KING'S College LONDON

University of London

6CCS3PRJ Background Specification

Progress Report
Breadboard Computer Architecture

Final Project Report

Author: Luca-Dorin Anton

Supervisor: Christian Urban

Student ID: 1710700

April 18, 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 18, 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.

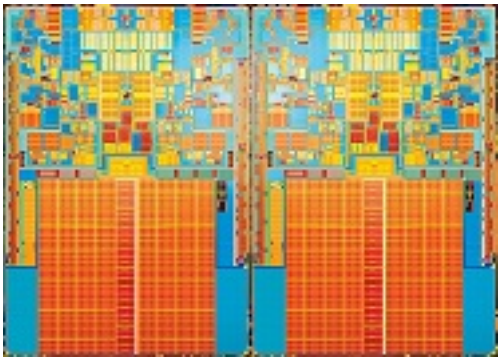


Figure 1.1: Intel quad-core 45nm CPU die

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 [20]. 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 [6]. 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. [25]

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!"* [4]. 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* [23].

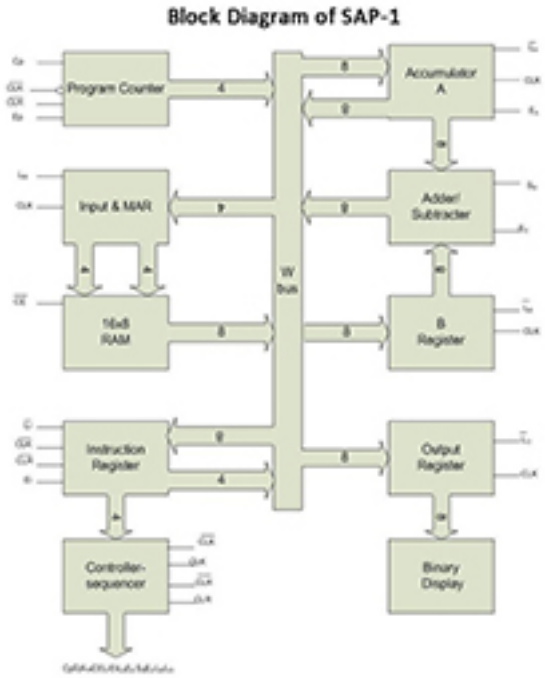


Figure 1.2: Block Diagram of the SAP-1 architecture

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 [4]. 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 build phase. A detailed chronological breakdown of build progress will be presented. Testing will be executed in parallel with the building, so testing documentation will be found in this chapter as well.

With the build 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.

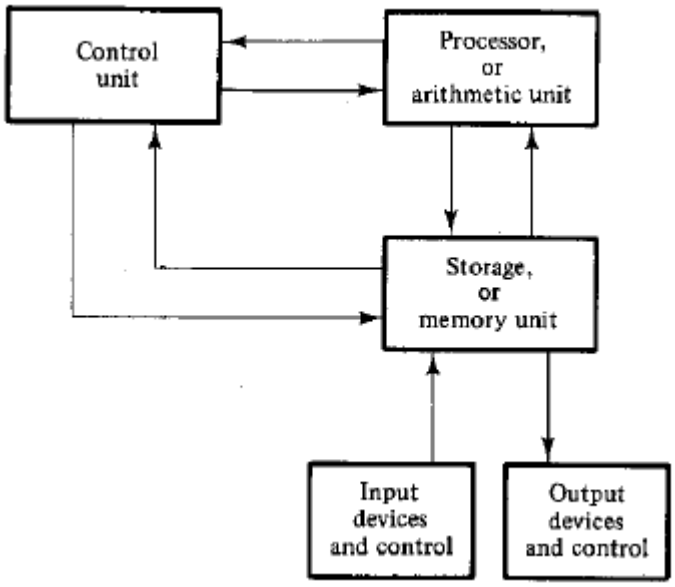


Figure 2.1: Block diagram of a digital computer, adapted from *Digital Logic and Computer Design* by M. Morris Mano [24]

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 [4]

This section concerns itself with the detailed analysis of the 8-bit computer architecture implemented by Ben Eater in his YouTube tutorial series [4]. 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[4]. 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[2] 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.

