characters each. When the OUT (Output) signal goes high, the display should take a word of data from the bus and interpret it as the next character to be written to the screen.

$InputControlSignals : OUT$

3.3.14 I/O

Besides a display, the computer should also feature a bidirectional interface to another miro-controller. In this case, the interface should be to an *Arduino Mega* [13]. If the $\overline{E}$ (Enable) signal goes low, based on the $R/\overline{W}$ (Read/Write) signal, the computer should interact with the Arduino. If the $R/\overline{W}$ signal is high the computer should read a word of data from the arduino, so that word of data should be asserted on the bus. If the $R/\overline{W}$ signal is low, the Arduino should read a word of data from the bus (and consequently displayed to the user).

$InputControlSignals : \overline{E}, R/\overline{W}$

$DirectConnectionto : ArduinoMega$

3.3.15 Memory Address Register

The memory address register, or MAR, is a 10 bit register. It connects directly to the address selector **??** and there is no other way to get the information latched into it. If the MI signal goes high, on the next clock pulse the MAR will latch into its storage the 10 least significant bits asserted on the bus. if $\overline{RST}$ goes low, the MAR will clear its contents and write zeroas to all 10 bits.

$InputControlSignals : CLK, MI, \overline{RST}$

$DirectConnectionto : AddressSelector$

3.3.16 Instruction Register

The Instruction Register, or IR, is a 16-bit register which holds the next instruction to be processed. It is special in that its most significant 6 bits have a different meaning. They

represent the opcode for the current instruction and are directly connected to the Control Logic module 3.3.19. When the II (Instruction Register In) signal goes high, on the next clock pulse the instruction register should latch in the data word asserted on the bus in its storage. If the $\overline{IO}$ (Instruction Register Out) signal goes low, the register should assert its least significand 10 bits to the bus. If $\overline{RST}$ (Reset) goes low, the instruction register should reset and latch in zeroes on all 16 bits.

$InputControlSignals : CLK, II, \overline{IO}$

$DirectConnectionto : ControlLogic$

3.3.17 Word Selector

The Word Selector is a special module, in that it serves the purpose of selecting between different sources of data and feeding them into the RAM module 3.3.10. There are three possible sources of data

1. The Data Bus
2. Dip Switches
3. An Arduino Mega

The Data Bus is the first and most straight forward data source. A 16-bit data word can be taken from the bus and then passed on to memory. The second source are Dip Switches. Dip Switches are rows of small binary switches, which can be used to manually feed binary data into a digital system. The last possible source of data is an arduino mega. The choice of data to be fed forward is governed by two control signals, $PROG$ and $ARDUINO$. If $PROG$ is low, data from the data bus will be selected, regardless of the state of $ARDUINO$. If $PROG$ is high, that means that the computer is in programming mode. In this mode, the data source depends on the $ARDUINO$ control signal. If this signal is high, then data will be fed forward from an external Arduino Mega [13]. If it is low, then a series of 16 dip switches will be used to manually program the computer.

$InputControlSignals : PROG, ARDUINO$

$DirectConnectionto : RAM, ArduinoMega$

3.3.18 Address Selector

Similar to the Word Selector 3.3.17, the Address Selector selects between different address sources to provide a 10-bit address to the RAM module 3.3.10. There are three possible data

sources:

1. The Memory Address Register 3.3.15
2. Dip Switches
3. An Arduino Mega

The choice of data source is defined by two control signals, *PROG* and *ARDUINO*. If *PROG* is low, then the computer is in run mode and the address latched in the *MAR* 3.3.15 is fed forward to RAM. If it is high, then that means the computer is in programming mode and the address choice is governed by the *ARDUINO* control signal. If it is low, then the memory address will be manually chosen using a series of 10 Dip Switches. If it is high, then an external Arduino Mega **??** will be the address source.
$InputControlSignals : PROG, ARDUINO$ $DirectConnectionto : RAM, ArduinoMega$

3.3.19 Control Logic

To draw another parallel to human anatomy, the control logic module can be thought of as the brain" of the computer. It takes in the 6-bit opcode from the Instruction Register 3.3.16, the 3 flags from the Flags Register 3.3.7, as well as a 3 bit number representing the step" of the current instruction which is to be executed and decides based on these pieces of information which control signals to keep active on the next clock cycle. It also maintains an internal 3 bit counter which counts on the inverted clock or counter clock signal to provide the 3-bit step.
$InputControlSignals : \overline{CLK}$ $OutputControlSignals : HLT, \overline{RST}, AI, \overline{AO}, BI, \overline{BO}, \overline{\varepsilon O}, SU, FI, CI, \overline{CO}, SI, \overline{SO}, SFL$
$Directconnectionto : InstructionRegister, FlagsRegister$

3.3.20 Control Signals

The Control Signals Module is used to visualise which control signals are active at any given time. This module should essentially consist of a labeled LED on each control signal.

3.4 Specification Conclusion

This concludes the specification phase of the project. The next step is to design each module to this specification.

3.5 Design

This section of the report focuses on the design process for the individual modules of the 16-bit breadboard computer. The goal is to satisfy all specification criterions designated in the last section, while also adhering to the general goal and philosophy of the project as close as project. First of all, the design process will be documented and analysed. After that, the tools used to facilitate the design will be presented. Finally, each module design will be presented and scrutinized.

3.5.1 How to design a 16-bit computer module

In order to successfully come up with a design for a computer module while also meeting the specification outline and keeping the schematics as simple as possible, a simple yet effective design process has been implemented. It consists broadly of 4 distinct steps:

- Chip Discovery
- Chip Analysis
- Chip Selection
- Schematic Creation

3.5.2 Chip Discovery and Analysis

The first step towards the design of a 16-bit computer module is the *integrated circuit/IC/chip* discovery and analysis phase.

Why use Integrated Circuits/Chips? Since the general goal of the project is to produce a Turing complete computer which is as simple as possible to understand and also relies as little as possible on any other form of abstraction, one might argue that using integrated circuits, or chips, woud be against that goal. While this argument is theoretically correct, it doesn't take into account any practical considerations. The lowest level of the technology stack from which one could go ahead and phisically construct a computer would probably be the transistor. That being said, trying to implement any computer, let alone a 16-bit computer with many different modules, would be next to impossible to successfully do within the time frame of this project and also without a team of people. The physical scale would be many, many times what the scale of the computer built in this project is and the amount of errors which would arise from

connecting individual transistors together by hand would probably be so large, that one would quickly lose interest in finishing the build. As such, simple integrated circuits pose themselves as a great compromise. These circuits, which usually implement simple logic functions, like *AND, OR, Inverter, etc.*, or even a little bit more complex functions, like *registers or binary adders*, can each contain tens to hundreds of individual transistors. That being said, they should all be simple enough so thatthe functionality provided by each chip should be *understandable down to the transistor level*.

Understanding Integrated Circuits While this may sound daunting at first, it can actually be relatively straight forward. Individual logic gates are each made up of a few transistors. For example, Ben Eater has created an excellent YouTube tutorial on how to create logic gates out of transistors [2]. Armed with this knowledge, we can now approach the datasheet of a particular circuit we intend to use. For example, on page 2 of the datasheet of the *74LS157* [26] multiplexer chip, we can find the logic diagram. By analysing the logic diagram and calculating boolean outputs based on arbitrary inputs and then comparing the results to the truth table of the chip provided on the first page of the datasheet, one can verify that the chip actually implements its specified interface in a logical manner. This can be applied to all chips which are considered to be potential building blocks for a module, as long as the individual chip complexity is relatively limited.

3.5.3 Where to look for Integrated Circuits

There are three main sources of chips which where consulted when conducting chip discovery. The first one was the Wikipedia list of chips from *7400* family [44]. This particualr logic family was chosen because it is the most popular and widley spread and adopted logic family in the world [43]. The second souce of chips, which served as a quick reference, was Eater's part listing for his 8-bit breadboard computer build [6]. Since the 16-bit computer which is to be built is largley based around the 8-bit computer built be Eater, many of the chips used in the 8-bit version will also work in modules of the 16-bit version. Finally, for the few chips which were needed but weren't part of the 7400 family, the websites of the retailers from which the chips were to be bought were searched. Two retailers were used, *Mouser UK* [11] and *Farnell UK* [12].

3.5.4 Chip Selection

Usually, the particular functionality of the module which is to be designed drastically boils down the list of potential chips. For example, when designing a simple 16-bit register, like the *A register* 3.3.4, the *74LS273* 8-bit flip-flop [22] poses a great option. By just putting two *74ls273* chips next to each other, most of the functionality of the register has been designed. Whenever there is a need for extra functionality, for example in the case of the *74ls273* where there is no enable line, additional logic can be implemented using chips containing simple logic gates, like *AND, OR, NOT*. In this case, the clock signal can be AND'ed together with the *AI* line to ensure that the register only reads from the data bus when the control signal is set high.

3.5.5 Schematic Creation

With the right chips chosen for a module, the next step is to create a module schematic, which serves as a blueprint or design document. The open-source computer-aided design software chosen to facilitate the creation of module schematics is *KiCad* [31]. An overview of the application, as well as of the process of designing 16-bit computer modules with it will be presented in detail in the next section.

3.5.6 Electronic Design Tools with KiCad/Eeschema

The main tool used to to design schematics for the 16-bit breadboard computer is *KiCad* [31]. KiCad is a broad and mature electronics design software widley used in industry. It contains many tools which together from a tightly integrated toolchain for designing complex electronics from the concept phase all the way to the printed circuit board. Since the goal of the project is to build a computer on breadboards by hand and not create printed circuit boards, only the first step of the KiCad pipeline has been used, namley *Eeschema* [30].

Eeschema *Eeschema* is a electronic schematic computer design software. It is fully featured, easily accessible and has a relatively shallow learning curve. By using *Eeschema,* the entire module design process has been streamlined significantly.

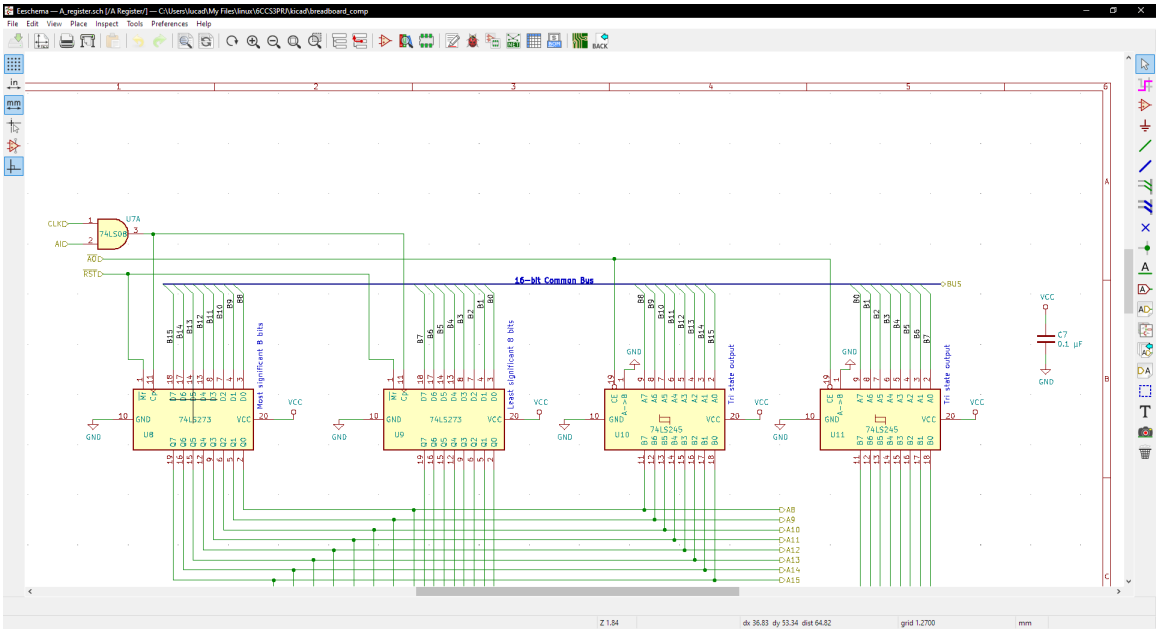


Figure 3.2: Eeschema Schematic Design software - part of the KiCad suite

In the following paragraphs the most important features of Eeschema will be presented.

Intuitive Interface The graphical user interface provided by Eeschema is similar to many other widley used photo or video editing software packages. As such, navigation is easy and intuitive. All tools and commands can be accessed both by pointer and keyboard, so one can become proficient in whatever interface method is preferred. Drag and drop functionality allows the user to easily reposition misplaced or misalligned circuits and components.

Hierarchical Sheets Hierarchical sheets is an Eeschema feature which allows one to embed entire schematics inside other schematics and seamlessly transition between child and parent schematics. This is a great way to organise the modular design of the 16-bit breadboard computer. Besides this, Eeschema offers the option to import input and output pins from the child schematic into the parent, so that they can be connected to the wider circuit. This makes designing a high-level overview diagram significantly easier.

Bus connections Wiring up 16 separate wires which are connected to most modules as both input and output can be extremley tedious. Fortunatley, *Eeschema* offers the option to bundle up" mutliple wires as a single Bus and then have connections going in and out of that single wire. Besides this, *Eeschema* can resolve which pins are connected to which over the bus by use of labels on the inputs and outputs 3.3.

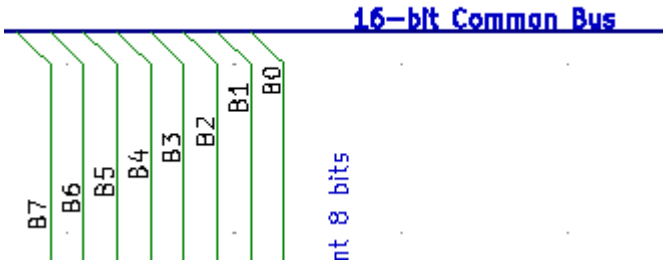


Figure 3.3: Labeled Bus Connections in Eeschema

Integrated Circuit Database *Eeschema* ships with a pre-loaded database of commonly used chips and circuits, called the *Symbol Library*. This makes rapid schematic prototyping possible by just looking up potential circuits for the module and then dragging and dropping them into the worksheet and then seeing if the connections would match up.

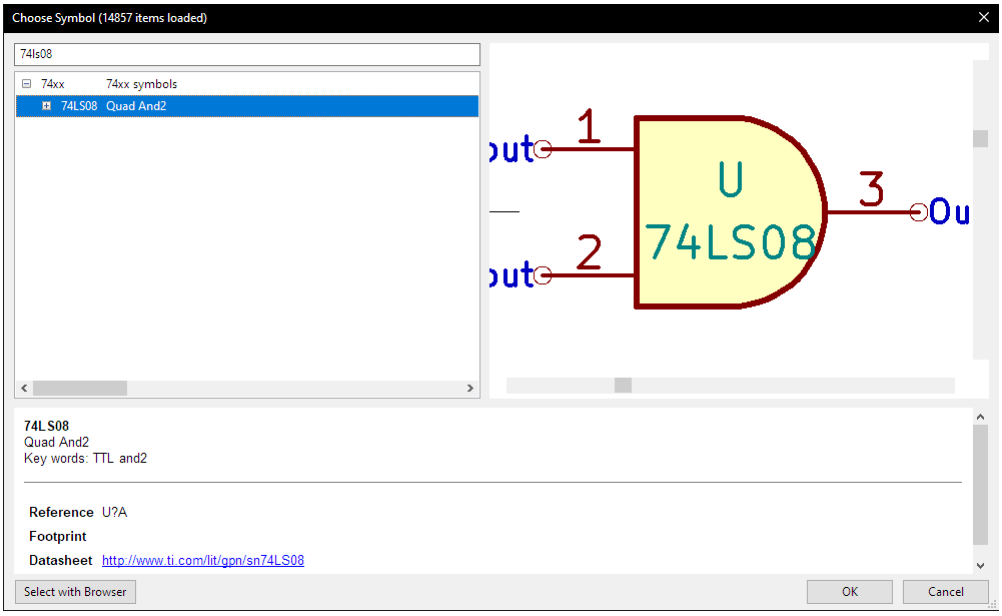


Figure 3.4: Component Database in Eeschema

Symbol Editor In the rare case where the *Symbol Library* didn't contain a circuit needed for a module, *Eeschema* provides a simple circuit symbol editor. This ensures that the final schematics produced were accurate and reliable, regardless of wherer the chip symbols were origianlly available in *Eeschema* or not.

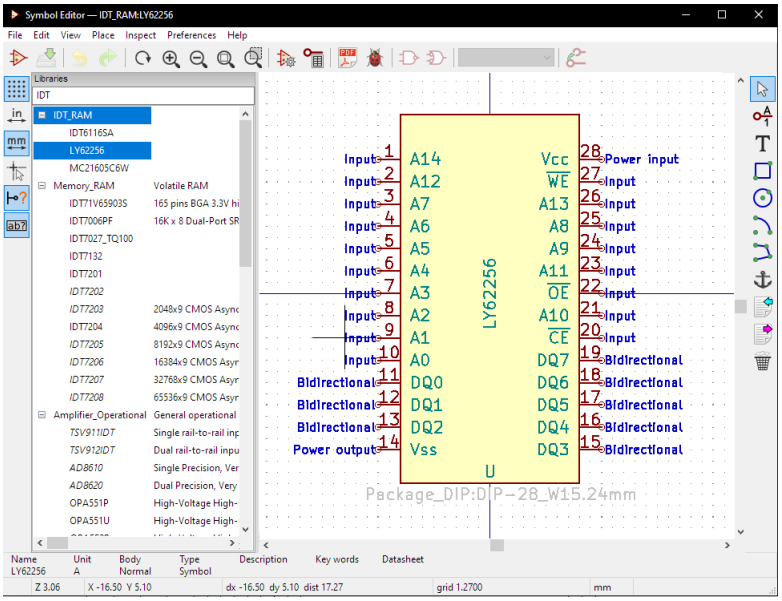


Figure 3.5: Symbol editor in Eeschema

Electrical Rules Check Besides providing an easy to use interface for designing accurate schematics, *Eeschema* also offers active help in the design process. By using the electrical

rules checker, *Eeschema* can check all connections against a user-defined set of rules and report errors. This can be used to detect potetential human errors or even chip incompatibilities and resolve them early on.

Figure 3.6: The Eeschema Electrical Rules Checker

Automatic Annotation When designing electrical circuits, it is common to use many identical or similar circuits next to one another. As such, when actually building the circuit, one can be confused about which chip correlates to which symbol in the schematics. To solve this, *Eeschema* assignes a unique identifier, or annotation, to each symbol. This annotation process can be done automatically.

Bill of Materials *Eeschema* can also generate a bill of materials, or *BOM*, based on all the symbols present in the schematic and hierarchical schemaitcs. The BOM is essential towards ensuring that all needed components are purchased.

PDF export Finally, Eeschema has been outfitted with an Export to PDF" option. This makes printing and compiling of the schematics into other documents, like this report, significantly easier.