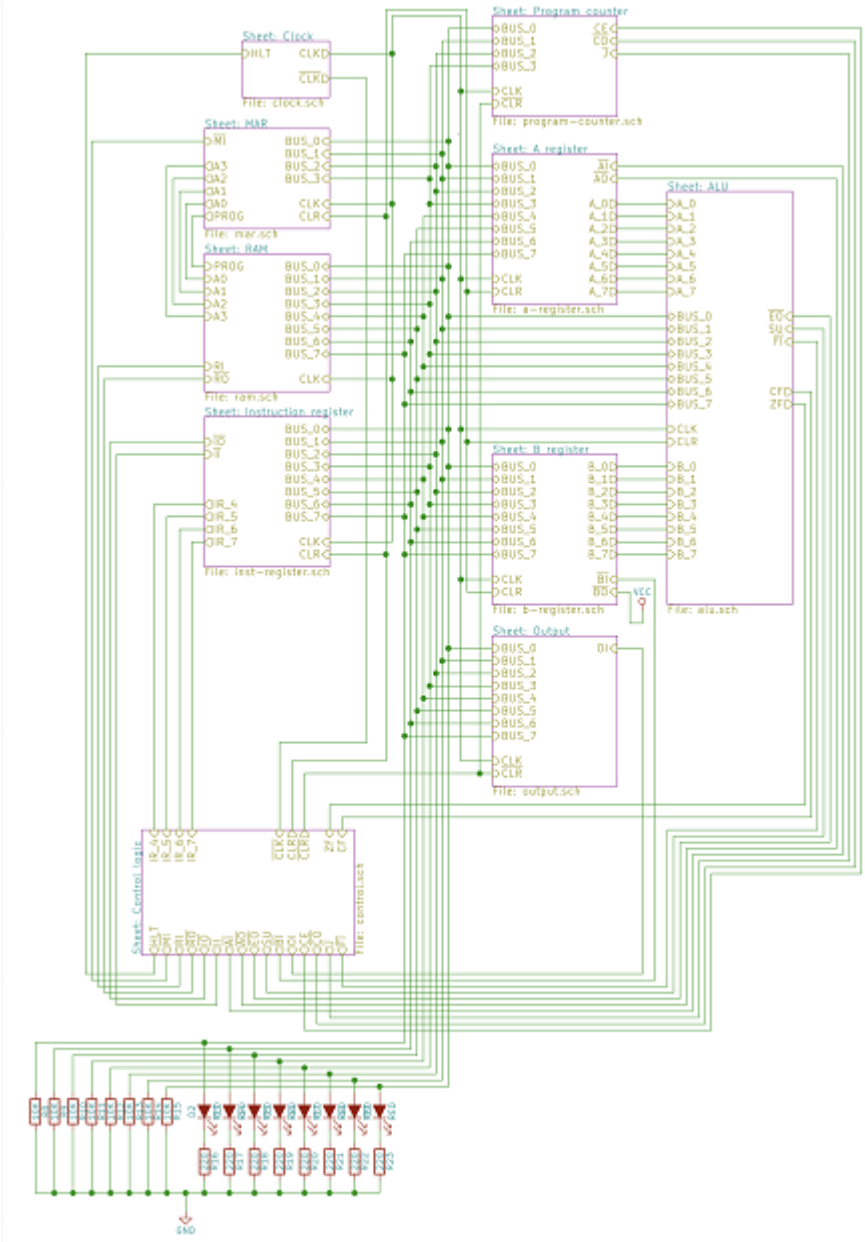


Figure 2.2: Block diagram of the 8-bit computer built by Ben Eater [3]

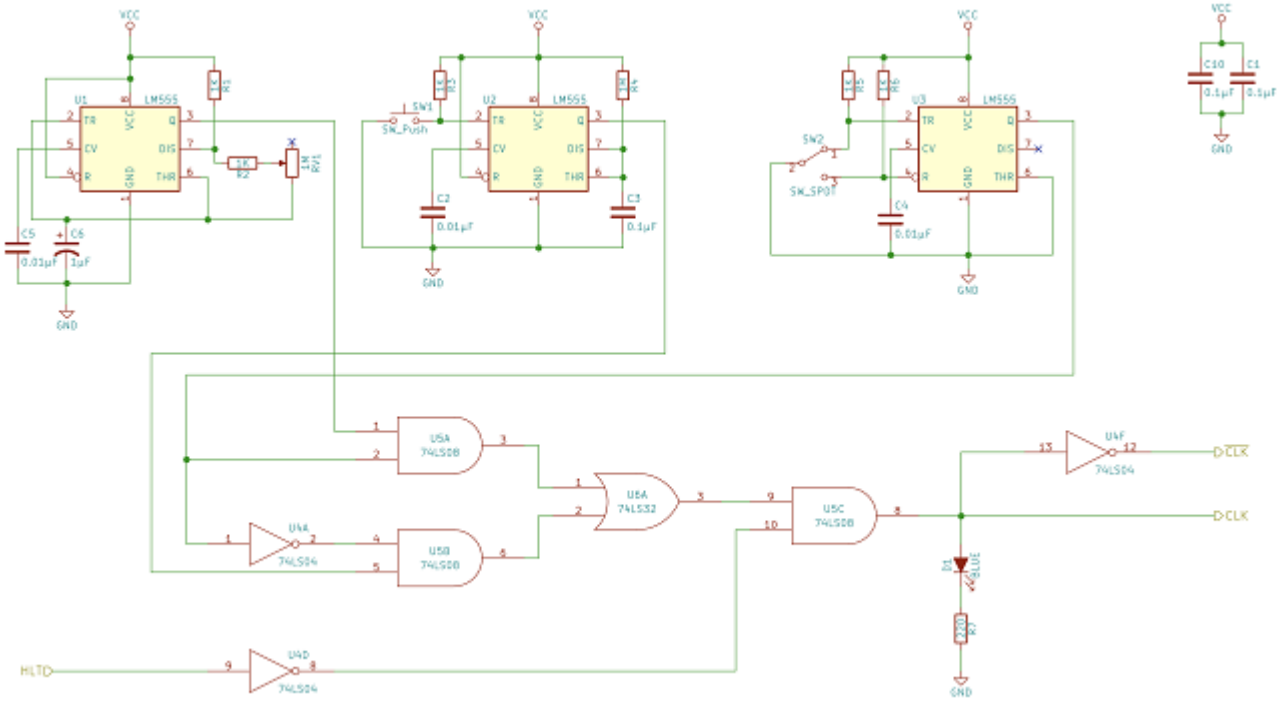


Figure 2.3: Schematic of the clock module in Bean Eater's 8-bit computer [3]