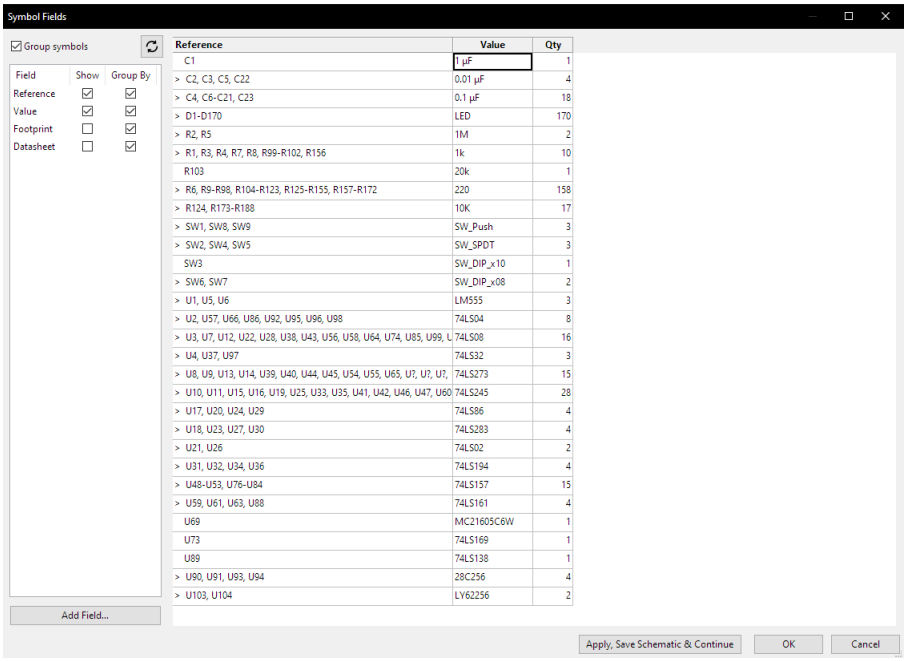


Figure 3.7: Simple Bill of Materials generated by Eeschema

3.5.7 Eeschema Design Process

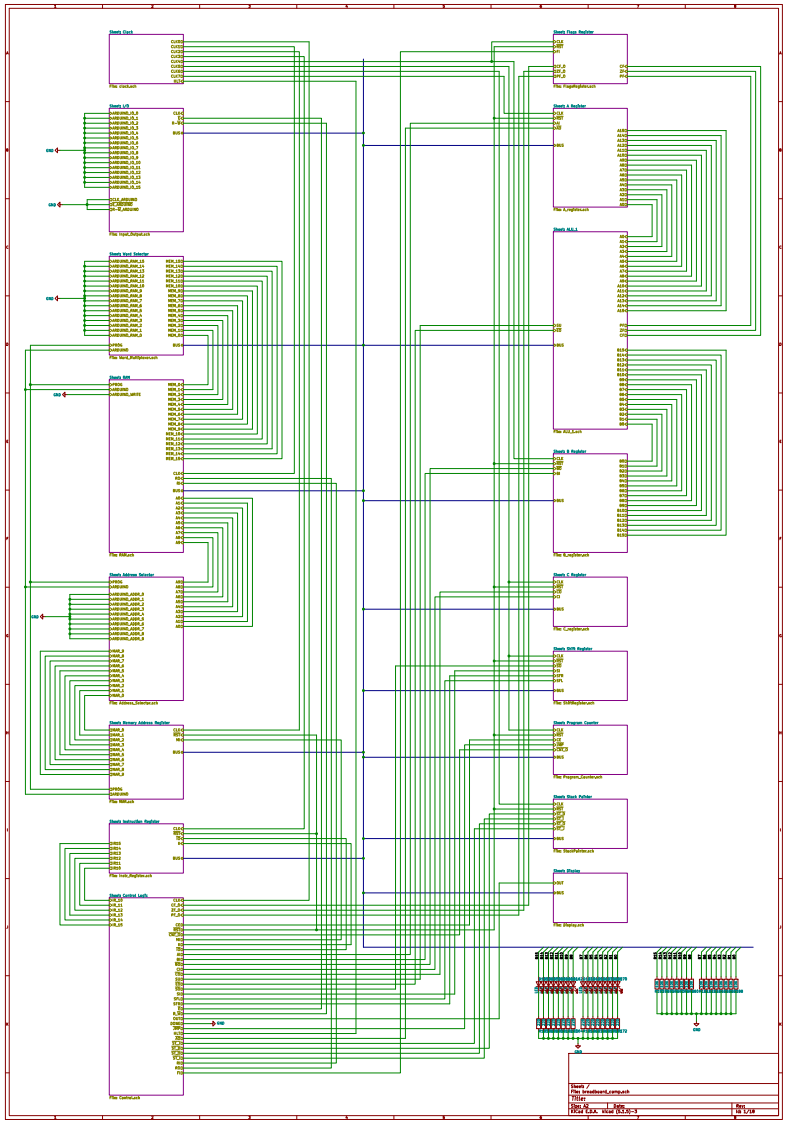
The following design process has been followed when creating schematics for a module as a part of the 16-bit breadboard computer:

1. Make an adequate chip selection
2. Create a new hierarchial sheet and enter it
3. Use the Symbol Editor to create chip symbols (in case the chips don't exist in the Symbol Library)
4. Place all needed chips in the schematic
5. Place all external control and I/O pins
6. Connect all the pins apropiateley
7. Leave the hierarchical sheet and import all I/O and control pins
8. Run the automated annotation tool
9. Run the electrical rules checker
10. Correct any errors

3.5.8 Finalised 16-bit breadboard computer design and schematics

The tools and processes discussed in the previous sections where extensivley used to produce the final schematics for the 16-bit breadboard computer, which were used as templates for the physical implementation of the computer. This section presents and discusses the schematics and the design decisions which were made during their creation.

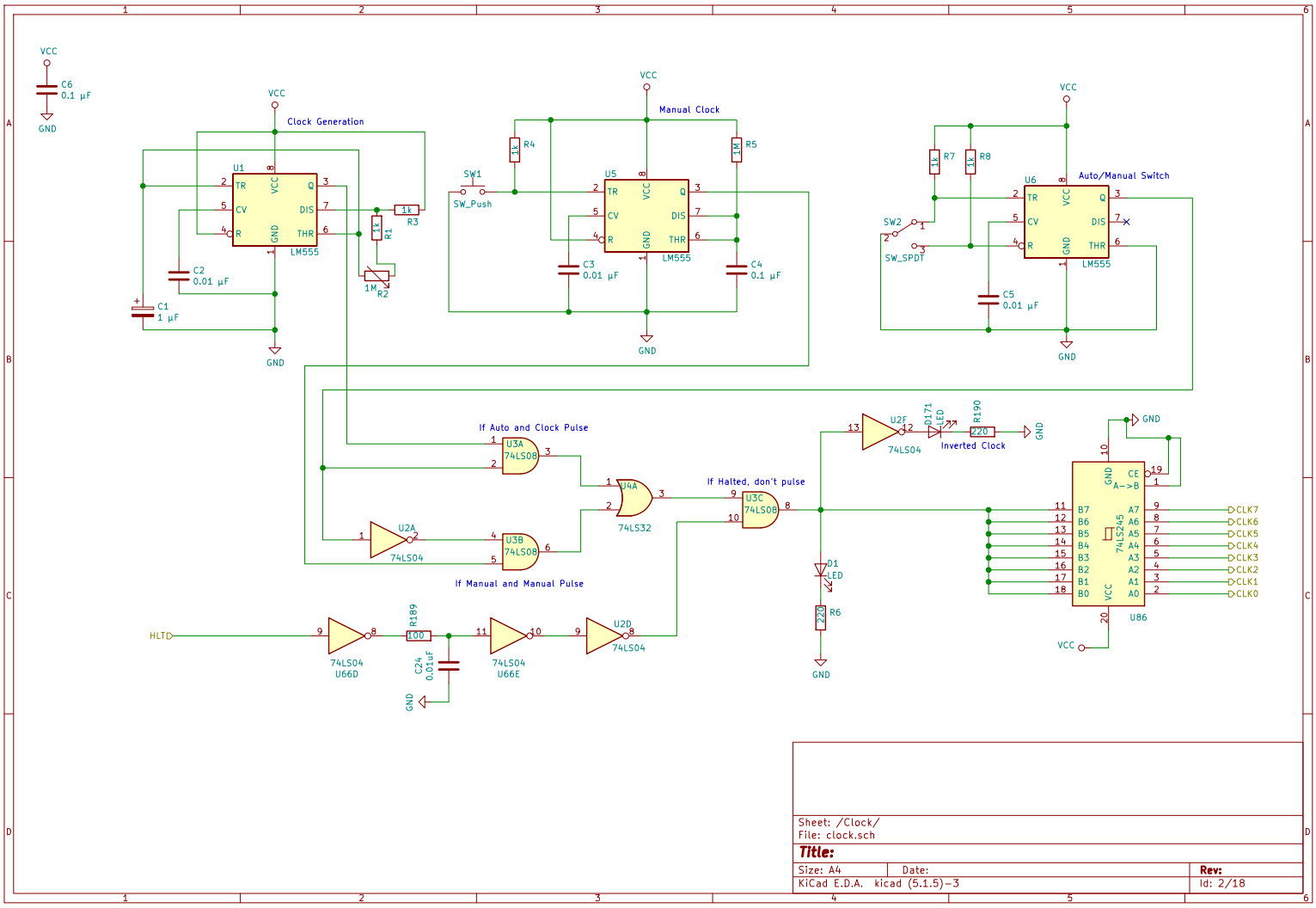
High-Level Schematic The high-level schematic provides a layout and scale overview for the finished computer. It also contains the *Data Bus LEDs* module 3.3.2, as well as the actual *Common Data Bus* 3.3.1, to which all components connect. Towards the bottom of the schematic the *LEDs* module 3.3.20 can also be found.


Sheets Clock
File: clock.sch
Sheets I/O
File: Input_Output.sch
Sheets Next Selector
File: Next_Multiplexer.sch
Sheets RAM
File: RAM.sch
Sheets Address Selector
File: Address_Selector.sch
Sheets Memory Address Register
File: MAR.sch
Sheets Instruction Register
File: Inst_Register.sch
Sheets Control Lach
File: Control.sch
Sheets Flags Register
File: FlagsRegister.sch
Sheets A Register
File: A_register.sch
Sheets ALU
File: ALU.sch
Sheets B Register
File: B_register.sch
Sheets C Register
File: C_register.sch
Sheets Shift Register
File: ShiftRegister.sch
Sheets Program Counter
File: Program_Counter.sch
Sheets Stack Pointer
File: StackPointer.sch
Sheets Status
File: Status.sch
GND
Sheets /
File: breadboard_comp.sch
Title:
Size: A2
Date:
Rev:
KiCad E.D.A. kicad (5.1.10)-1
Id: 1/1

Clock 3.3.3 The clock module is very similar to the clock designed and built by Ben Eater [10]. It uses *555 timers* [20] to generate a square wave which acts as the clock for the system. The *555* is presented in all three most commonly used configurations:

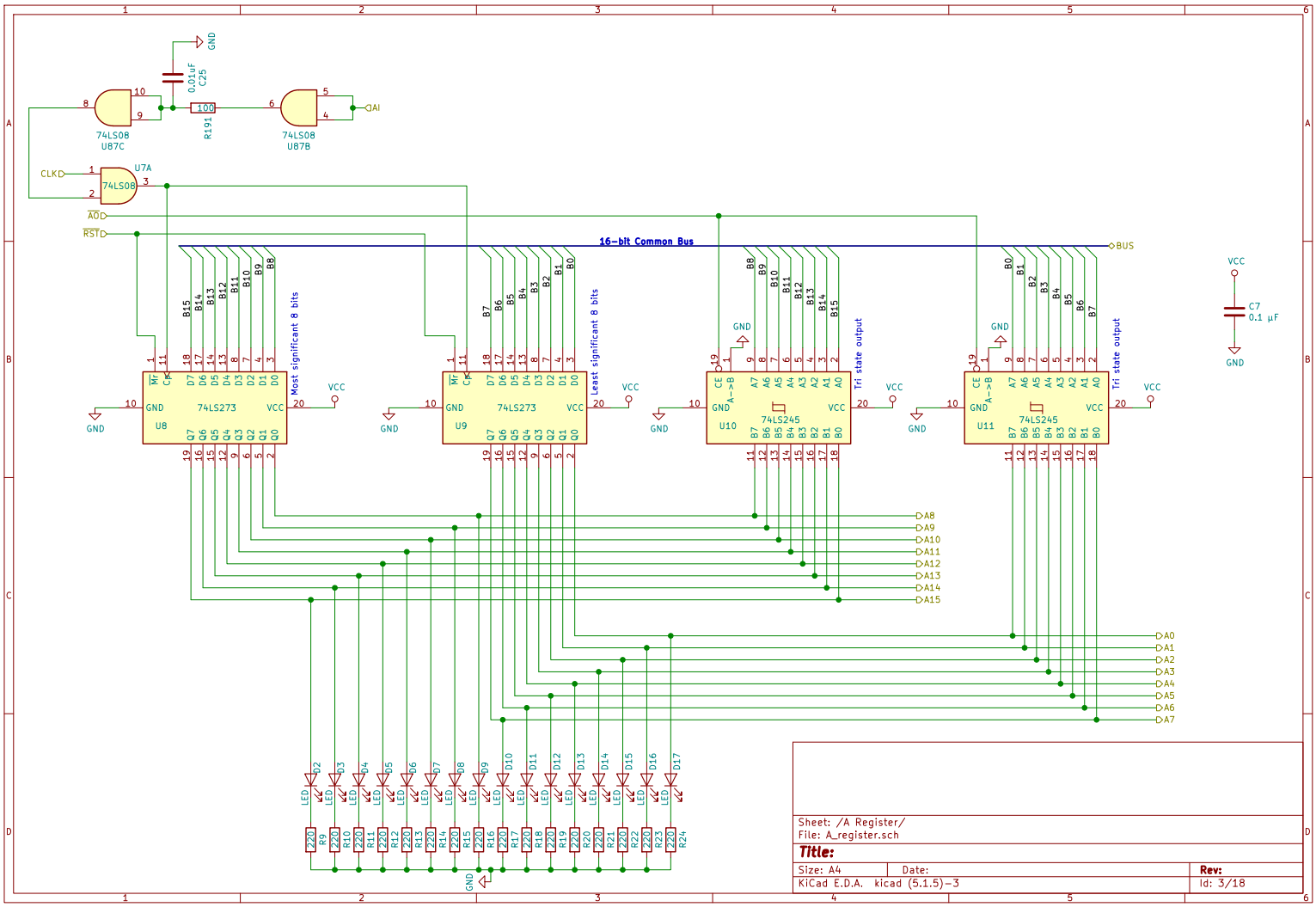
- *Astable:* it oscillates between to states (This 555 generates the main clock)
- *Monostable:* it always „settles down“ to one state after a short period of time (used for the manual push-button clock mode)
- *Bistable:* it has two states between which it can toggle (used for the toggle switch which goes from automatic to manual mode)

The clock-pulse generating *555* is connected to a variable resistor which can be adjusted to increase or decrease the frequency accordingly. A simple combinatorial circuit is used to select the appropiate clock signal and inhibit it if the *HLT* (Halt) signal is active. The main addition to the Eater design is a *74ls245 buffer* [21] used to amplify and isolate clock signals going to each module.

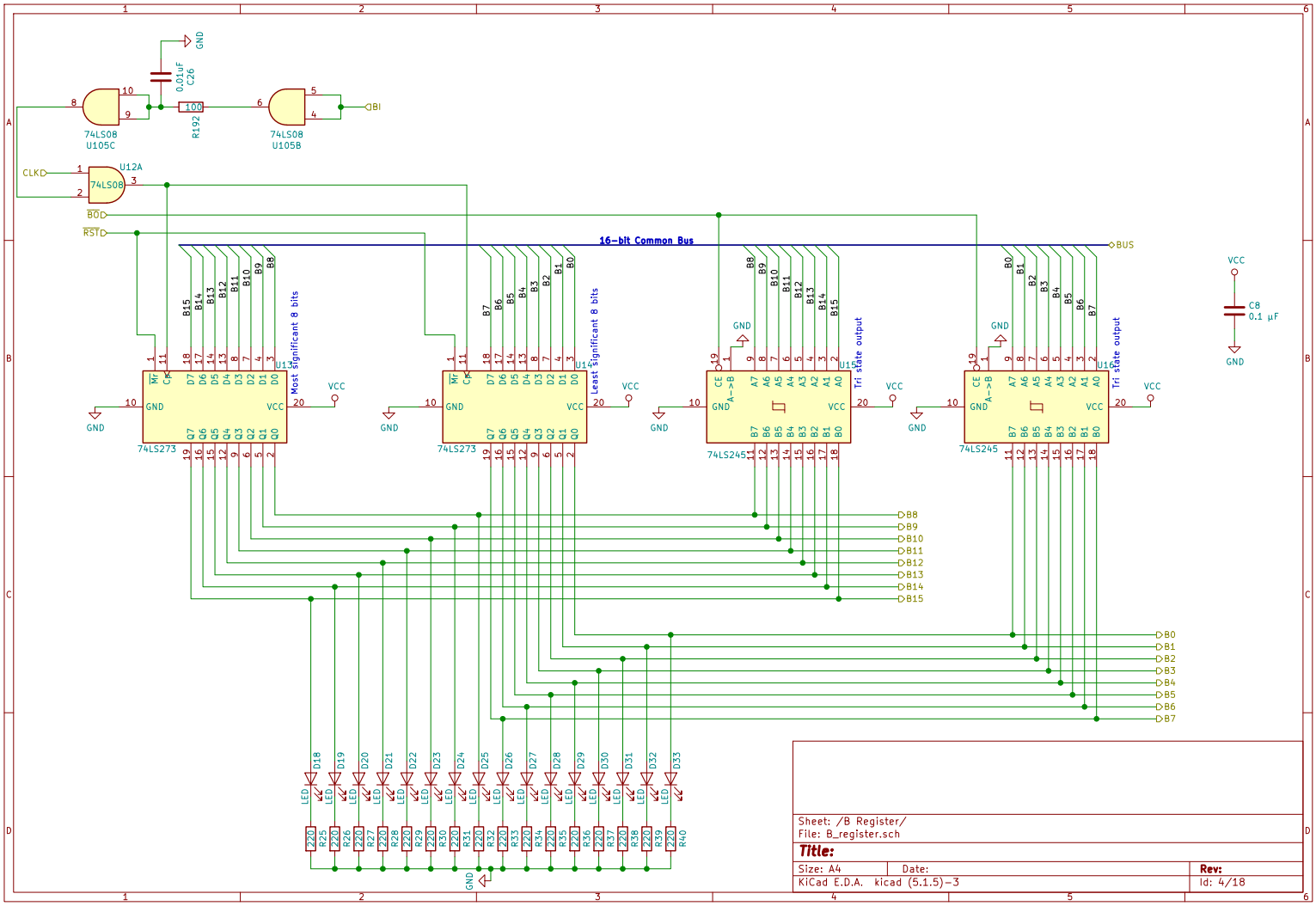


Sheet: /Clock/		
File: clock.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (5.1.5)-3		Id: 2/18

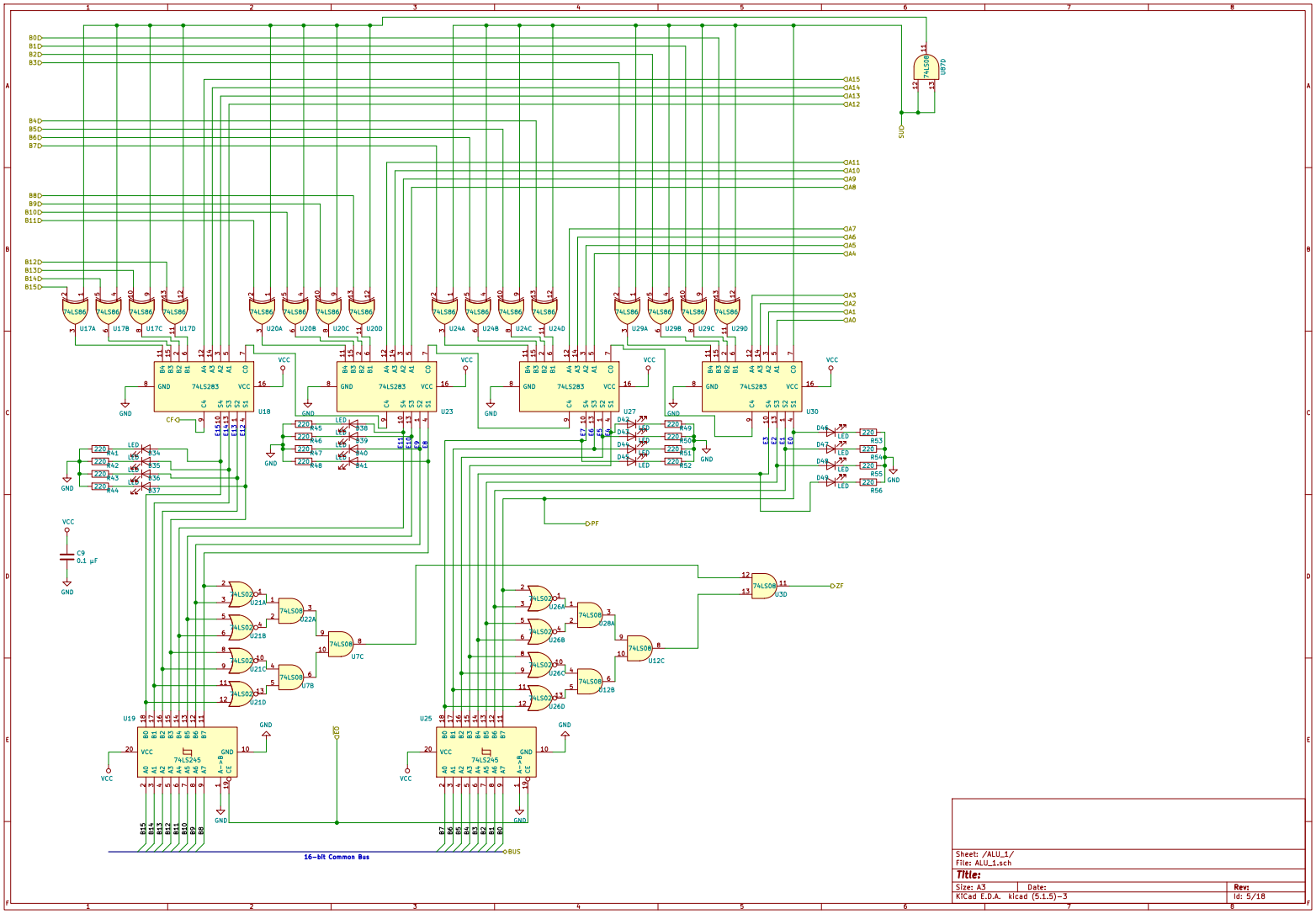
A Register 3.3.4 The A register consists of two *74ls273* [22] 8-bit flip-flop chips and two *74ls245* [21] buffer interfaces to the bus. The buffers are *tri-state* buffers, which means that they can pass through their input, either high or low, or they can disconnect their input from their output by putting themselves in a high-impedence state. This functionaility, used across most modules which interface with the bus, is used to allow the register to assert its contents onto the bus when reading from it (this happens when the $\overline{AO}$ signal is taken low). Writing to the register is handled by ANDing together the AI signal with the clock pulse. The contents of the A, which can be foud on the Q pins, are connected directly to the ALU 3.3.6.