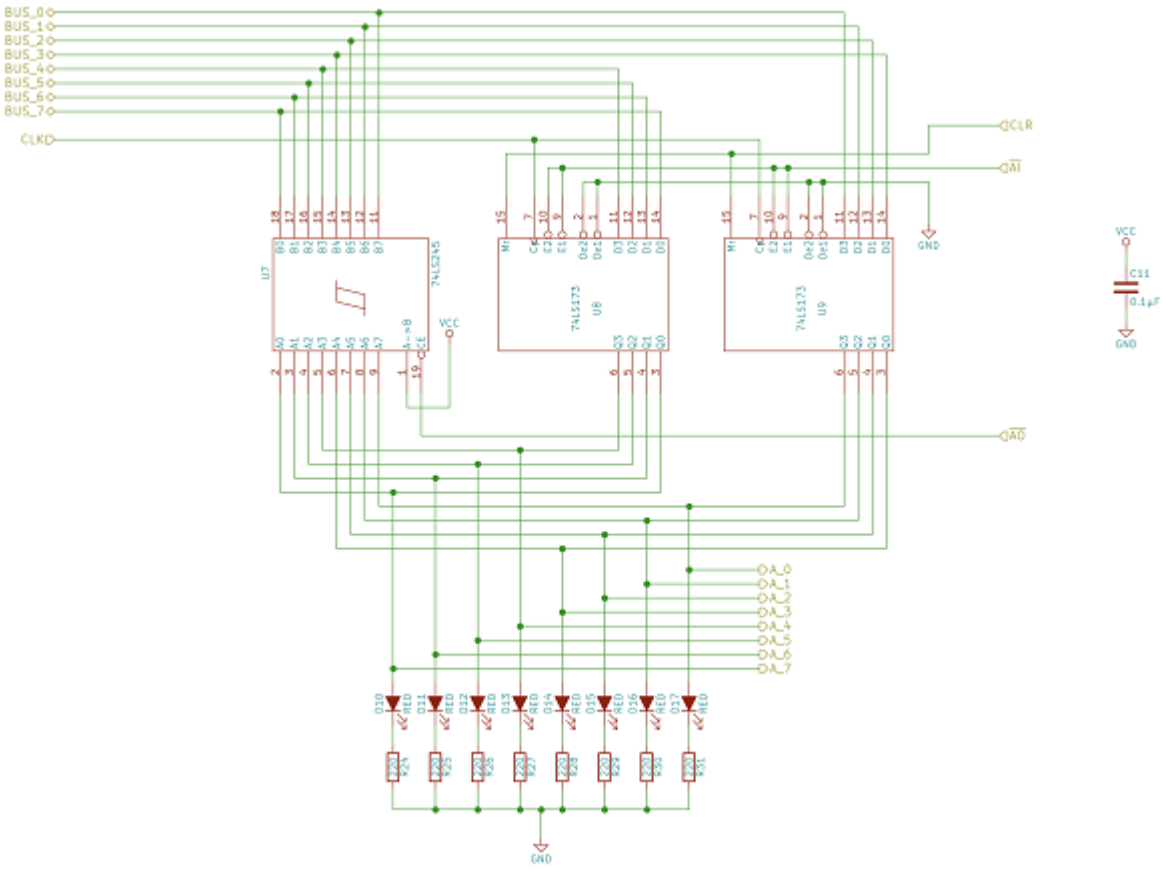


Figure 2.4: Schematic of the A register in Bean Eater's 8-bit computer [3]

Chapter 3

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 [4], but it will also include parts of SAP-2 and SAP-3 [23]. 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 [23] and built by Bean Eater [4]. 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:

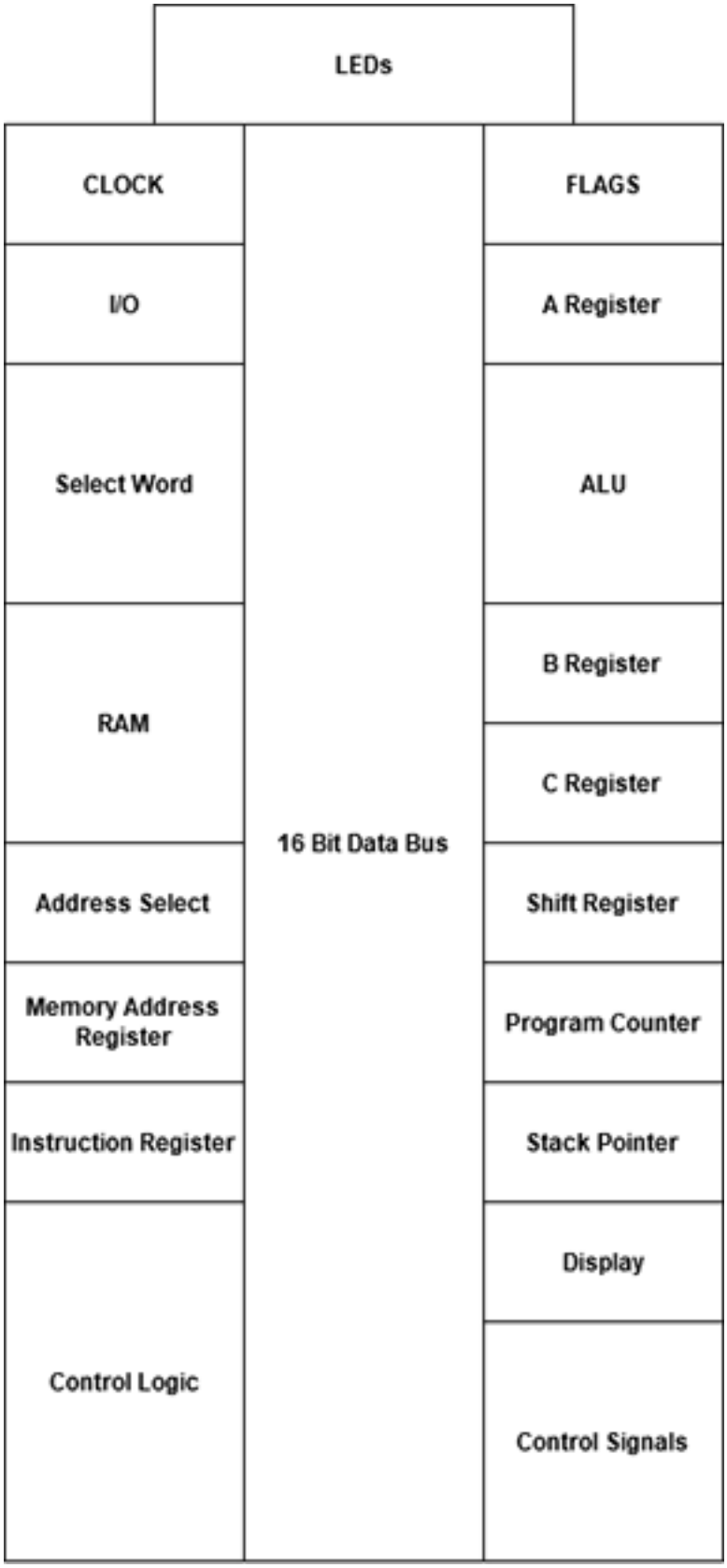


Figure 3.1: High Level Module Overview and Layout of the 16-bit Breadboard Computer

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) **??**
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 **??**
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 **??**.
$InputControlSignals : CLK, AI, \overline{AO}, \overline{RST}$ $DirectConnectionto : ALU$

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* **??**. $InputControlSignals : CLK, BI, \overline{BO}, \overline{RST}$
$DirectConnectionto : ALU$

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 **??**: 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* **??** 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* [10] 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* [10]. 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 3.3.18 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 [10]. 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 [1]. 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* [19] 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 [27]. This particualr logic family was chosen because it is the most popular and widley spread and adopted logic family in the world [26]. The second souce of chips, which served as a quick reference, was Eater's part listing for his 8-bit breadboard computer build [5]. 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* [8] and *Farnell UK* [9].

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 [15] 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.