74LS08
U87C
74LS08
U87B
0.01uF
C25
100
R191
GND
AI
CLK
U7A
74LS08
AO
RST
16-bit Common Bus
BUS
Most significant 8 bits
Least significant 8 bits
Tri state output
74LS273
U8
U9
74LS245
U10
U11
VCC
C7
0.1 µF
A8
A9
A10
A11
A12
A13
A14
A15
A0
A1
A2
A3
A4
A5
A6
A7
LED
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
R19
R20
R21
R22
R23
R24
Sheet: /A Register/
File: A_register.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.5)-3
Id: 3/18

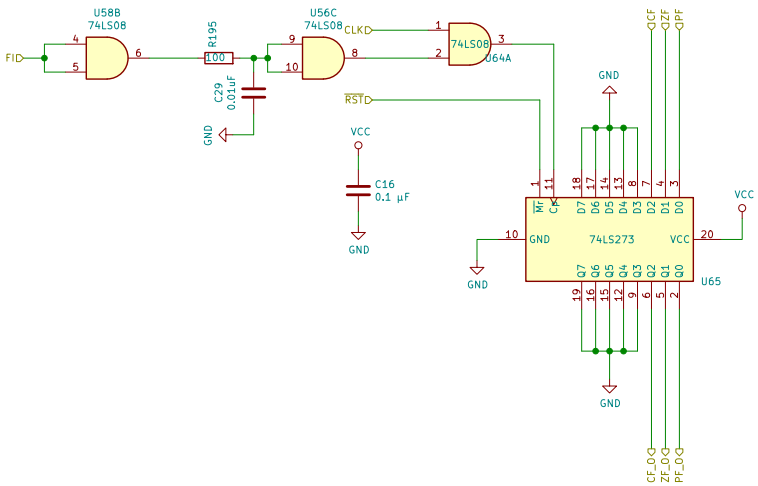
B Register 3.3.5 The B register is built essentially the same as the A register.


74LS08
U105C
0.01uF
C26
100
R192
74LS08
U105B
BI
CLK
U12A
74LS08
BO
RST
16-bit Common Bus
BUS
C8
0.1 µF
U13
Most significant 8 bits
74LS273
U14
Least significant 8 bits
74LS273
U15
Tri state output
74LS245
U16
Tri state output
74LS245
LED
220
GND
VCC
Sheet: /B Register/
File: B_register.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.5)-3
Id: 4/18

Arithmetic-Logic Unit 3.3.6 The *ALU* has been designed around the *74LS283* [19] chip. It contains a 4-bit binary full adder. They can be chained together by connecting the Carry-Out of one chip to the Carry-In of the next chip. This way, by using 4 chips, a full 16-bit adder can be constructed. The A inputs are connected to the A register 3.3.4 and the B inputs are conected to the B register 3.3.5. The sum output is connect to two *74LS245* [21] buffers which make the connection to the data bus. To achieve subtraction functionality, the inputs from the B register are converted to two's complement binary notation, since subtracting a value equates to adding up the same negative value. To convert a positive binary value to a two's complement negative value of the same magnitude, all 16 bits have to be flipped and then 1 has to be added to the number. Flipping the bits is accomplished using the *74LS86* [23] , which provides four independet XOR (exclusive or) gates. By XORing each bit of the bit register with the value of the *SU* control signal, each bit will just pass through the gate if *SU* is low and it will be flipped if *SU* is high. Besides this, *SU* is also connected directly to the Carry- In Input of the first *74LS283* full adder chip, to artificially add one to the final sum, representing the one which needs to be added in for the two's complement form of the B register contents. The *ALU* also has three *flag signals* which go out to the *Flags Register* 3.3.7. These are the *Parity Flag (PF)*, which represents the state of the lowest-significand bit of the sum, the *Zero Flag (ZF)*, which is high if the sum is 0 and low otherwise, and the *Carry FLag (CF)*, which is set if the sum inside the *ALU* cannot be expressed within 16 bits. These flags can be used to make branching decisions based on the result of a calculation. The *Carry Flag* and *Paritiy Flag* are straight-forward to obtain. The *Carry Flag* is the Carry-Out pin of the most-significand adder circuit. The *Parity Flag* is the least significand sum pin of the least significand adder circuit. To calculate the *Zero Flag*, *NOR* (inverted or) gates are employed on each pair of two sum bits. These gate can be found on the *74LS02* [25] chips. If any of the sum bits goes high, then the output of at least one *NOR* gate will go low. Subsequently, all *NOR* outputs are ANDed together using successive *AND* gates found on *74LS08* [24] chips. By using the and operator, if any one of the *NOR* gates goes low, then the final output, which is the *Zero Flag*, will go low. The only case where the *Zero Flag* will go high is if all sum bits are 0.

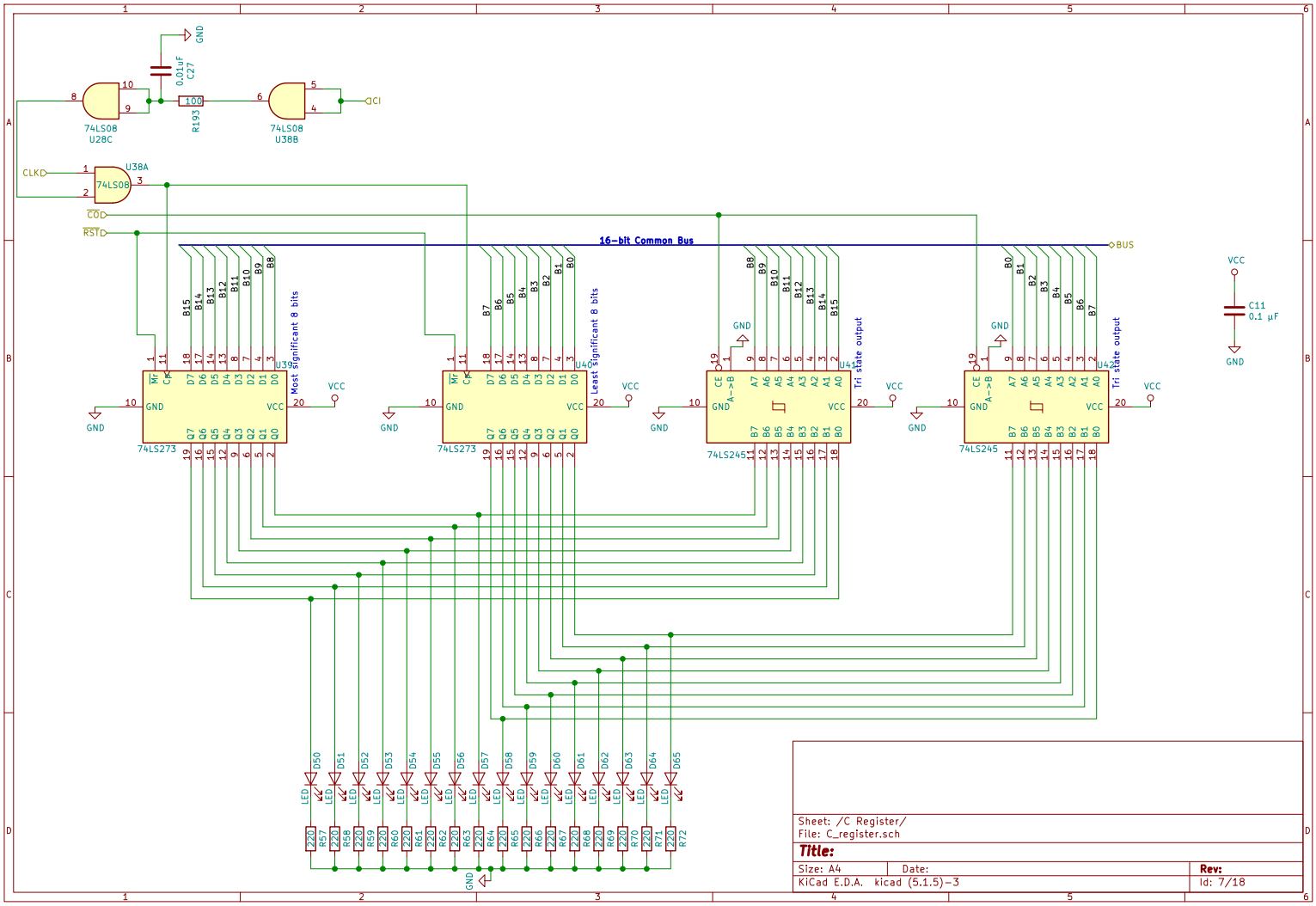

B0D
B1D
B2D
B3D
B4D
B5D
B6D
B7D
B8D
B9D
B10D
B11D
B12D
B13D
B14D
B15D
BUS
16-bit Common Bus
ZF
PF
CF
VCC
GND
74LS86
74LS283
74LS08
74LS02
74LS245
74LS04
Sheet: /ALU_1/
File: ALU_1.sch
Title:
Size: A3
Date:
Rev:
KiCad E.D.A. kicad (5.1.5)-3
Id: 5/18

Flags Register 3.3.7 The Flags register consists of just one *74LS273* [22] and some simple logic to handle the selective reads. Allthough it is inneficient to use an 8-bit register to store only three bits, this design decision has been made because the *74ls273* is used in many other modeuls of the system and to avoid having to understand how another chip works and how to use it.



Sheet: /Flags Register/		
File: FlagsRegister.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (5.1.5)-3		Id: 11/18

C Register 3.3.8 The C register is a general purpose 16-bit register built just like the A register 3.3.4 or the B register 3.3.5.

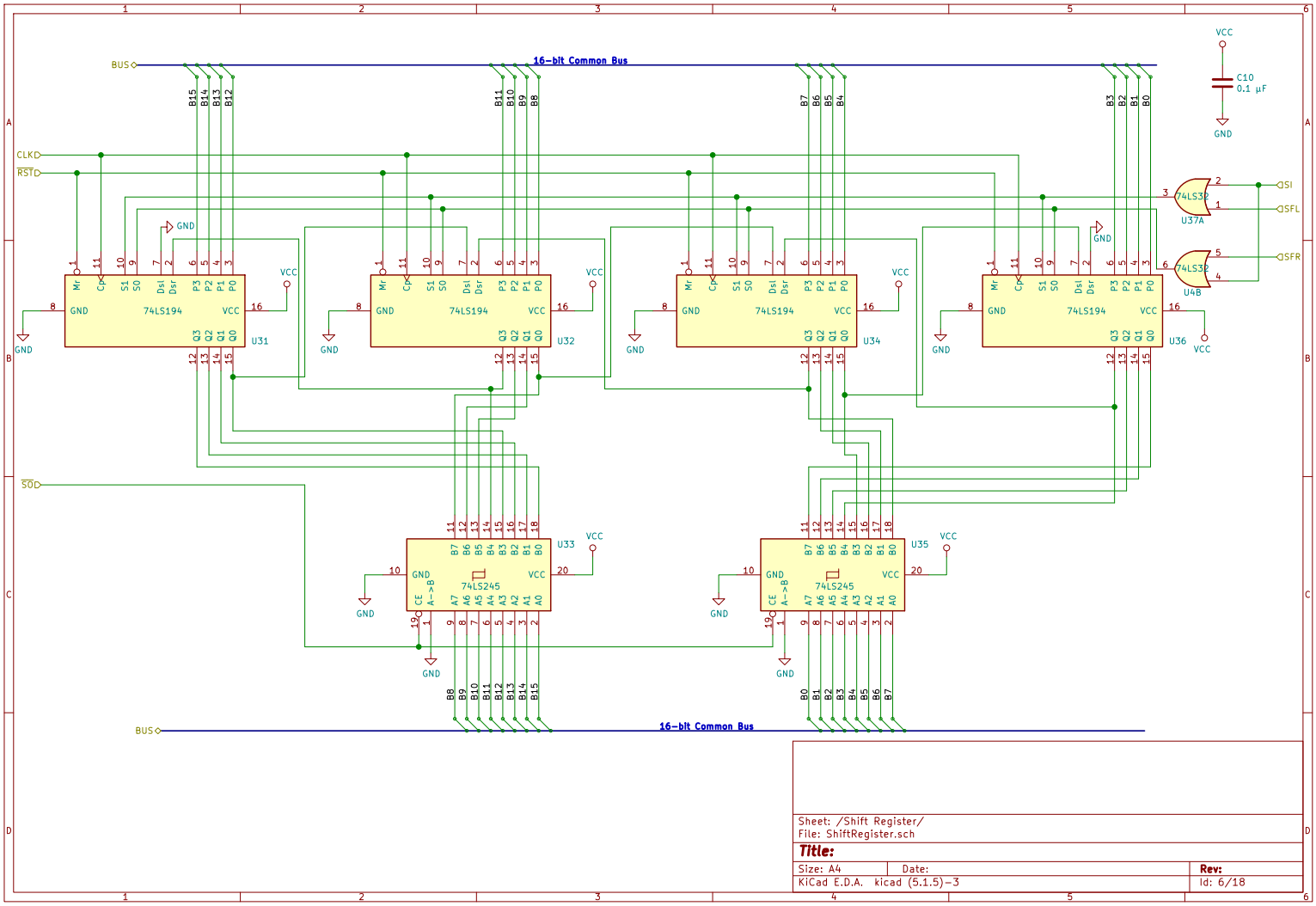


Sheet: /C Register/
File: C_register.sch

Title:

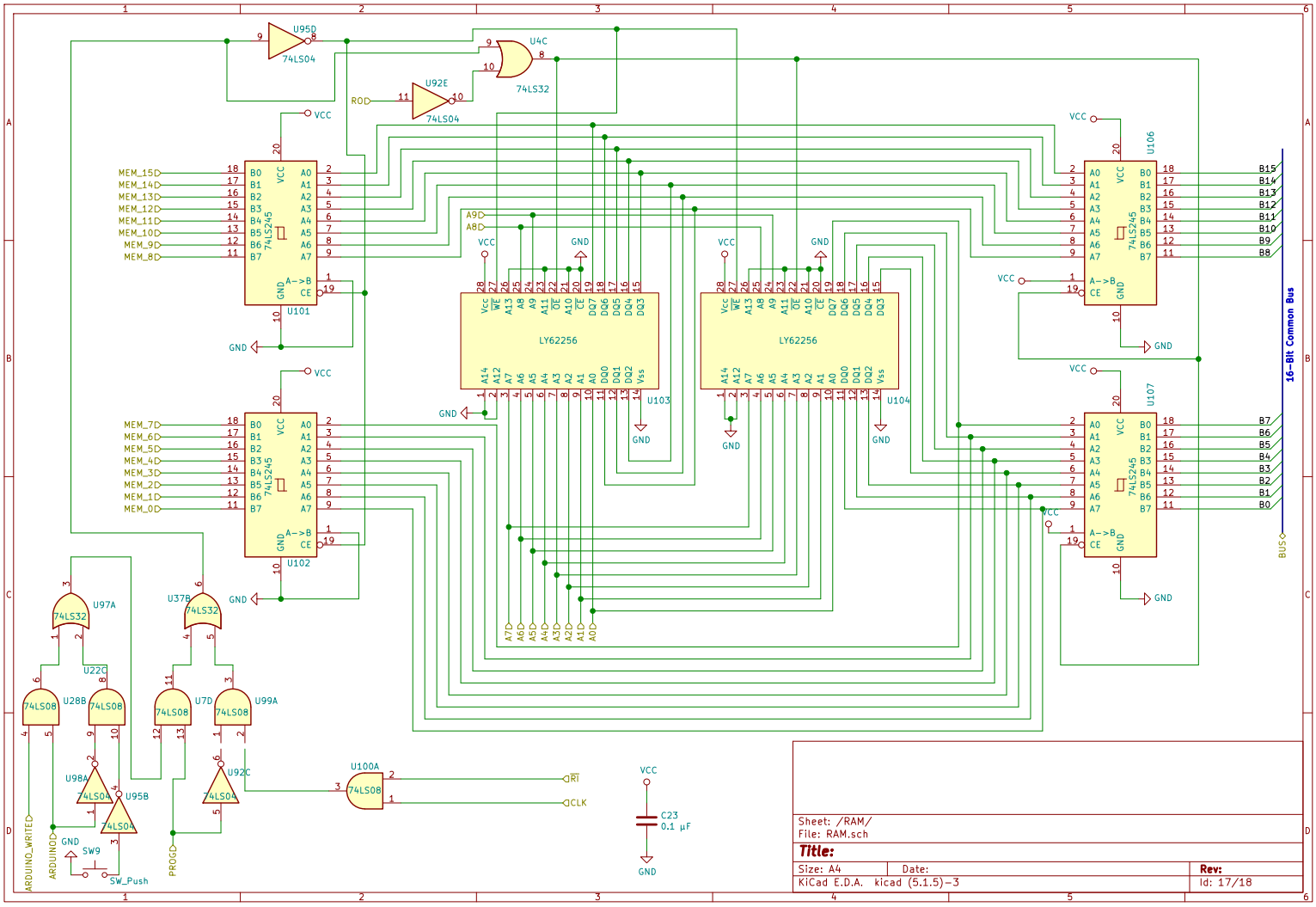
Size: A4 | Date: | **Rev:**
KiCad E.D.A. kicad (5.1.5)-3 | Id: 7/18

Shift Register 3.3.9 For the design of the shift register, the *74LS194* [18] was chosen. It is a 4-bit register which implements bit shifting in both directions. Multiple *74LS194* chips can be chained together to create a larger shift register by connecting the Serial-Left and Serial-Right inputs of one chip to the data outputs of the chips to its left and right. The chips have two control inputs, *S0* and *S1*. If both inputs are low, the register will do nothing on the next clock pulse. If S0 or S1 is high, but not both, then a shift-right or a shift-left will occur, depending on which input is high. If both are high, then a parallel load will occur. The data outputs of each chips are connected two two *74LS245* [21] chips to provided the interface to the data bus. The two control inputs are fed in from a simple combinatorial circuit which converts the three control signals, *SI*, or shift register in, *SFL*, or shift left and *SFR*, or shift right, into the two lines which connect directly to the register chips.

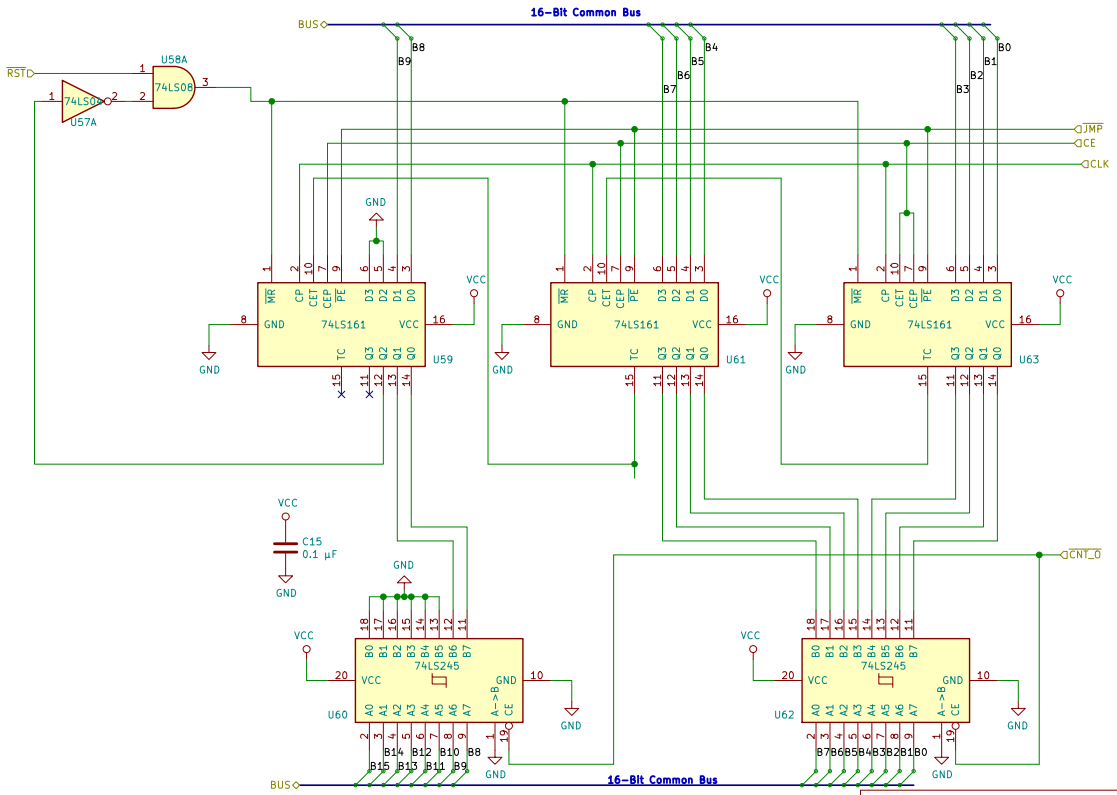

16-bit Common Bus
BUS
CLK
RST
SO
SI
SFL
SFR
VCC
GND
C10
0.1 µF
74LS194
U31
U32
U34
U36
74LS32
U37A
U4B
74LS245
U33
U35
Sheet: /Shift Register/
File: ShiftRegister.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.5)-3
Id: 6/18

Random Access Memory (RAM) 3.3.10 The *Random Acces Memory* module, or RAM module, poses an interesting design challenge. Thee *7400* family of integrated circuits does not include any circuits which would fulfil the 16K memory space specification previously chosen in a simple and intuitive manner. As such, different chips had to be chosen. In the end the *LY62256* 32K x 8-bit static RAM chip [16] from *Lynotek Inc.* and stocked by [12] was selected. This chip was chosen because it provides a sufficiently large address space as well as simple control inputs. Since the chips are not part of the standard symbol library, a symbol had to be created for the *LY62256* chips in the Symbol Editor. The chips have 15 address lines, too many for the 16-bit breadboard computer. This is not an issue though, as the extra lines, *A10* through *A14*, can be tied to logic 0 and ignored.

One chip stores one byte of data at any given address, so two chips were needed to adequately cover the entire 16K memory array. Since the chips use the same pins for both data reads and writes, two sets of two *74LS245* buffers [21]. In this configuration, the data pins of each RAM chip are locked between two buffer chips and is, as such, effectivley isolated from the rest of the computer. If reads a read operation is requested, the data bus facing buffers will activate and allow the data pins of the RAM chips to drive the bus. If a write operation is requested, the word selector facing buffers will activate and allow data to flow from the word selector to the RAM chip data pins. Besides this, some simple combinatorial logic is used to implement a selector for the write *RI* signal based on the *ARDUINO* and *PROG* signals.

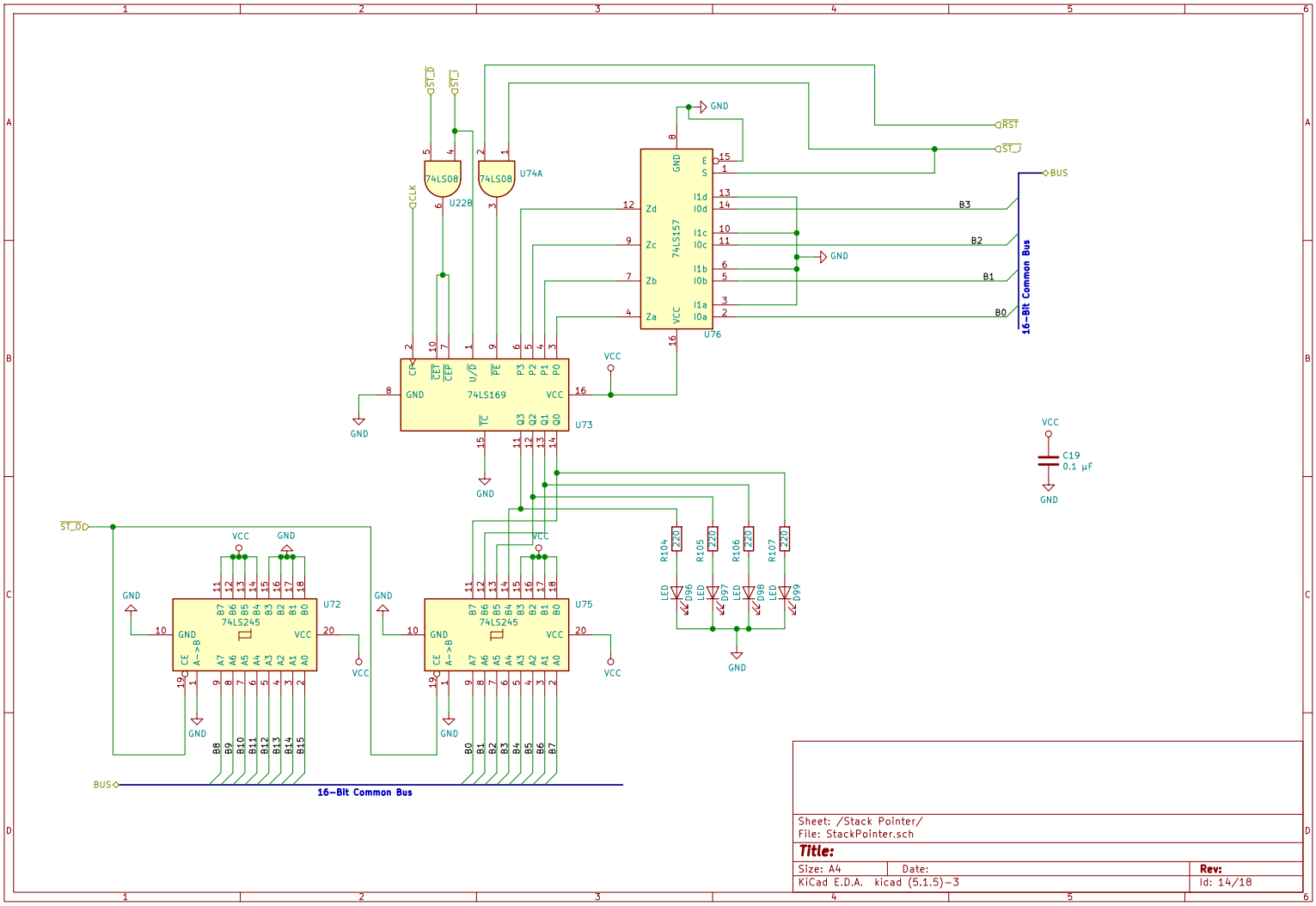

U95D
74LS04
U4C
74LS32
U92E
74LS04
ROD
VCC
GND
U101
U102
U103
U104
U106
U107
74LS245
LY62256
A9
A8
MEM_15
MEM_14
MEM_13
MEM_12
MEM_11
MEM_10
MEM_9
MEM_8
MEM_7
MEM_6
MEM_5
MEM_4
MEM_3
MEM_2
MEM_1
MEM_0
B15
B14
B13
B12
B11
B10
B9
B8
B7
B6
B5
B4
B3
B2
B1
B0
16-Bit Common Bus
BUS
U97A
U37B
U22C
U28B
U7D
U99A
U98A
U95B
U92C
U100A
IRT
CLK
ARDUINO_WRITED
ARDUINOD
PROGD
SW9
SW_Push
C23
0.1 µF
Sheet: /RAM/
File: RAM.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.5)-3
Id: 17/18

Program Counter 3.3.11 The centerpiece of the program counter module design is the *74LS161* 4-bit binary counter [28]. This chips implement most of the functionaility needed for the program counter. If the two enable signals are high, the chip will count up in binary on the next clock pulse. If the *Ripple Carry Output* pin is fed forward to the *Enable T* signal of the next counter, multiple counter can be used to create a binary counter of arbitrary length. In the case of the 16-bit breadboard computer, to cover a 1K address space 10 bits are needed. This translates to three counters. Since the most significand two bits of the most significand counter are not needed, the Q2 pin can be integrated into the reset circuit, so that the counter loops back to 0 after reaching *0x3FF*. Besides this, by having a parallel load pin, the *74LS161* facilitates jump functionailty, used for programming branch and loop instructions. As usual, the bus interface is handled by two *74LS245* [21] chips. Since only 10 bits are connected from the counter chips to the buffers, the remaining six bits are tied low. The control signals are decoded using simple combinatorial logic.

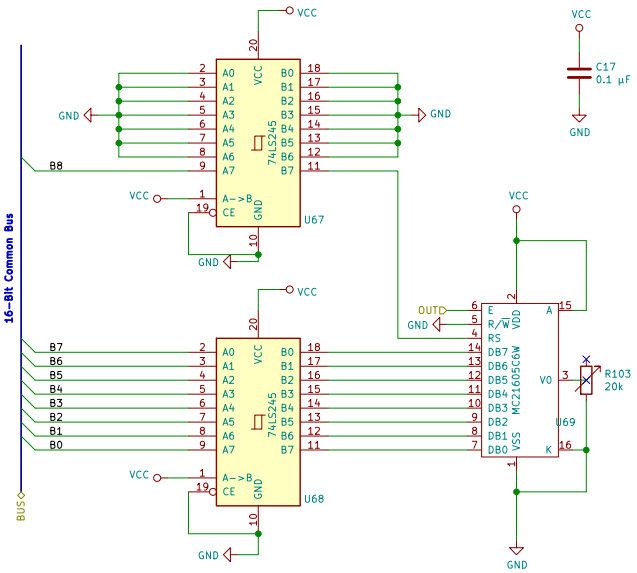


Sheet: /Program Counter/	
File: Program_Counter.sch	
Title:	
Size: A4	Date:
KiCad E.D.A. kicad (5.1.5)-3	Id: 10/18

Stack Pointer 3.3.12 The stack is designed using a *74LS169* 4-bit binary up-down counter chip [27]. Allthough limited, the chip fits the specification because only a small fraction of the address space is alloted to the stack. The stack is formed of the last 16 addresses, so only 4 bits are needed to specify them, since the remaining 6 bits are identical for all stack addresses. The *74LS169* differs from the *74LS161* [28] used in the Program Counter 3.3.11 in the fact that it trades in the master reset pin with the up-down pin, which controls which way the counter counts. As such, to preserve reset functionality, a *74LS157* [26] multiplexer chip is used. This chip selects between two sets of 4 bits based on the state of a single control bit. This is used in the stack pointer for loading in either the lowest significant four data bus bits, in case a stack jump is to be executed, and 4 low values, in case a reset is to be executed. This restores the missing reset functionality. Besides this, two *74LS245* [21] chips are used to interface to the bus. Besides the four bits provided by the stack pointer, the next six bits are tied to logic 1, to reflect the position of the stack address range in the memory space. The rest are tied to logic 0. Finally, some simple gate logic manages the control signals.

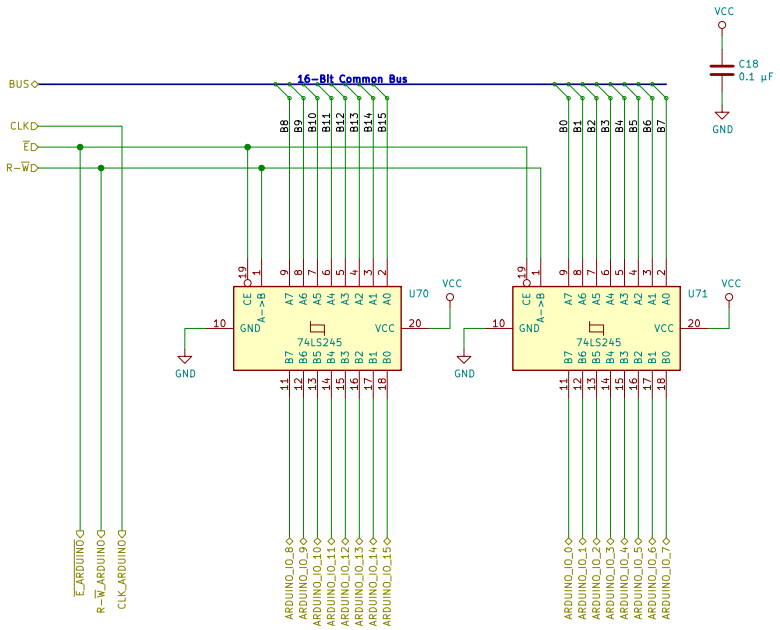

ST_D
ST_I
74LS08
U74A
U22B
CLK
GND
RST
ST_I
BUS
74LS157
U76
B3
B2
B1
B0
16-Bit Common Bus
74LS169
U73
VCC
C19
0.1 µF
ST_O
74LS245
U72
U75
R104
R105
R106
R107
220
LED
D96
D97
D98
D99
16-Bit Common Bus
Sheet: /Stack Pointer/
File: StackPointer.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.5)-3
Id: 14/18

Display 3.3.13 To fit the specification of a 16 characters by two line display, while maintaining relative simplicity, the *c1602a v1.2* [32] was chosen. This simple and widley popular LCD modules are easy enough to use and abstract a sizable amount of the issues with character generation and display, so that a single person can master them in a relativley short span of time. They have an internal character set and also allow the user to specify as small set of custom characters. To interact with the *c1602a v1.2*, a word of 9 bits is sent to the display on the rising edge of the *Out* signal. The first bit specifies wheter a command or a character is sent to the display, and the remaining byte represents the command or the character. The display also implements functionailty which allows the main processor to read data from it, but this will be abstracted away, as it is not part of the specification. Besides the data and *Out* signal, the *c1602a* requires just power connection for the processor and backlight, as well as a variable resistor for the adjustable contrast. Besides the display, two *74LS245* buffer chips [21] are used to interface the display to the data bus. That being said, only the least significand 9 data bits pass trough the buffer to the display, as that's all that is required.



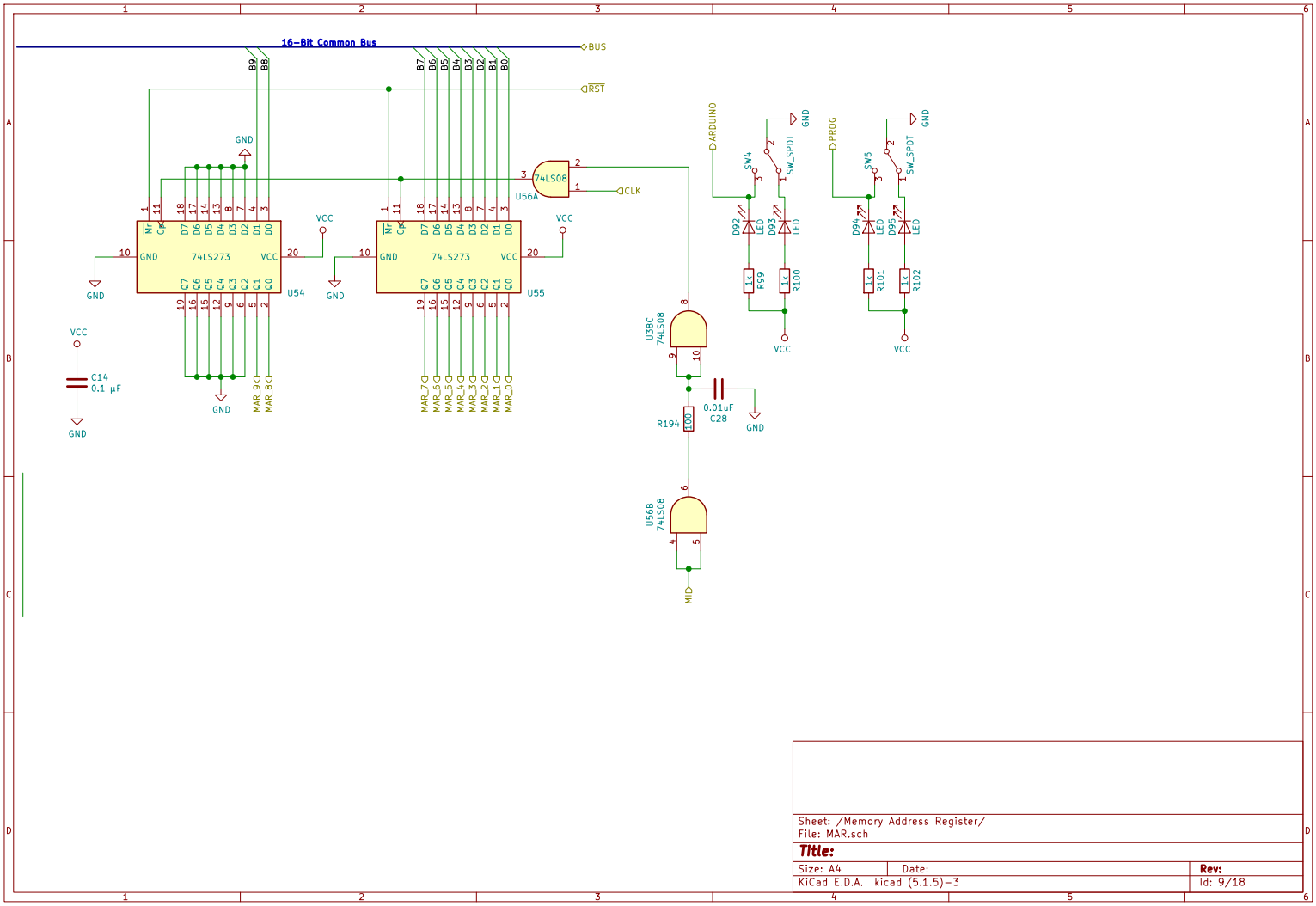
Sheet: /Display/		
File: Display.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (5.1.5)-3		Id: 12/18

I/O 3.3.14 The I/O module provides outside connective to an *Arduino Mega* [13]. To achieve this, two *74LS245* [21] buffer chips are used. When the buffers are disabled, the Arduino and the main system are essentially disconnected. This ensures that there is no interference when no I/O requests are issued. Furthermore, the buffers are actually bi-directional, so by means of the *Read/Write* control signal, the computer can control if data flows from the data bus to the arduino in case of an I/O write, or the other way around in case of an I/O read. The arduino can use the *Enable* signal to trigger an interrupt and handle the I/O request and the same *Read/Write* to decide what the type of the I/O request is and handle it accordingly.



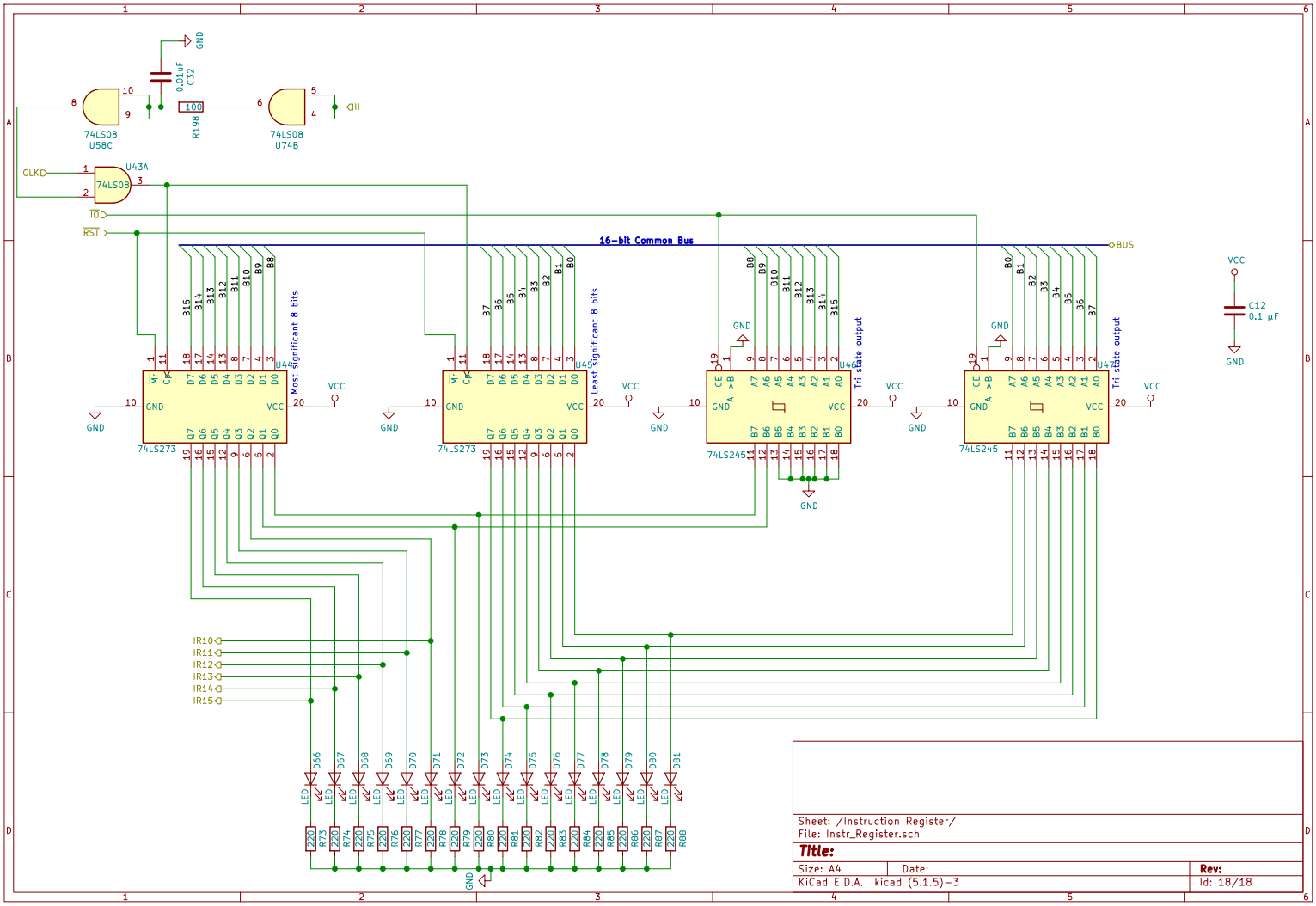
Sheet: /I/O/		
File: Input_Output.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (5.1.5)-3		Id: 13/18

Memory Address Register 3.3.15 To design the memory address register, the same chips which was used in the A register 3.3.4, the B register 3.3.5, the C register 3.3.8 and the flags register 3.3.7, namely the *74LS273* **??**, comes to mind. Since the memory address register will not assert its contents to the bus, there is no need for a data bus buffer. But, since the module is built up of few components and is placed in the vicinity of RAM 3.3.10 and the address selector 3.3.18, it lends itself as a good location for the two toggle switches which will be used to set the *ARDUINO* And *PROG* control signals.

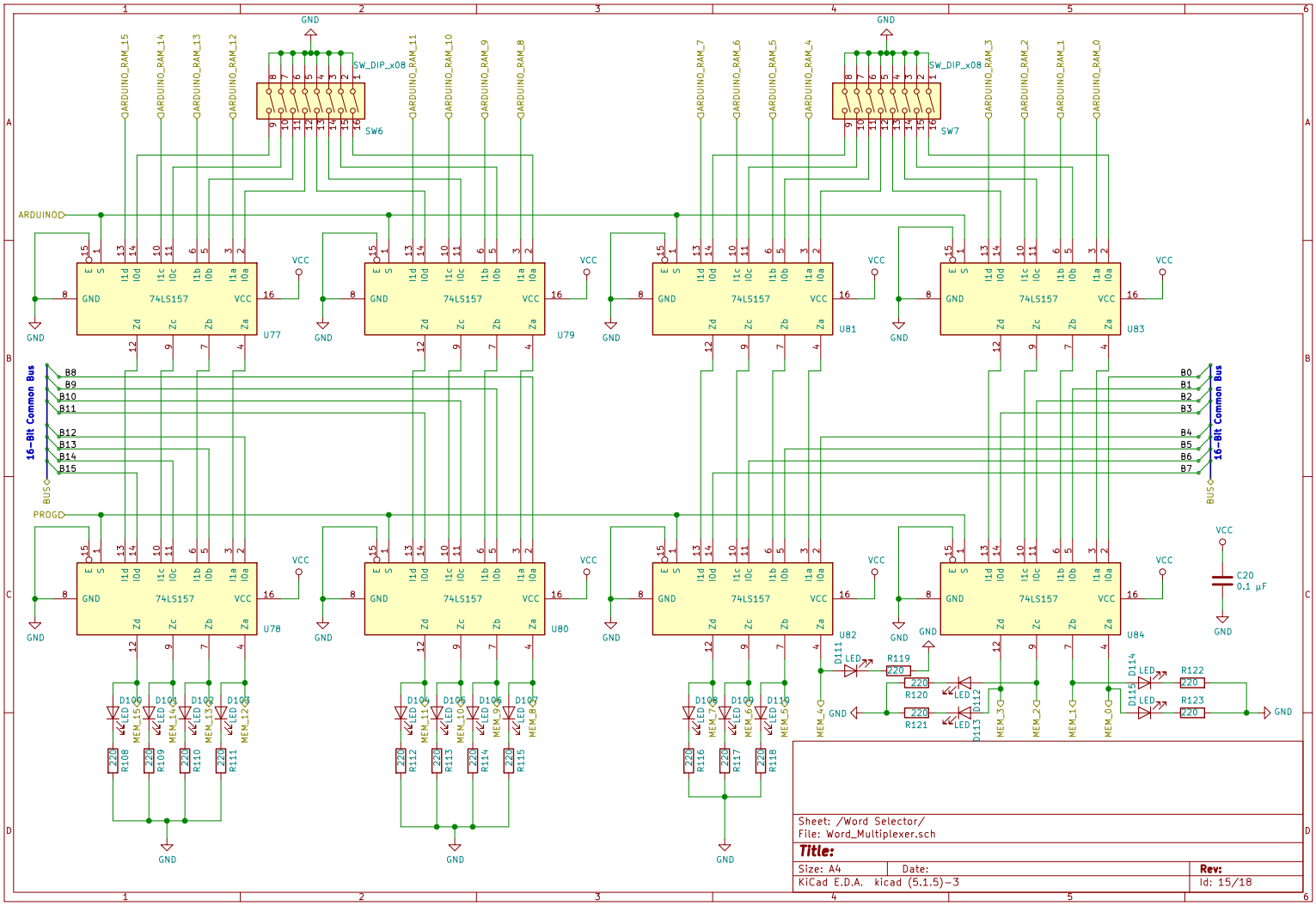


Sheet: /Memory Address Register/		
File: MAR.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (5.1.5)-3		Id: 9/18

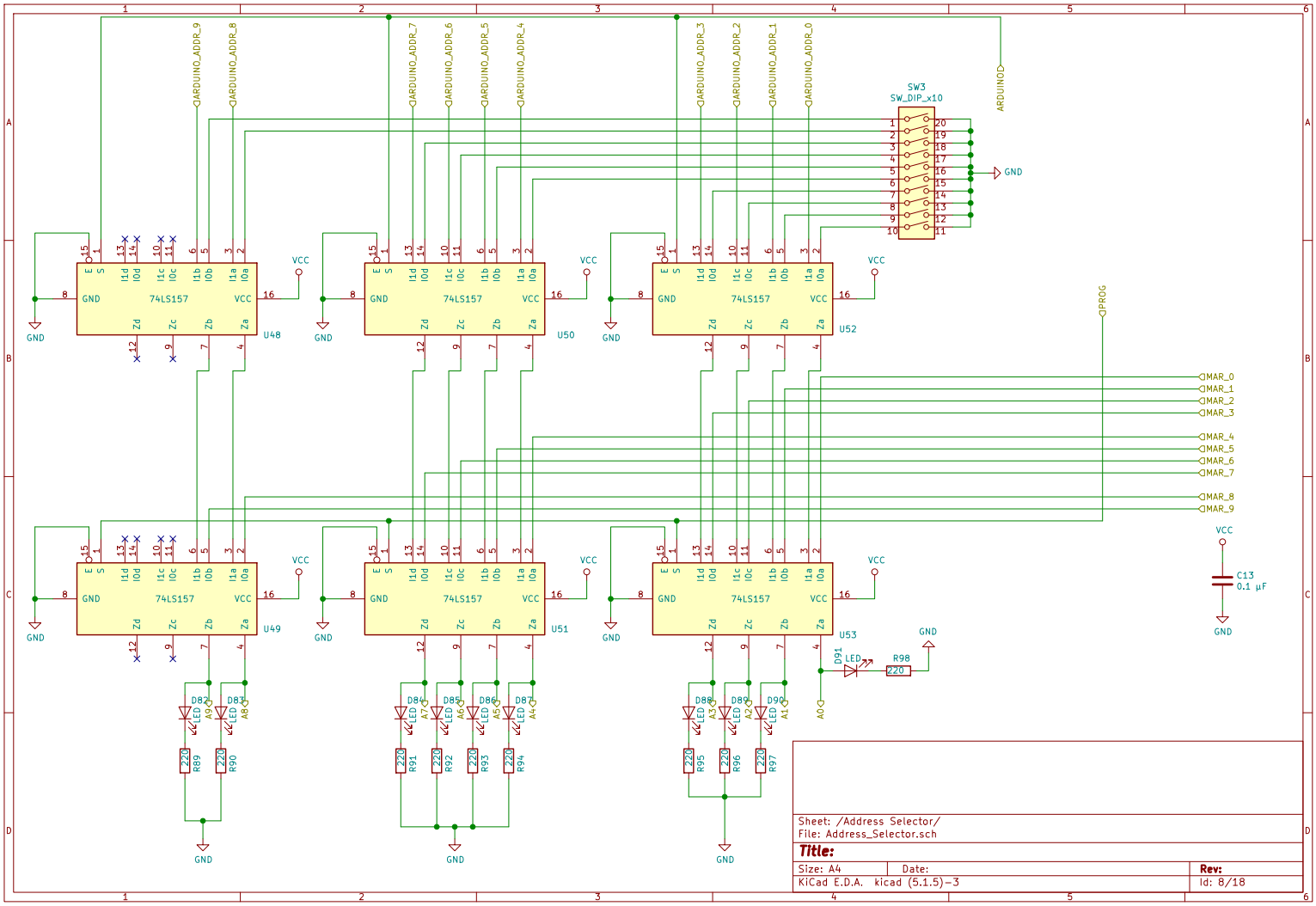
Instruction Register 3.3.16 The instruction Register is very similar in construction the the A register 3.3.4, B register 3.3.5 and C register 3.3.8. It uses two *74LS273* [22] 8-bit flip-flop chips to store one 16-bit word of data, which can be latched in from the data bus. It also has two *74LS245* buffer chips used to assert its contents onto the bus when a read request is issued. The main difference between the Instruction Register and any other register is the fact that only its 10 least significand bits are asserted back onto the data bus during a read operation. This is because the Instruction Regsiter holds instructions, which are structered as a six-bit *opcode* and a 10-bit *address or immediate operand*. The 6-bit *opcode* is connected directly to the *control logic* module. Control Signal processing is handled using a simple AND gate.


GND
0.01uF
C32
74LS08
U58C
100
R198
74LS08
U74B
CLK
U43A
74LS08
IO
RST
16-bit Common Bus
BUS
VCC
C12
0.1 µF
GND
Most significant 8 bits
U44
74LS273
Least significant 8 bits
U45
74LS273
Tri state output
U46
74LS245
Tri state output
U47
74LS245
IR10
IR11
IR12
IR13
IR14
IR15
LED
220
Sheet: /Instruction Register/
File: Instr_Register.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.5)-3
Id: 18/18

Word Selector 3.3.17 The Word Selector design heavily relies on the *74LS157* data selector chip [26]. A two stage design is proposed. The first stage of four selector chips chooses between the data provided by the /emphDIP switches and the *Arduino Mega* [13]. This stage is handeled by the *ARDUINO* control signal. The output of this selector stage is fed forward to the second stage of 4 selector chips. These chips decide wheter to output the programming data provided by the previous selectors or the bus data based on the *PROG* signal.


SW_DIP_x08
SW6
SW7
ARDUINO_RAM_15
ARDUINO_RAM_14
ARDUINO_RAM_13
ARDUINO_RAM_12
ARDUINO_RAM_11
ARDUINO_RAM_10
ARDUINO_RAM_9
ARDUINO_RAM_8
ARDUINO_RAM_7
ARDUINO_RAM_6
ARDUINO_RAM_5
ARDUINO_RAM_4
ARDUINO_RAM_3
ARDUINO_RAM_2
ARDUINO_RAM_1
ARDUINO_RAM_0
ARDUINOD
PROGD
74LS157
U77
U78
U79
U80
U81
U82
U83
U84
16-Bit Common Bus
BUS
B8
B9
B10
B11
B12
B13
B14
B15
B0
B1
B2
B3
B4
B5
B6
B7
GND
VCC
C20
0.1 µF
D100
D101
D102
D103
D104
D105
D106
D107
D108
D109
D110
D111
D112
D113
D114
D115
LED
MEM_16
MEM_15
MEM_14
MEM_13
MEM_12
MEM_11
MEM_10
MEM_9
MEM_8
MEM_7
MEM_6
MEM_5
MEM_4
MEM_3
MEM_2
MEM_1
MEM_0
R108
R109
R110
R111
R112
R113
R114
R115
R116
R117
R118
R119
R120
R121
R122
R123
220
Sheet: /Word Selector/
File: Word_Multiplexer.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad (5.1.5)-3
Id: 15/18

Address Selector 3.3.18 The Address selector is designed the same way as the Word Selector described in the previous paragraph. The main difference is that, since it only has a final output of 10 bits, each selector stage only contains three *74LS157* [26] selector chips.



Sheet: /Address Selector/		
File: Address_Selector.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (5.1.5)-3		Id: 8/18

Control Signals The design of the Control Logic module 3.3.19 is centered around the *AT28C64B* EEPROM chip from *Atmel* [1]. EEPROM stands for *Electronically Erasable and programmable Read-Only Memory.* Essentially, an EEPROM memory acts as a look-up table. It's input is an address and the output is a word of data. This can be used as a substitute for any combinatorial circuit made out of ANDs, ORs, inverters etc.. By choosing an EEPROM, instead of building a combinatorial circuit, the completed design benefits from multiple advantages:

- *Simplified design:* if a combinatorial circuit would be designed in the place of EEPROMS, it would be of a significant size, given the large number of variables (10 binary inputs and 31 binary outputs). Such a circuit would not only be very dificult to design and physically implement with chips and wires, it would also go against the core design principle of simplicity of understanding.

- *Easy re-progamming:* EEPROMs can be re-programmed using an of-the-shelf USB programmer, a microcontroller like an Arduino **??** or even a simple custom built circuit. This makes them especially suitable for application in the design of 16-bit breadboard computer as they can easily be removed from the module, reprogrammed to include new instructions or modify existing ones and then re-inserted into to rest of the circuit.

- *No extra conceptual abstraction*: While the EEPROM chips themselves are much more complex chips compared to most of the chips used in other modules (their data sheet doesn't provide a transistor diagram or even a low-level diagram, just a high-level block diagram), conceptually they don't increase the complexity of the design, since they can be thought of as a combinatorial circuit made of simple gates or a memory array made up of rows of registers with a selection circuit. As such, the simplicity of both concept and physical implementation is maintained.

The reasons for choosing this particular EEPROM chip are as follows:

- *Parallel Interface:* There are two main types of EEPROMs: parallel and serial. A serial interface uses a single pin to transmit addresses and data respecitvley. This poses issue to the design of the module since a separate, faster clock and a shift circuit would be needed to shift in addresses and shift out data on each clock cycle. This significantly adds to the complexity of the entire design. A parallel interface, on the other hand, is much easier to work with, since all address bits and all data bits are available at the same time, which means they can be accessed synchronously on the same clock cycle.

- *Address width:* The *AT28C64B* [1] is part of a larger family of EEPROM chips designed and built by atmel. The main differences between the chips are access time and address width. They all have the same data width of 1 byte. Since the plan is to run the clock of the breadboard computer at a relatively low frequency (under 1 kHz), the access time will not make a significant difference. What is important to consider, though, is the address width. The different chips in the Atmel EEPROM Family (AT28C16, AT28C64, AT28C256 etc.) have addresses of different widths. The *AT28C64B* [1] offers 13 address bits, enough to accommodate the 10-bit address width of the breadboard computer (this requirement will be explained shortly).

- *Wide-spread adoption and compatibiliy:* The *AT28C64B* is a widley used chip and thoroughly understood chip. As such, it makes interfacing with programmers trivial, as most programmers have a profile for it already built in, and it also comes with a plethora of online tools, documentation, personal experiences and tips and tricks from other users of the chip.

The design of the *Control Logic* module 3.3.19 makes use of 4 *AT28C64* chips. Each chip is reasponsible for 8 control signals, with the exception of one chip which handles only 7 signals (the computer features 31 contol signals without the two clock signals). The 13 address bits of the EEPROMs are split up into a logical address of 10 bits and a zero- address of three bits, since only 10 address bits are required by the computer. This 10 bit address is split up into three components:

- *6-bit opcode:* The lowest six bits of the address are the opcode of the instruction. These are connected directly from the *Instruction Register* 3.3.16

- *3-bit flags:* The next three bits are the flags which come directly from the *Flags register* 3.3.7

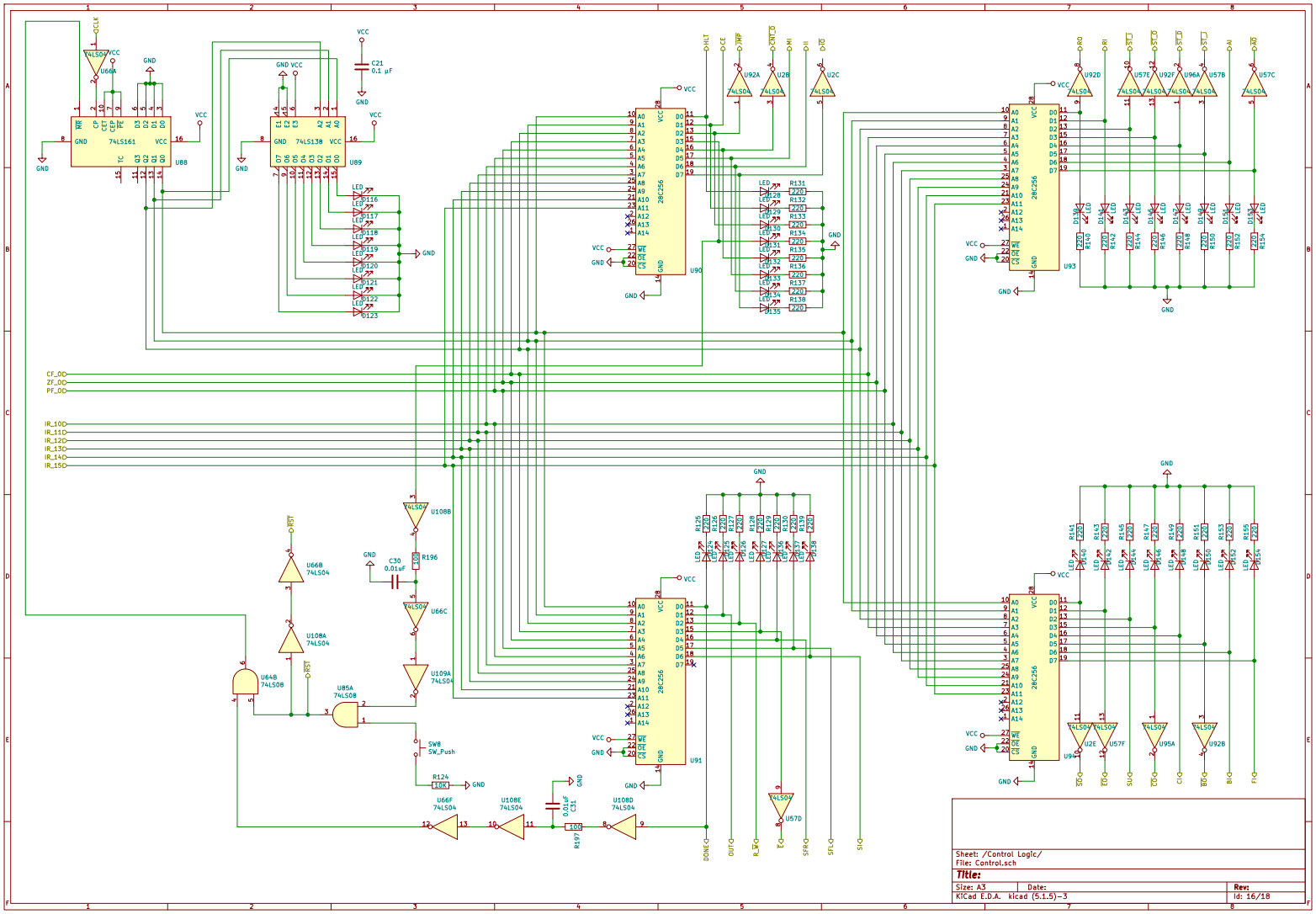
- *3-bit time step:* The last three bits form a step counter.

The reasoning for including a time-step counter into the address is because of the fact that each instruction has to execute multiple sequences of control signals to fullfil its task. The counter is built out of a *74LS161* [28] 4-bit counter chip, from which only the lowest order three bits are used. These bits are fed into the four eeprom chips, as well as into a *74LS138* [17] chip which turns a three bit code into 8 bits, where only one bit is active at one time. The decoded 8 bits are connected to LEDs to make visually recognising the current time-step easier. The

74LS161 [28] chip counts based on the negative, or inverted clock. As a consequence of this, the control logic module has time between the clock pulses of the main clock to set a new address and then set appropiate control signal for the next main clock pulse. To make programming of the EEPROM chips easier, all signals which are active-low are inverted, so that a logic 1 will be programmed if the signal should be active and a logic zero if the signal should be inactive. Besides this, the *control signals* module 3.3.20, which just consists of LEDs on all control signal lines, was built into this diagram. The final part to discuss on the design of the control logic module is the reset circuit. Three elements play into the reset circuitry:

- *A simple push button*, so that the computer can be manually reset
- *The RST control signal*, whih allows the computer to reset itself
- *The DONE control signal*, which allows the computer to terminate an instruction early, in case it has less then 8 steps.

The reset push-button as well as the *RST* signal are combined and together create the *RST control signal*, which is then spread out to all modules of the computer. Besides this, this signal is further combined with the *DONE* signal, as this signal, if taken high, should reset *only* the *74LS161* step counter.
This concludes the design of the control logic module.


CLK
U66A
74LS04
VCC
GND
U88
74LS161
U89
74LS138
C21
0.1 µF
D116
D117
D118
D119
D120
D121
D122
D123
LED
HALT
CE
IMP
CNT_D
MI
II
DI
U92A
U2B
U2C
U90
28C256
U93
U92D
U57E
U92C
U96A
U57B
U57C
RD
RI
STI
STO
STD
STI
AI
AO
R131
R132
R133
R134
R135
R136
R137
R138
D135
D139
D140
D141
D142
D143
D144
D145
D146
D147
D148
D149
D150
D151
D152
D153
D154
R140
R142
R144
R146
R148
R150
R152
R154
220
CF_OD
ZF_OD
PF_OD
IR_10
IR_11
IR_12
IR_13
IR_14
IR_15
RST
U108B
U66B
74LS08
U108A
74LS04
U64B
74LS08
U85A
74LS08
C30
0.01uF
R196
100
U66C
U109A
74LS04
SW8
SW_Push
R124
10K
U66F
74LS04
U108E
74LS04
U108D
74LS04
C31
0.01uF
R197
U91
R125
R126
R127
R128
R129
R130
R139
D124
D125
D126
D127
D128
D129
D138
U94
U2E
U57F
U95A
U92B
U57D
DONE
OUT
R_W
E
SYNC
SYNC
SIG
SIO
EO
CI
BO
FI
Sheet: /Control Logic/
File: Control.sch
Title:
Size: A3
Date:
Rev:
KiCad E.D.A. kicad (5.1.5)-3
Id: 16/18

Chapter 4

Hardware Implementation

4.1 Hardware choices

With the specification and design phase of the 16-bit breadboard computer now complete, the next step towards having a woking and functional computer was to make some judicious hardware choices. First up, the breadboards had to be chosen.

4.1.1 Breadboard selection

Following Ben Eater's advice [7], the first choice of breadboards was the *BB830* from *BusBoard Technologies*, retailed by *Mouser* [41]. Unfortunately, due to limited stock at the time, only 10 such breadboard could be aquired. Because of this, a compromise had to be met. The rest of the boards bought were cheaper BB830 clones retailed by *Farnell* [40]. The operating principle of a breadboard is that each vertical column of 5 pin is connected, together, as well as the horizontal top and bottom two rows which provide power and ground. The way breadboards provide resilient connections without using solder is by having small metal clamps inside the pin holes which, when pried apart by an inserted pin, will rub against it and, as suh, provide high quality contact points. The main issue which could arrise from using low quality breadboards is the fact that, after only a few pin insertions, the metal clamps loose their elastical properties and stay pried open, not making contact anymore. Since the plan is to insert all components once and then leave them in their position, this issue shouldn't manifest itself. Besides breadboard choice, how to connect multiple breadboards is also an important question. The boards are modular, meaning that each board can have its two power deliver strips removed. By only removing one strip from each board, two breadboards can be connected together and each

board will still have two power strps: one at the top and one at the bottom. The extra power strips can be connected together and used to build up the common bus at the center of the computer. Unfortunatley, due to the large size of the computer, it became clear that the strips removed from breadboards wouldn't be enough to cover the entire length. Fortunatley, *Farnell* stocked the power strips as separate items, so a few of those were purchased [39]. Given the number pins on each breadboard and the number of pins on all required chips, the number of required breadboards could be calculated. This calculation lead to the creation of a simple schematic which details how each module is organized on breadboards 4.1.

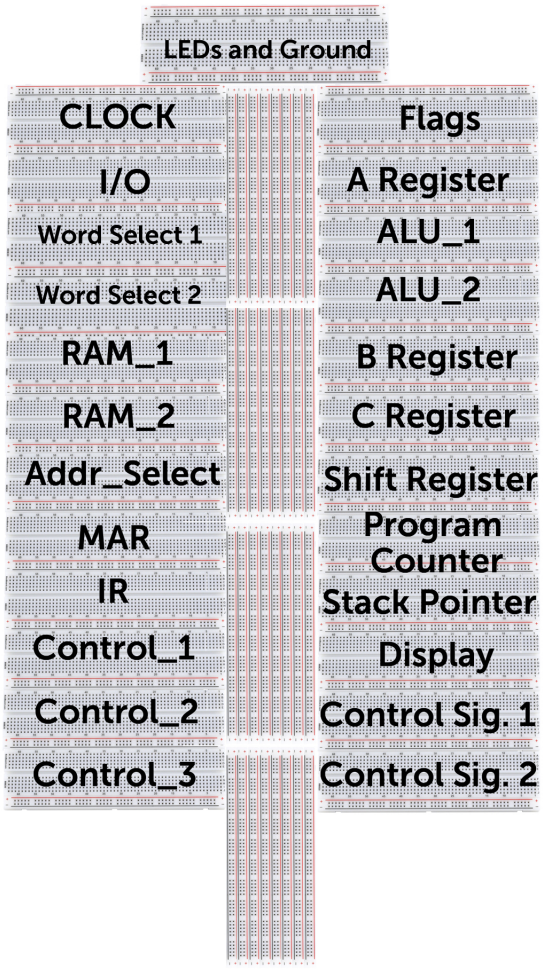


Figure 4.1: Module Layout onto BB830 breadboards

4.1.2 Wire choice

The choice of wires was mostly based on the breadboards. According to the specification provide by *BusBoard* [35], the BB830 can use wires of thickness ranging between 29AWG and 20AWG. Multiple 10 meter spools of PVC-insulated multi-coloured 22AWG monofilar copper wires were purchased. The multiple colors were mapped as follows to different functions in the computer:

- Red: Power (+5V/VCC)
- Black: Ground (GND)
- Purple, Brown and Orange: Internal module connections
- Blue: Bus connections
- Green: Control signals
- Yellow: Clock signal and Arduino Connections

Purple, Brown and Orange ran out towards the end of the build phase (Control Logic), so Green, Yellow and Blue was used.

4.1.3 Power Delivery

One important area which was glossed over during the Specification and Design phase was the power delivery. While it is convenient to assume easy access to perfect 5V voltage accross all modules, in practice this is not trivial to achieve. Besides this, there are also current limitations to take into account.

Main Power Delivery

Ben Eater uses in his build a simple 5V USB wall charger usually used for phones, from which he cut off the microUSB connector and soldered a barrel connector to the 5V and Ground leads [9]. This has also been attempted for the 16-bit breadboard computer, but it quickly became apparent that, given the many more components used in this upgraded and extended version, the current draw far exceeded the 1A current limit of the phone charger. As such, a more robust power solution was required. Since there was no laboratory power supply available, an old and discarded ATX 450 Watt power supply was used 4.2. This power supply has a current limit of 35A on the 5V rail, so it should manage to supply the 16-bit breadboard computer with as much current as needed. To interface the power supply to the computer, an ATX power

connector was de- soldered from an old PC motherboard, and then cable bundles were soldered on to the 5V and Ground pins of the connector 4.3. Those cables were then inserted into a power strip at the top of the computer, near the LEDs module 4.4. Finally, two pairs of 5V plus Ground cables were routed from the power strip to each breadboard 4.5. This ensured that each power strip was at a relativley equal resistance from the main power strip, so the voltage across strips was essentially equal. This also ensured that the breadboards would have enough current draw available, as each module was connected more or less directly to the PSU. After testing random voltage samples across all breadboards using a multimeter, the average voltage was found to be around 4.8 Volts, which is well within the specification of the chips used (the specification usually states that the chips work in a 4.75 - 5.25 Volts voltage range).

Figure 4.2: 450W ATX Power Supply used to provide 5V power to the breadboard computer

4.1.4 Auxiliary Power Delivery

Besides the main power delivery, there is another aspect which needs to be taken into account when discussing adequate power delivery to all modules. An integrated circuit, when powered but not in active use, draws a relatively stable amount of current. Whenever the circuit has an input change, the transistors inside the circuit switch state. If many transistors switch state at approximatively the same time, this can lead to noticable increases in momentary current draw. To mitigate these current spikes, each chip is outfitted with a small 100 Nano-Farad ceramic capacitor across its power and ground pins. These capacitors act like small batteries, storing up charge when the circuit is stable and not drawing excessive power and then releasing that charge when the current spikes occur. Besides the 100nF capacitors, larger capacitors were desoldered from the motherboard from which the ATX power connector was harvested and

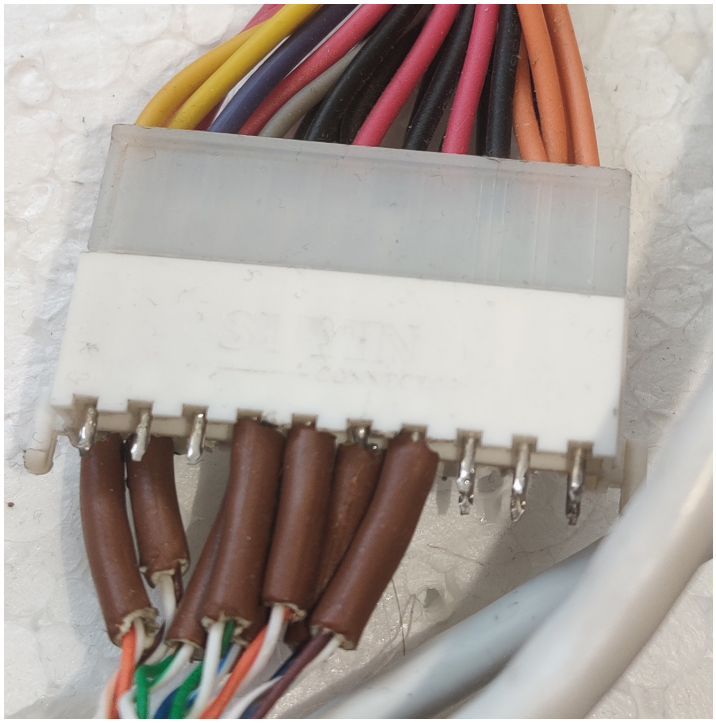


Figure 4.3: ATX connector salvaged from an old motherboard

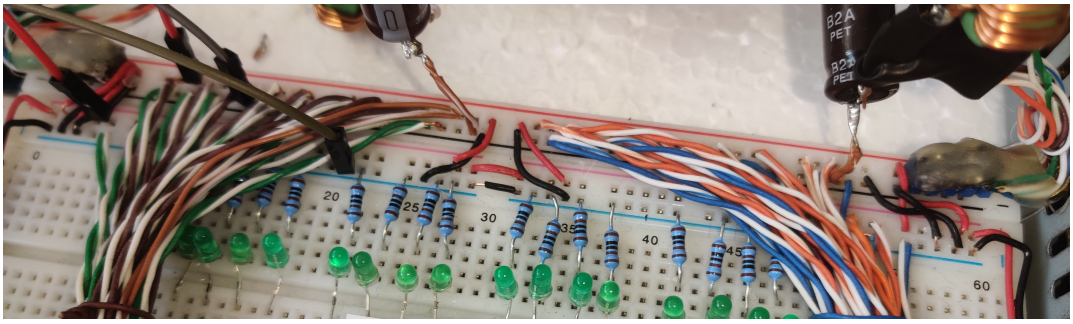


Figure 4.4: Power Strip connecting power cables from the salvadged ATX power connector

then inserted at regular intervals across the power strips of each modules 4.6. These capaitors act just like the smaller oanes, just at a larger scale, mitigating current spikes for multiple integrated circuits at the same time.

4.1.5 Sockets and Resistors

Sockets There were to areas where the implementation of the 16-breadboard computer slightly deviated from the schematics designs. First of all, *Zero-Insertion-Force* (ZIF) Sockets were added between the EEPROM chips on the control logic module 4.7. These sockets use a small lever to lock the pins of a chip in place. The chip is simply placed into the socket and then the lever is pulled to tighten the chip in place. If the chip is to be removed from the socket, the

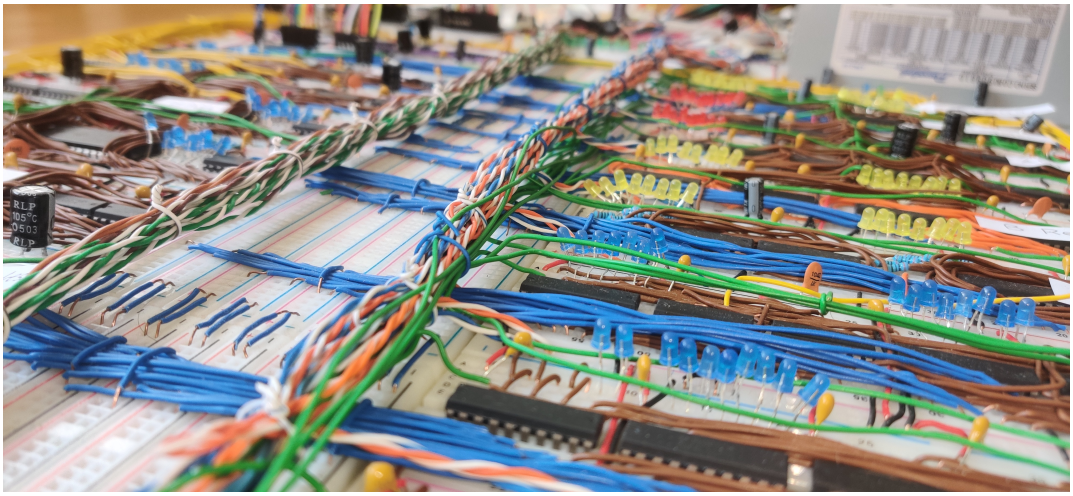


Figure 4.5: Power cable bundles spreading out across all breadboards

lever is pulled back and the chip can be removed without the need to apply any force (hence the name ZIF). The reason why the sockets were used was that the EEPROM chips were to be removed and re-inserted into the sockets many times, so as to re-program them with new or altered instructions. This would have put a large strain on the breadboards, which could have led to contact issues fairly quickly, an issue which would have been hard and time-consuming to debug.

Resistors Normally when installing an LED, it is important to use a resistor in series between the LED and either the ground or the voltage lead. This prevents the LED from drawing too much current and burning up. In many instances across multiple modules, there was not enough space on the breadboards to fit both LEDs and resistors. As such, in those ares the resistor was not installed and instead the LED was connected directy between the voltage source and the ground lead. While this is definetly bad practice and it was avoided as much as possible, this compromise ws made where necessary because the LEDs were always connected to a chip output and not to the 5V rail directly. The chips from the *74LS* family have a very limited maximum ouput current. This current was not high enough to burn up the LED. As such, the LEDs could be connected directly to the chips and then to Ground without the fear of having them burn up. This was not without consequences though. The small maximum output current of the *74LS* chips was still larger than if it would have been limited through a resistor. As such, the overall current draw of the computer increased. Given the robust power solution put in place, this didn't cause any issues.

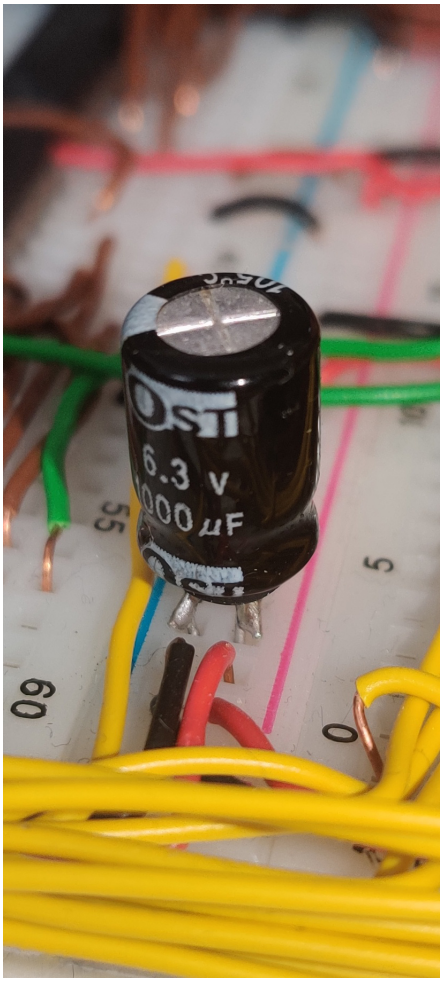


Figure 4.6: Power filter capacitor placed across breadboard power strips

4.2 Hardware Implementation Tools

Since the breadboards don't require solder to form connections, very few tools are actually needed to build on them. A pair of pliers and side cutters were used to cut off the isolation on the ends of wires to form pins which can then be inserted into the breadboard. In case it is availalbe, a wire stripper can also be used instead. It would be preffered, since a wire cutter is much more reliable at producing equally sized pins. A small flat-head screwdriver can be used to extract wires and chips which have been wrongly positioned. The breadboards come preapplied with an adhesive backing. To separate the breadboards from the power strips, a cutter was used to cut through this backing. An a few occasions, a soldering iron and solder was needed (for example to extract components from the donor motherboard). Finally, a multimeter was used to ensure that connections have been made correctly and to measure voltages.

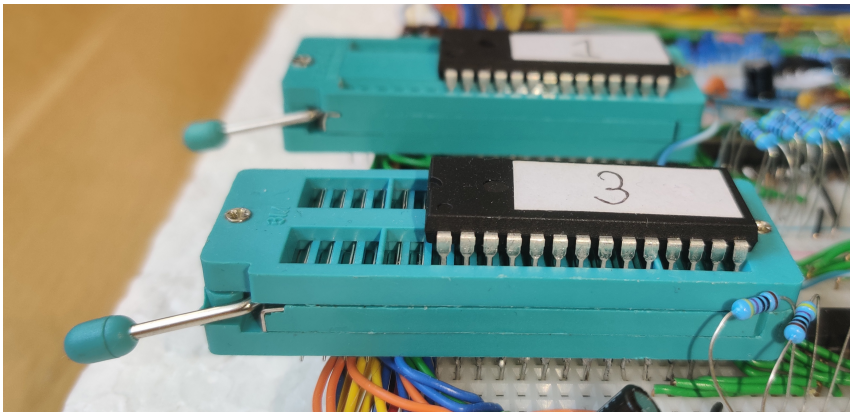


Figure 4.7: AT28C64B [1] EEPROM chips inserted into *ZIF* sockets

4.3 Implementation process for one module

The implementation process for a 16-bit breadboard computer modules is straight forward and easy to follow.

1. Ensure all needed tools and components (breadboards/ICs/wires/capaitors/LEDs) are at hand, as well as the design schematic for that particular module
2. Lay out the chips on the breadboard
3. Based on the connections made in the schematic, measure out wires to connect the pins of the different integrated circuits
4. cut the wires to that length, plus an added centimeter for the two pins, then strip out 5 milimeters of insulation off of each end to form the pins
5. connect the wires to the breadboard
6. plug the LEDs and Capacitors into the breadboard
7. Connect and label control signals going into and out of the module

4.4 Physical implementation

The process outlined previously process was followed thoroughly for each module of the 16-bit breadboard computer. The modules were built in the following order:

1. **Clock Module** 4.8
2. **A Register** 4.9

3. **B Register** 4.10
4. **C Register** 4.11
5. **Instruction Register** 4.12
6. **Memory Address Register** 4.13
7. **Flags Register** 4.14
8. **I/O** 4.15
9. **ALU** 4.16
10. **Shift Register** 4.17
11. **Program Counter** 4.18
12. **Stack Pointer** 4.19
13. **Word Selector** 4.20
14. **Address Selector** 4.21
15. **RAM** 3.3.10
16. **Display** 3.3.13
17. **Control Signals** 4.24
18. **Control Logic** 4.25
19. **High level overview (with bus and LEDs visible)** 4.26

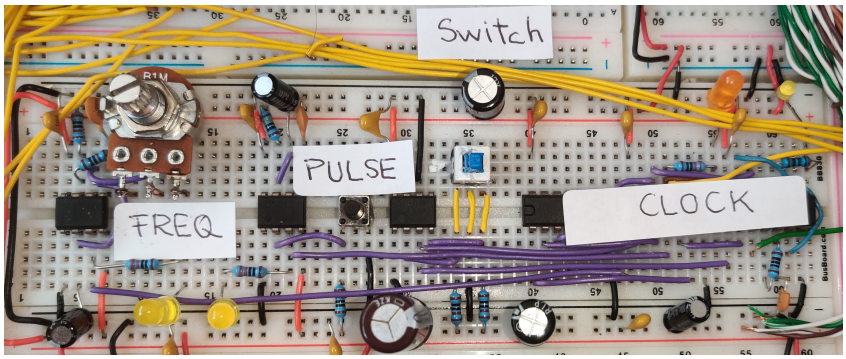


Figure 4.8: Clock implementation

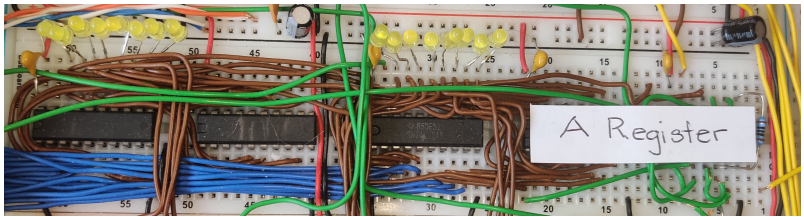


Figure 4.9: A register implementation

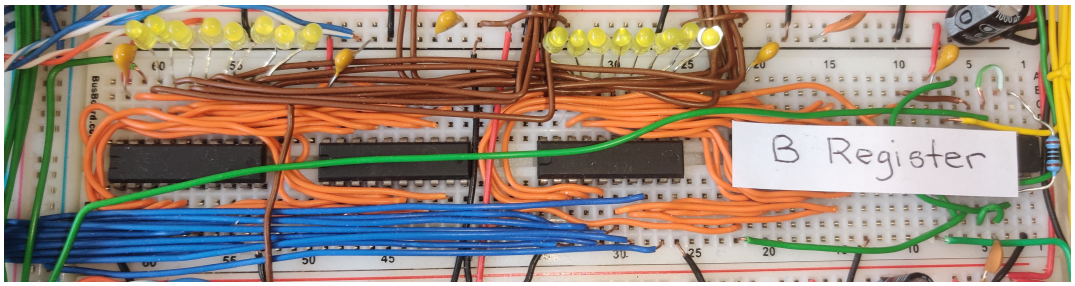


Figure 4.10: B register implementation

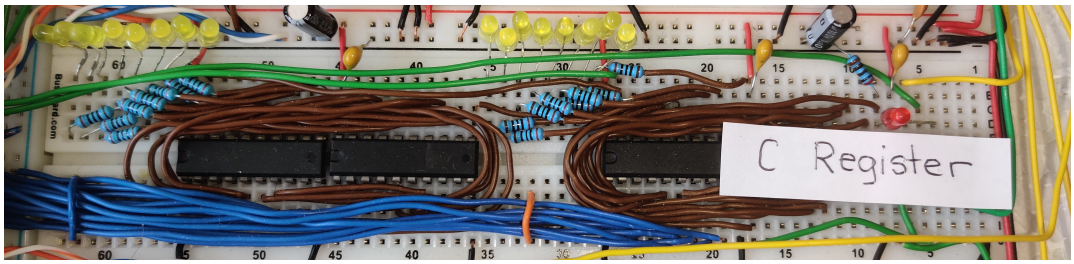


Figure 4.11: C register implementation

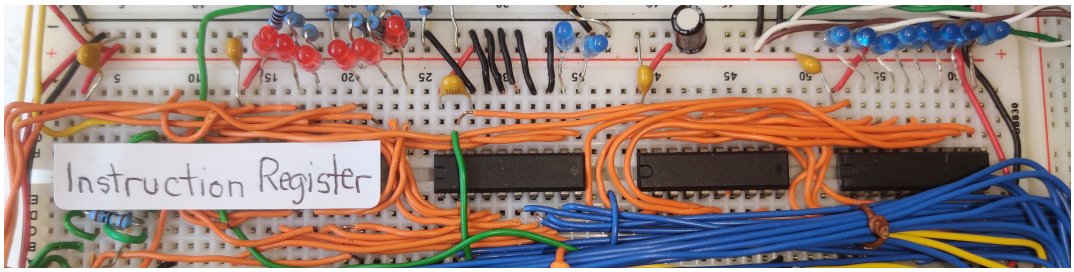


Figure 4.12: Instruction register implementation

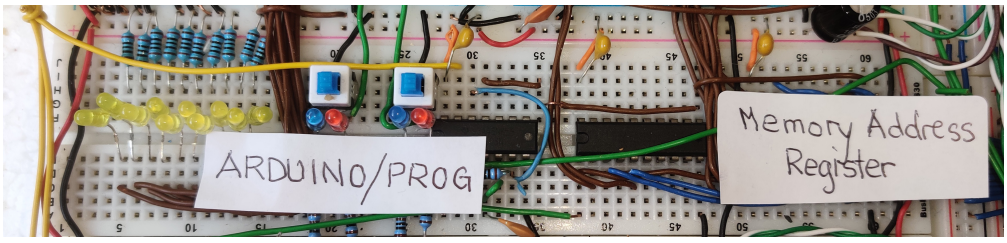


Figure 4.13: Memory address register implementation

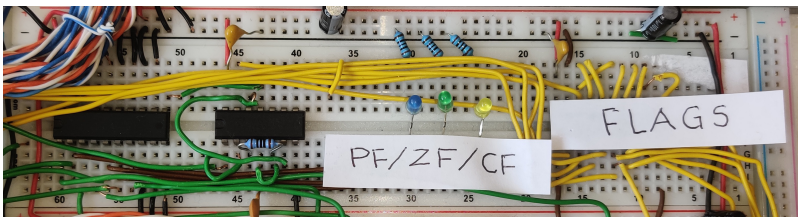


Figure 4.14: Flags register implementation

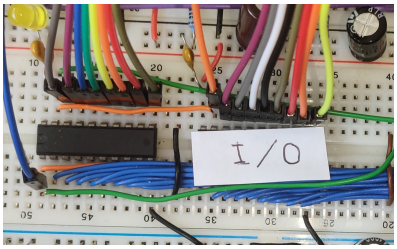


Figure 4.15: Input/Output implementation

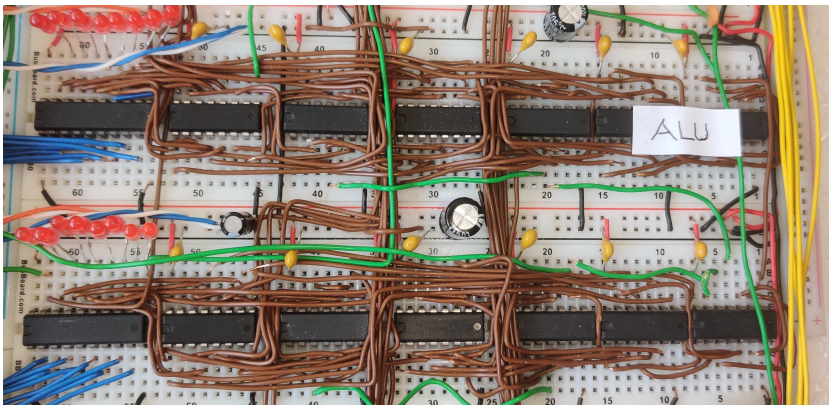


Figure 4.16: Arithmetic-Logic Unit implementation

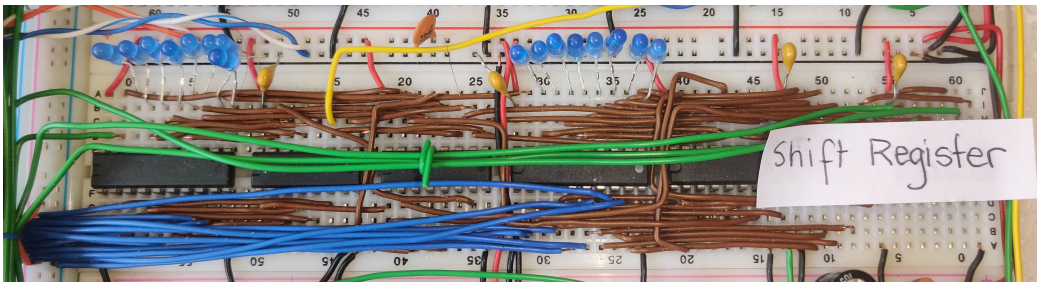


Figure 4.17: Shift Register implementation

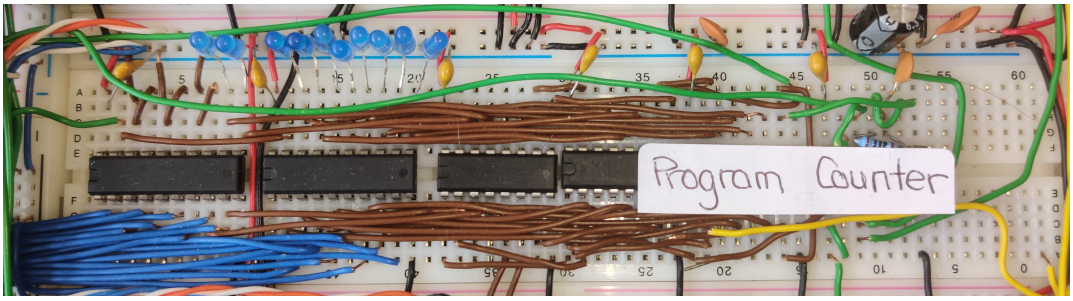


Figure 4.18: Program Counter implementation

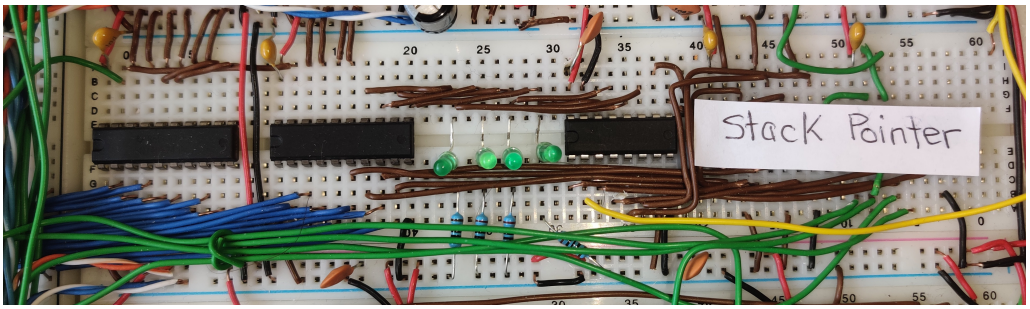


Figure 4.19: Stack Pointer implementation

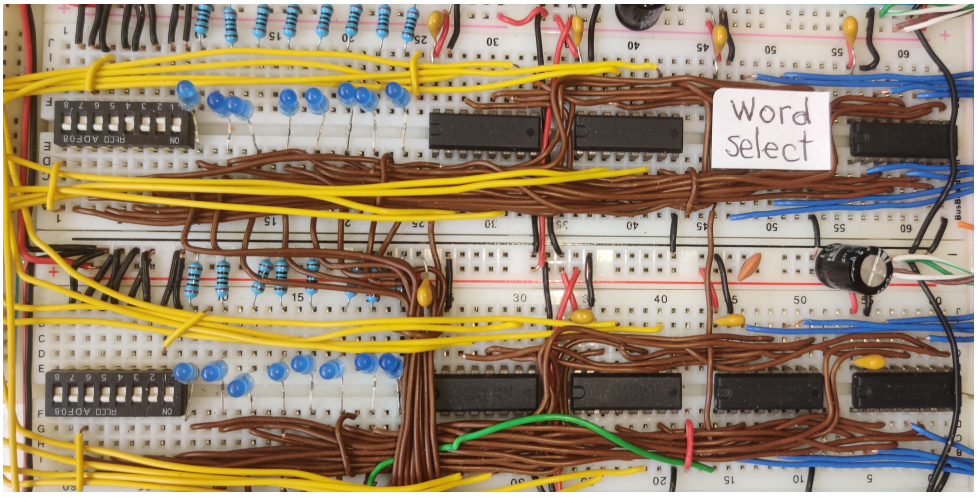


Figure 4.20: Word selector implementation

Figure 4.21: Address selector implementation

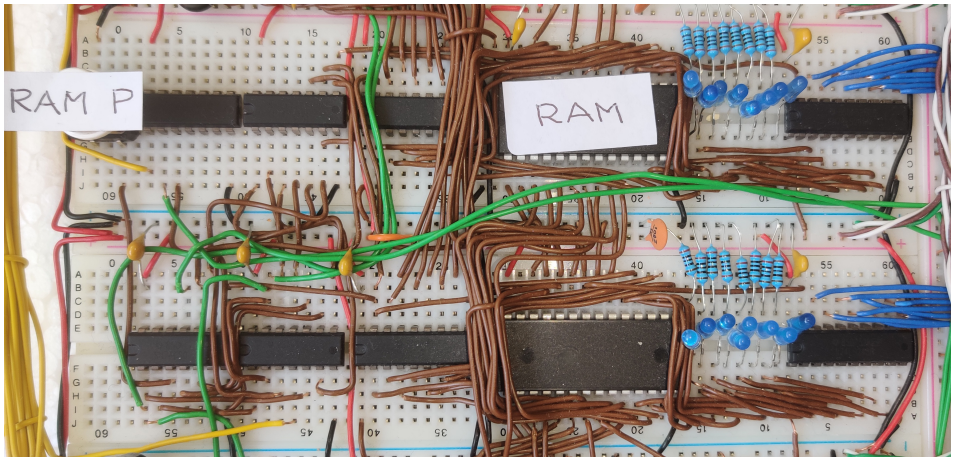


Figure 4.22: Random Access Memory implementation

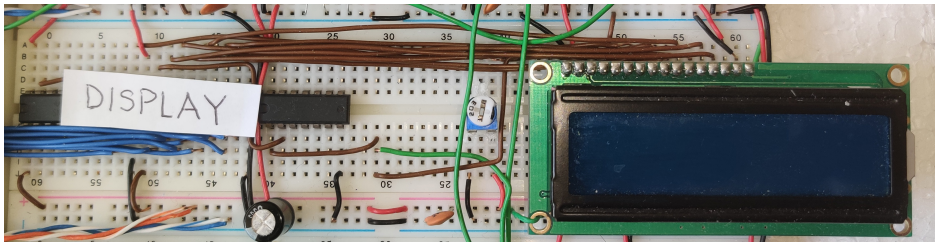


Figure 4.23: Display implementation

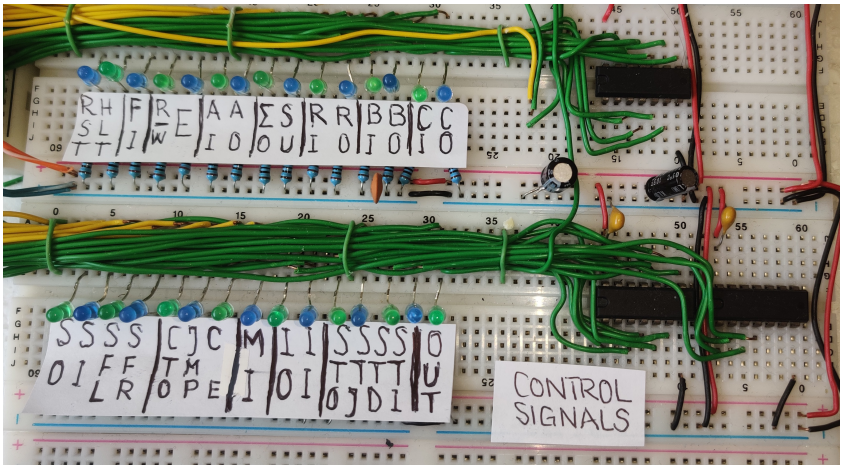


Figure 4.24: Implementation of the contol signals module

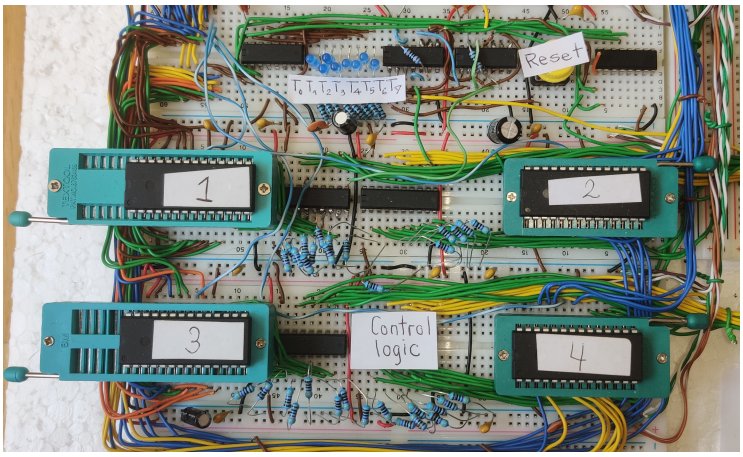


Figure 4.25: Control Logic implementation

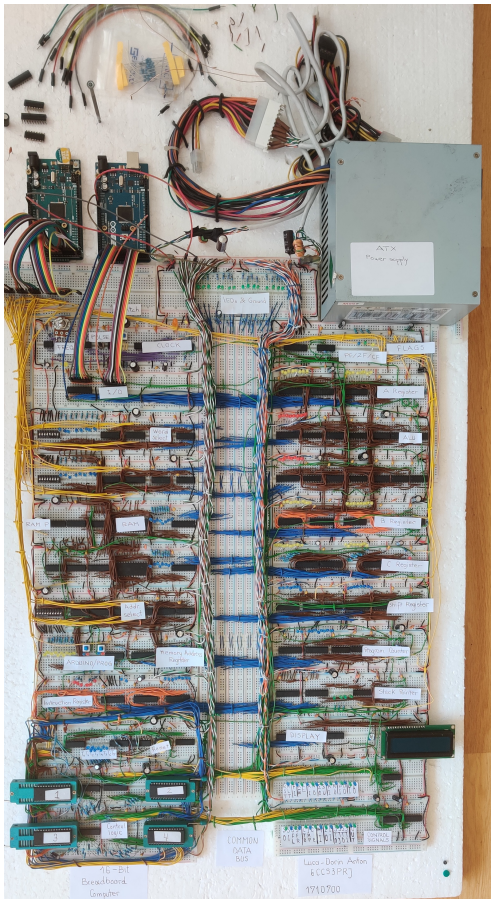


Figure 4.26: High level overview (with bus and LEDs visible)

After the first few modules and with enough power strips gathered, the first segment of the data BUS was created. After that, all modules created so far were attached to the bus. From that point on, after each module was built, it was attached to the bus. After a module was attached to the bus, the direct connections to other modules which might exist were created. Initially, control signals were connected with jumper wires, so that they could be manually turned on and off. This was usefull for testing.

4.5 Module Testing

After the implementation of a module was completed, the next stage was to test it to ensure it meets its specification. This was done by manually setting the control signals to either high or low. To test using specific bus inputs, 16 jumper wires were connected to the bus, 1 to each bud line, and then manually connected on a power strip to either high or low. This was used to input bit patterns to a module to test that each bit was mapped correctly from the bus to the module input and from the module output to the bus. Each module was tested this way.

4.6 Implementation issues

Implementation of most modules went through without significant issues. The main issue which initially arrised was the power delivery. This prompted the switch from the simple phone charger to the ATX power supply solution. Towards the end of the hardware implementation stage, a major issue appeared.

EEPROM issues

After finalising the construction and connection of each module except for the control logic module, all modules behaved as expected. Data could be read to and from the bus, the ALU would add up number correctly, the program counter counted only when instructed, the display could receive and dispplay instructions, RAM could be addressed correctly and manually programmed with success. After the installation of the control logic module, issues started to arrise. Modules would behave eratically, the counter on the clock module would count twice on every count cycle, it would skip bits, ot sometimes not even count at all. Modules would read from the bus when they weren't instructed to and sometimes short 'glitches' appeared on the LEDs module which monitored the state of the bus. This issues would be removed if all the AT28C64B [1] EEPROM chips would be removed from their sockets. This prompted an

investigation which ended up taking over **one week of continuous work,** before the root of the problem was found. After many different theories and patches without any effect, the issue ended up being not with the circuit, but with the EEPROM chips themselves. According to the documentation, the chips have a 150 nano- second time window after each address change in which they set up the output of the data contents for the new address. During this very short time window, the output of the chips is "undefined" It seems that during this period, the undefined behaviour can equate to significant voltage spikes. This voltage spikes, albeit short, are long enough to momentarily change the state of the chips their are connected to (most *74LS* are rated for clock speeds of in the tens of MHz). This caused the unexpected behaviour. The sollution was a combination of multiple patches:

- *Control Signals buffer chips:* Directly after the EEPROM chips, 4 *74LS245* [21] buffer chips were installed. These chips have their advantage that their output is only logic 1 or 0, nothing in between, and that they amplify the signal going into them

- *Pull-down Resistors:* 470 Ohm pull-down resistors were installed between the buffer chips and the EEPROM chips. This ensured that if the EEPROMs would have to drive to a relatively large voltge for a logic 1 input to be detected by the buffers

- *RC filter networks:* On the most critical control signals *RC networks* (Resistor - capacitor) networks were installed to filter out any potential voltage spikes. This circuits work by employing a very small capacitor which charges up very fast 4.27. This way, if a low frequency signal passes through the network, like the signals which are intended to be generated by the control logic module, it will quickly charge up the capacitor and then continue to propagate further on. This is intended. If a high frequency signal passes through the network, like for example the voltage spikes generated by the EEPROMs during address change, the signal won't have time to charge up the capacitor fully, and as such it won't propagate further thorugh the circuit. This filter circuit is then clamped between to gates (for example AND gates with inputs tied togheter or inverters) which are unused on the module where the singal is required, so as to ensure that the signal going in and out of the filter circuit is still digital. Then, the entire circuit is placed in series with the signal which it is supposed to filter. In practice in this case, 100 Ohm resistors coupled with 10nF capacitors were used. These RC networks were placed on the *CLK, RST, DONE, CE, AI, BI, II, MI* signals **??**.

With this mitigation measures in place, the computer started behaving normally again.

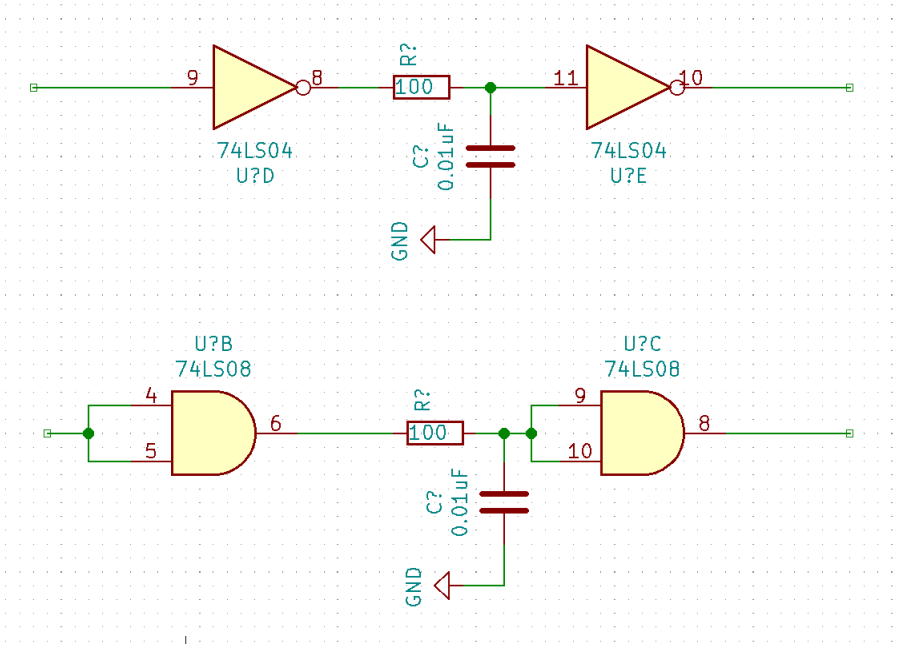


Figure 4.27: Example of AND-clamped and inverter-clamped RC networks in KiCad

Chapter 5

Software Specification

To transform the 16-bit breadoard computer from an exercise in logic design into an advanced and rich learning meadium, a suite of feature-rich software tools is needed.

5.1 Main functionailty to be implemented

The software for the breadboard computer should implement the following pieces of high-level pieces of functionailty to be considered successfull:

- *Microcode definition and re-programming:* The computer needs the control logic EEPROM's 3.3.19 to be programmed with microcode. There should also be an accessible way to define and modify microcode.
- *Assembler:* Given a set of instructions defined through microcode, there should be a way to write assembly instead of plain binaries and then assemble the code produced down to binaries
- *Compiler:* Going up one more level of abstraction, one should be able to write a program in the WHILE programming language and then compile it down to either assembly or binaries directly.
- *Programming arduino drivers:* Currently, the computer is programmed by setting the *PROG* switch and then manually flipping dip switches on the word selector 3.3.17 and the address selector 3.3.18. This process is tedious, prone to mistakes and it can take a relativley long period of time to program a small number of instructions. Since an

Arduino is connected to the word and address selectors, there should be a way to have th arduino program a binary produced by the assembler.

- *I/O Arduino drivers:* While the I/O 3.3.14 module has been built to its specification, it relies on the Arduino handling its read and write request. To achieve this, some driver software needs to be created.

The specification for each piece of software will be detailed in the following sections.

5.2 Microcode Definition and Programming

5.2.1 Microcode format

Currently, the only way to define instructions for the 16-bit breadboard computer is to write binaries by hand, and then program them to the control logic EEPROMs. This is tedious and prone to errors. There should exist a format through which instructions for the 16-bit breadboard computer can be designed and stored. This format should be simple to read, understand, write and change. The format should be high-level enough, so that the instruction designer doesn't have to think about setting individual bits, while still being able to thik about which control signals are set at which time step during which flag conditions. The format should specify the following pieces of information regarding an instruction:

1. Mnemonic (three-letter code)
2. Opcode (6-bit number)
3. Description (String)
4. Whether the instruction requires an operand or not (boolean)

Following this, each instruction should have associated to it at least one execution path. This path should be dependant on the flag environment, but there should be at least one path to serve as a default in all flag environments conditions which are not specified. An execution path consists of a list of steps, each step detailing which control signals should be activated at that particular time step.

5.2.2 Parsing and code generation

After a file declaring instructions conforming to the format described above is created, a script should exist which parses that file and generates 4 binaries from it, one for each EEPROM.

5.3 Assembler

Given a microcode generation tool, while it is now easy to write an *instruction set* for the 16-bit breadboard computer, one would still have to write *programs* in binary format. The first level of abstraction away from writing binaries by hand should be an assembler. The Assembler should fulfill the following requirements:

1. Dynamically recognize mnemonics based on a provided instruction definiton file
2. Recognise labels and replace them with memory locations
3. Recognise aliases and replace them accordingly
4. Recognise operands in decimal, hexadecimal and binary format
5. Generate binaries based on an assembly source file and an instruction definiton file

5.4 Compiler

To have a thorough understanding of the requirements and specifications for the *WHILE* language compiler, the language must be first presented and discussed.

5.4.1 WHILE language

The *WHILE* language is a very simple Turing-complete programming language which is in syntax very similar to pseudocode 5.4.1.

Listing 5.1: *WHILE* language Fibonacci program

```
read n;
minus1 := 0;
minus2 := 1;
while n > 0 do {
        temp := minus2;
        minus2 := minus1 + minus2;
        minus1 := temp;
        n := n - 1
};
write minus2
```

Some of the most important features of the *WHILE* language are:

- *Limited types:* the language only has two types *strings*, which can only be printed out, and *integers*
- *Simple statements:* the statements possible in the while language are *while, if, assign, read, write*
- *Restricted boolean and arithmetic expressions:* The while language an process data by executing instructions using simple arithmetic and boolean expressions. The allowed arithemtic operators are $+, -, *, /, \%$. The allowed boolean operators are $\&\&, ||, \leq, <$ $, ==, >, \geq$
- *Simplified I/O:* The *WHILE* language uses the read and write statements to abstract the process of reading and writing data.

5.4.2 Requirements

Given the feature list of the *WHILE* programming language described in the previous section, a compiler should fulfill the following requirements:

- Parse *WHILE* source files
- Understand the underlying structure of the program
- Generate assembly code for a certain instruction set for the 16-bit breadboard computer which is built into the compiler.

5.5 Programming Arduino driver

In order to program the 16-bit breadboard computer with a binary generated by the assembler, an Arduino driver coupled with a script to run on the main computer. The two pieces of software should work in tandem to transmit a source file to the arduino and then on to the breadboard computer. This should work as follows.

5.5.1 Protocol specification

1. The Arduino is connected to the computer over USB
2. The Arduino is manually reset

3. The Arduino maintains an internal address counter, reset to 0
4. The computer-side script reads a binary file and stores it in memory
5. The computer sends the arduino two bytes at a time (two bytes form 1 breadboard computer instruction)
6. The Arduino receives the 2 bytes and constructs a data word from them
7. The Arduino programs the byte in the breadboard PC RAM at the address specified by its counter and then increments the counter
8. The Arduino replies to the computer with a success message
9. After the computer receives the success message, repeat from step 5

5.6 I/O Driver

The I/O driver should work in a similar manner to the programming driver. It should allow the the breadboard computer to interface with a personal computer by using the Arduino as an information relay.

5.6.1 Protocol Specification

- The Arduino is connected to the computer over USB
- The Arduino is manually reset
- The Arduino maintains an internal circular queue of 10 16-bit data words, which is initially reset to 0.
- If the computer sends data over to the Arduino, it should store that data at the end of the queue and update its queue pointer.
- If the breadboard computer triggers a read request, the Arduino should serve it a data word from the queue and then update its queue pointer.
- If the breadboard computer triggers a write request, the Arduino should read the data word found in the I/O module and then send it on to the main computer.
- The main computer should then display that data on the screen.

- At the start of the script on the main computer, a file should be provided which contains 10 numbers in either decimal, hexadecimal or binary format. This numbers should form the basis for the queue to be sent to the Arduino

Chapter 6

Software Design

6.1 Microcode format and programmer

6.1.1 Design Choices

Language choice Given the fact that the microcode programmer will be tasked with processing both text and binary files, *Python* [15] comes up as an obvious choice for an implementation language. Python makes file manipulation and text processing enjoyable instead of frustrating, as in many other languages, because it requires essentially no boilerplate to achieve those goals. Moreover, Python is an implements object-oriented philosophy, which will facilitate the creation of an internal model for the instructions which are to be programmed. The syntax is also light weight, concise and understandable, which makes *Python* a good choice for a scripting language as well. This is important because the microcode programmer will be a command-line script.

Microcode Instruction Format Given the specification for a simple to learn and used format for declaring microcode instructions, two possiblities come to mind:

Language specific One approach would be to create classes for the microcode instructions in some object-oriented programming language. This way, a user who is familiar with the language could easily create multiple objects which are instantiations of that particular class. After the user has created the class instances, he could run a script to generate microcode instructions from them. Since the implementation language chosen is *Python*, Instructions would be represented as Python objects.

Language agnostic Another approach would be to make use of a format which doesn't depend on a specific programming language. The main advantage of this approach would that someone who isn't familiar with the implementation language could use the tool efectivley. This would also mean that the programmer would have to contain a parser for this format, separate from the parser of the implementation language. A good choice of format for such an approach might be *YAML* [37]. *YAML Ain't Markup Language* (recursive definition), is a data serialization language designed around the main goal of being human-readable [37]. As such, it poses itself as a great choice for the microcode definiton format, as it is so simple that one does not have to be familiar with it to quickly undestand it and use it. Both approaches towards formatting microcode instructions should be implemented.

6.1.2 Programmer design

Given a file in either of the two formats discussed in the previous section, the programmer should follow the following broad steps

1. If provided a Python source file, parse it and extract the instruction objects
2. If provided a YAML file, parse it and then convert the YAML objects to Python instruction objects
3. Generate 4 bytearrays containing zero bytes. The bytearrays should be of the size of the memory arrays of the control logic EEPROMs
4. For each instruction, for each flag environment, for each step:
5. Concatenate the binary formatting of the opcode, flag environment and time step
6. Interpret the resulting string as a binary number and convert it to an integer
7. Use the resulting integer as a pointer to write the control signal bytes of that timestep to the right place in the bytearrays
8. If no control signal sequence is provided for a certain flag environment, use the default one
9. Write the resulting bytearrays to files

6.2 Assembler

6.2.1 Design choices

Language For the assembler for the 16-bit computer the choice was made to use a different language then the one used for the microcode programmer, namely *Scala* [45]. *Scala* is a modern multi-faceted multi-paradigm all-purpose built on top of the Java ecosystem. *Scala* is large-scale widley-used and fully-featured to such an extent that it would be infeasable to cover all of its features in this report. However, noting the most important language features will shed light onto why it poses itself as a great tool for implementing assemblers and compilers:

Functional programming Scala has all the functionality needed to implement programs and scripts following a functional paradigm. This feature lends itself to the advantage in the task of implementing an assembler and compiler because of the natural tendency to solve assembling and compiling tasks using recursive functions. Functional programming encourages and simplifies the creation of recursive functions.

Java Ecosystem Since *Scala* is built on top of *Java* [36], the entire *Java Ecosystem* is a subset of *Scala*. As such, *Scala* doesn't confine the developer to functional programming, allowing great freedom in mixing and matching styles and paradigms.

Robust yet Concise Typesystem *Scala* comes with all the benefits of the *Java* typesystem, including stron typing, type checks, object oriented approach, subclasses, abstract classes and many other, while at the same time not having the same downsides. The main advantage of the *Scala* typesystem is the fact that it requires significantly less boilerplate than its *Java* counterpart. This feature is useful given the inherent need of compilers and assemblers generally to create many internal data structures to faciliate the transformation of source code from one format to the next.

Dynamic and flexible Syntax The *Scala* syntax is concise and understandable, similar to Python in many ways. Unlike Python syntax, it has some key advantages. It doesn't rely on indentation to determine scope, which can be a source of frustration allthough it makes code much more readable and it gives the user the option to extend and augment the syntax using implicit definitions. This makes the life of the assembler and compiler designer easier by allowing him to write shortcuts in places were repetitive tasks are undertaken.

YAML Parser Since the assembler will have to be aware of any changes in instruction set defined by the user for the 16-bit breadboard computer, the need for a YAML parser arrises. This parser will be used to process the same list of instructions used by the microcode programmer and then collect a list of mnemonics. Using this list, the parser can determin if the assembly source file conforms to that particular instruction set.

6.2.2 Assembler Design

The Assembler will be built around a two-module design. A *Lexer* and a *Code generator*.

Lexer The task of the *Lexer* is to transform the contents of an assembly file, essentially a simple string, into a list of tokens. Then, the *Lexer* can pass on the token list to the *Code generator*. The *Lexer* uses *regular expressions* to read the input string and break it down into tokens. When a piece of the input string matches a particular token pattern, an instance of that token is created with that specific piece and then added to the list. Then the *Lexer* continues to process the remaining input string. Besides this, the *Lexer* also has the task of eliminating whitespace and comments from the token list, as these play no role in code generation. The lexer contains a custom-built regular expression engine built around the concept of *regular expression derivates*. This alternate way to think about and implement regular expressions is taught by *Dr. Christian Urban* in the **6CCS3CFL** course[42]. The use of derivative regular expressions significantly improves performance in certain edge cases. When a match is found, an injection function is used to rebuild a "value" from the derivative which points towards which token has matched.

Code generator The *code generator* receives the list of tokens from the *Lexer* and then does further processing on it. Since assembly doesn't have a complex syntax or structure, there is no need for a parses. The assembler completes the following steps to fullfil its task:

1. Alias extraction
2. Subroutine ordering
3. Subroutine separation
4. Reference solving
5. Instruction decoding

Once these steps are completed, the code generator is able to generate a bytearray and write a binary file to disk.

6.3 Compiler

6.3.1 Design Choices

Language Since the Compiler and the Assembler share a significant amount of structure and code, the choice was made to also use *Scala* for the compiler. The reasons for this choice have been stated in the Assembler design choices section 6.2.1.

Structure The compiler has a similar structure to the assembler. It also has a *Lexer* and a *Code Generator*. The main difference is the presence of a third module, namely the *Parser*. The *Parser* sits between the *Lexer* and the *Code Generator* and is tasked with generating an *Abstract Syntax Tree* from the list of tokens.

6.3.2 Compiler Design

Lexer The compiler *Lexer* follows the same design pattern as the one used for the assembler. The main difference between the two is that they use different regular expressions to parse the input string.

Parser The *Parser* has the task of taking the token list, which has no structure except for the total ordering imposed onto it and then turn it into a *Abstract Syntax Tree*, or *AST* for short. An *AST* encapsulates the internal structure of a program, like for example nested loops, if-else branches and the order of operations in an arithmetic expression. The *Parser* of the compiler is based on a parsing philosophy called "parser combinators", which is also taught by *Dr. Christian Urban* in the **6CCS3CFL** module [42]. Parser combinators work by defining simple as objects which try to parse as much of the input tokens as possible and then return the amount they could parse, as well as the remaining part of the input which they couldn't process. By taking multiple simple parsers and *combining* them (hence the name), a more complex parser can be created. Their nature is to return a set of pairs of processed and unprocessesed input. This way, the next parser in line can try to parse based on all the different ways the previous parsers parsed. When the infrastructure has been set up, writing parser combinators for a language can be as simple as declaring the grammar of that particular

language. Parser combinators make use of *lazy evaluation* which is a mode in which arguments are only evaluated at the last possible moment in time.

Code generator The code generation stage for the Compiler is simplified, since most of the processing has been completed by the *Parser*. In this case, the *Code generator* has to traverse an abstract syntax tree recursivley and then map its elements to chunks of code from the target language, in this case 16-bit breadboard computer assembly.

6.4 Programming and I/O Arduino Drivers

6.4.1 Design choices

Language For the design of the programming and I/O Arduino drivers, *Python* was chosen for the script on the computer side and *C* was chosen for the Arduino script. *Python* is a great scripting language and it handles file I/O, both binary and text, in an elegant manner. So it presented itself as a good choice for the scripts running on the computer. *C* was chosen for the Arduino scripts because it is the only language supported by the Arduino Integrated Development Environment [14].

6.4.2 Script Design

The scripts for running the Arduinos do not offer much in term of design depth, since they are relativley short and simple. What is notable is the need to use interrupts to interface with the breadboard computer and the use of the serial port to communicate to the computer. The Arduinos also make extensive use of bitshifting and bit extraction to convert data from the internal datatypes to voltages on multiple pins and pin voltages to datatypes.

References

[1] Atmel. At28c64b parallel eeprom with page write and software data protection. http://ww1.microchip.com/downloads/en/DeviceDoc/doc0270.pdf, 2020. Accessed: 2020-03-04.

[2] Eater Ben. Making logic gates from transistors. https://www.youtube.com/watch?v=sTu3LwpF6XI, 2015. Accessed: 2020-03-03.

[3] Eater Ben. Conditional jump instructions. https://www.youtube.com/watch?v=Zg1NdPKoosU, 2018. Accessed: 2019-12-13.

[4] Eater Ben. 8-bit computer high level architecture diagram. https://eater.net/8bit/schematics, 2019. Accessed: 2019-12-13.

[5] Eater Ben. Building an 8-bit breadboard computer! https://www.youtube.com/playlist?list=PLowKtXNTBypGqImE405J2565dvjafglHU, 2019. Accessed: 2019-07-15.

[6] Eater Ben. 8-bit computer complete parts list. https://eater.net/8bit/parts, 2020. Accessed: 2020-03-04.

[7] Eater Ben. Why build an entire computer on breadboards? https://www.youtube.com/watch?v=fCbAafKLqC8, 2020. Accessed: 2020-03-04.

[8] Philippe Coussy and Adam Morawiec. *High-level synthesis: from algorithm to digital circuit*. Springer Science & Business Media, 2008.

[9] Ben Eater. 8-bit cpu reset circuit and power supply tips. https://www.youtube.com/watch?v=HtFro0UKqkk&t=814s, 2017. Accessed: 2020-03-04.

[10] Ben Eater. Clock module. https://eater.net/8bit/clock, 2020. Accessed: 2020-03-04.

[11] Mouser UK Electronics. Mouser uk electronics. https://www.mouser.co.uk/, 2020. Accessed: 2020-03-04.

[12] Farnell. Farnell. https://uk.farnell.com/, 2020. Accessed: 2020-03-04.

[13] Arduino Foundation. Arduino mega rev3. https://store.arduino.cc/arduino-mega-2560-rev3. Accessed: 2020-03-16.

[14] Arduino Foundation. Arduino ide. https://www.arduino.cc/en/main/software, 2020. Accessed: 2019-12-8.

[15] The Python Foundation. The python programming language. https://www.python.org/, 2020. Accessed: 2019-12-8.

[16] Lynotek Inc. 32k x 8 bit low power cmos sram. http://www.farnell.com/datasheets/1674430.pdf?_ga=2.20786168.1760150799.1587129410-1339816523.1584103728&_gac=1.45988182.1584104976.Cj0KCQjw3qzzBRDnARIsAECmryp7Xm_L8n0v0L4x40nmS9pNfCX4L14vdWPPvpZ51Fykmg1E0xJCjQ8aArv0EALw_wcB. Accessed: 2020-03-03.

[17] Texas Instruments. 3-line to 8-line decoders/de-multiplexer. https://www.ti.com/lit/ds/symlink/sn54ls138-sp.pdf. Accessed: 2020-03-03.

[18] Texas Instruments. 4 bit bidirectional universal shift register. http://www.ti.com/lit/ds/symlink/sn54s194.pdff. Accessed: 2020-03-03.

[19] Texas Instruments. 4-bit binary full adders with fast carry. http://www.ti.com/lit/ds/symlink/sn54ls283-sp.pdf. Accessed: 2020-03-03.

[20] Texas Instruments. 555 precision timers. http://www.ti.com/lit/ds/symlink/ne555.pdf. Accessed: 2020-03-03.

[21] Texas Instruments. Octal bus transceivers. http://www.ti.com/lit/ds/symlink/ne555.pdf. Accessed: 2020-03-03.

[22] Texas Instruments. Octal d-type flip-flop with clear. https://www.ti.com/lit/ds/sdls090/sdls090.pdf. Accessed: 2020-03-03.

[23] Texas Instruments. Quad two input exclusive or gates. https://www.ti.com/lit/ds/symlink/sn54s86.pdf. Accessed: 2020-03-03.

[24] Texas Instruments. Quadruple 2-input positive and gates. https://www.ti.com/lit/ds/sdls033/sdls033.pdf. Accessed: 2020-03-03.

[25] Texas Instruments. Quadruple 2-input positive nor gates. https://www.ti.com/lit/ds/symlink/sn54ls02-sp.pdf. Accessed: 2020-03-03.

[26] Texas Instruments. Quadruple 2 line to 1 line data selector / multiplexer. https://www.ti.com/lit/ds/sdls058a/sdls058a.pdf. Accessed: 2020-03-03.

[27] Texas Instruments. Synchronous 4-bit up-down counter. http://www.ti.com/lit/ds/sdls134/sdls134.pdf. Accessed: 2020-03-03.

[28] Texas Instruments. Synchronous counter with clear. http://www.ti.com/lit/ds/symlink/sn54ls161a-sp.pdf. Accessed: 2020-03-03.

[29] Rott Jeffrey. (AES-NI) intel advanced encryption standard instructions. https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni, 2012. Accessed: 2019-12-8.

[30] KiCad. Eeschema. https://docs.kicad-pcb.org/5.0.2/en/eeschema/eeschema.html, 2020. Accessed: 2020-03-04.

[31] KiCad. Kicad - open source electronics design software. https://www.kicad-pcb.org/, 2020. Accessed: 2020-03-04.

[32] Shenzen Eone Electronics Co. Ltd. 16 characters by 2 line lcd display. https://www.openhacks.com/uploadsproductos/eone-1602a1.pdf. Accessed: 2020-03-03.

[33] Albert Paul Malvino and Jerald A Brown. *Digital computer electronics*. Glencoe, 1992.

[34] M Morris Mano. *Digital logic and computer design*. Pearson Education India, 2017.

[35] Multicomp. Multicomp bb830. http://www.farnell.com/datasheets/1751666.pdf?_ga=2.35381601.1589000560.1587628777-1339816523.1584103728&_gac=1.19338954.1584104976.Cj0KCQjw3qzzBRDnARIsAECmryp7Xm_L8n0v0L4x40nmS9pNfCX4L14vdWPPvpZ51Fykmg1E0xJCjQ8aArv0EALw_wcB, 2020. Accessed: 2020-03-04.

[36] Oracle. Java. https://www.java.com/en/, 2020. Accessed: 2019-12-8.

[37] The YAML organization. Yaml: Yaml ain't markup language. https://yaml.org/, 2020. Accessed: 2019-12-8.

[38] Bruce Schneier. *Secrets and lies: digital security in a networked world*. John Wiley & Sons, 2011.

[39] Farnell UK. Mcbb100 - breadboard, abs, 8.3 mm, 20.5 mm, 165.3 mm. https://uk.farnell.com/multicomp/mcbb100/bread-board-abs-165-3mm-x-20-5mm/dp/2764642?CMP=i-bf9f-00001000, 2020. Accessed: 2020-03-04.

[40] Farnell UK. Mcbb830 - breadboard, solderless, abs (acrylonitrile butadiene styrene), 8.5mm, 165mm x 56mm. https://uk.farnell.com/multicomp/mcbb830/bread-board-abs-solderless-56/dp/2527467?CMP=i-bf9f-00001000, 2020. Accessed: 2020-03-04.

[41] Mouser UK. Bb830 development board. https://www.mouser.co.uk/ProductDetail/BusBoard-Prototype-Systems/BB830?qs=sGAEpiMZZMtgbBHFKsFQgkU9HqdjFsiq3piHUASHS%252BU%3D, 2020. Accessed: 2020-03-04.

[42] Christian Urban. 6ccs3cfl. https://keats.kcl.ac.uk/course/view.php?id=66995, 2020. Accessed: 2019-12-8.

[43] Wikipedia. 7400-series integrated circuits. https://en.wikipedia.org/wiki/7400-series_integrated_circuits. Accessed: 2020-03-04.

[44] Wikipedia. List of 7400-series integrated circuits. https://en.wikipedia.org/wiki/List_of_7400-series_integrated_circuits. Accessed: 2020-03-04.

[45] Switzerland ÃĽcole Polytechnique FÃľdÃľrale Lausanne (EPFL) Lausanne. The scala programming language. https://www.scala-lang.org/, 2020. Accessed: 2019-12-8.