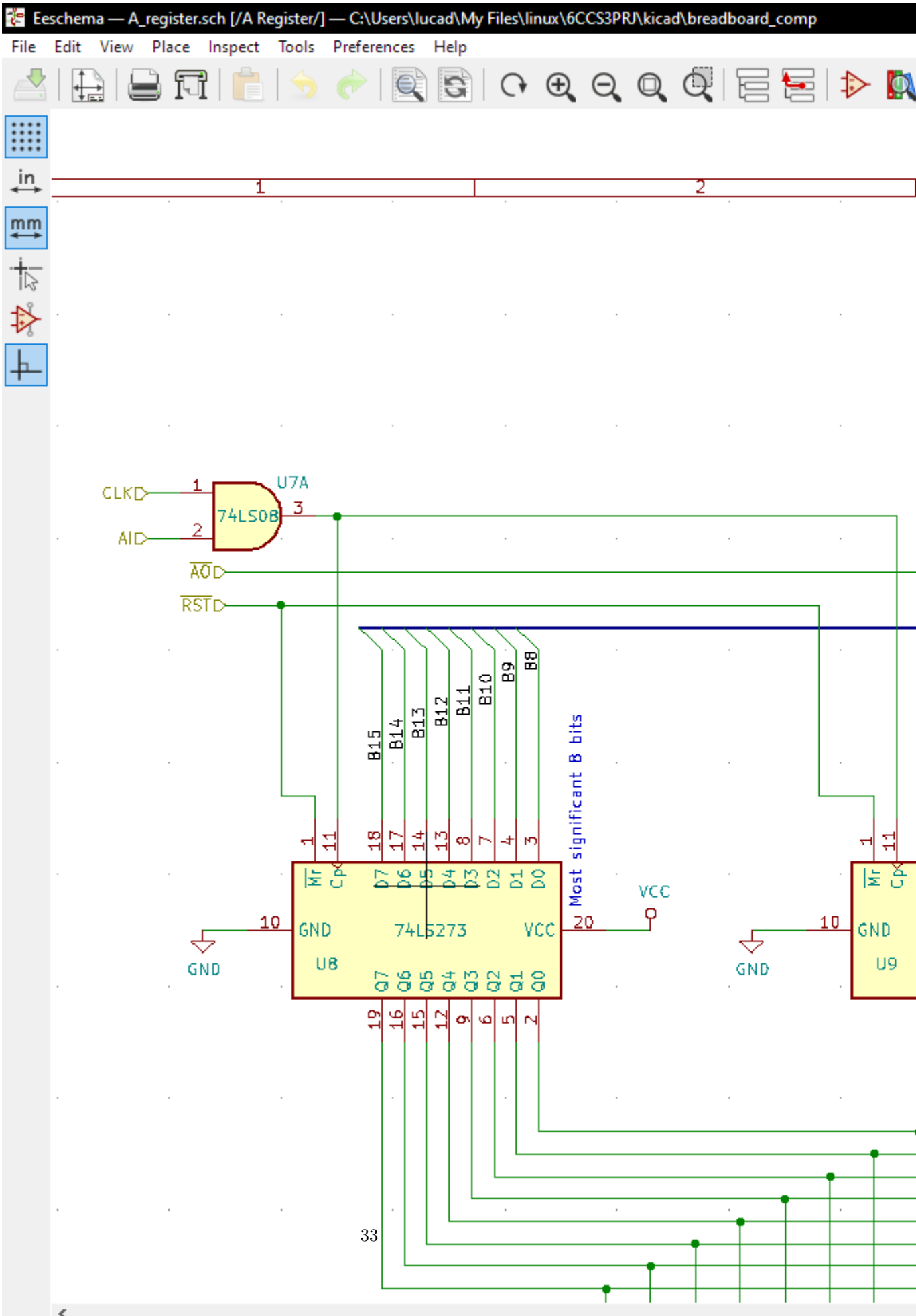
3.5.5 Schematic Creation

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* [22]. 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 Electornic Design Tools with KiCad/Eeschema

The main tool used to to design schematics for the 16-bit breadboard computer is *KiCad* [22]. 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* [21].

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.


Eeschema — A_register.sch [/A Register/] — C:\Users\lucad\My Files\linux\6CCS3PRJ\kicad\breadboard_comp
File Edit View Place Inspect Tools Preferences Help
1
2
CLK
AID
AOD
RST
U7A
74LS08
B15 B14 B13 B12 B11 B10 B9 B8
Most significant 8 bits
VCC
GND
74LS273
U8
U9
Mr
CP
D7 D6 D5 D4 D3 D2 D1 D0
Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0

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.

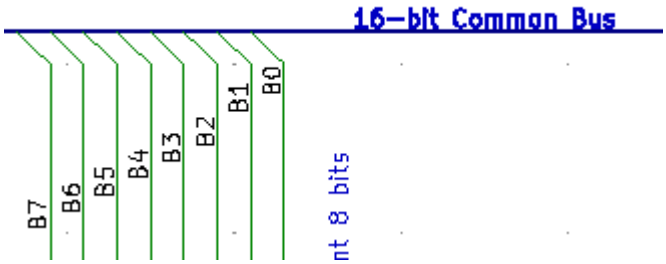


Figure 3.3: Labeled Bus Connections in Eeschema

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.

Sympbol 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.

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.

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.

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:

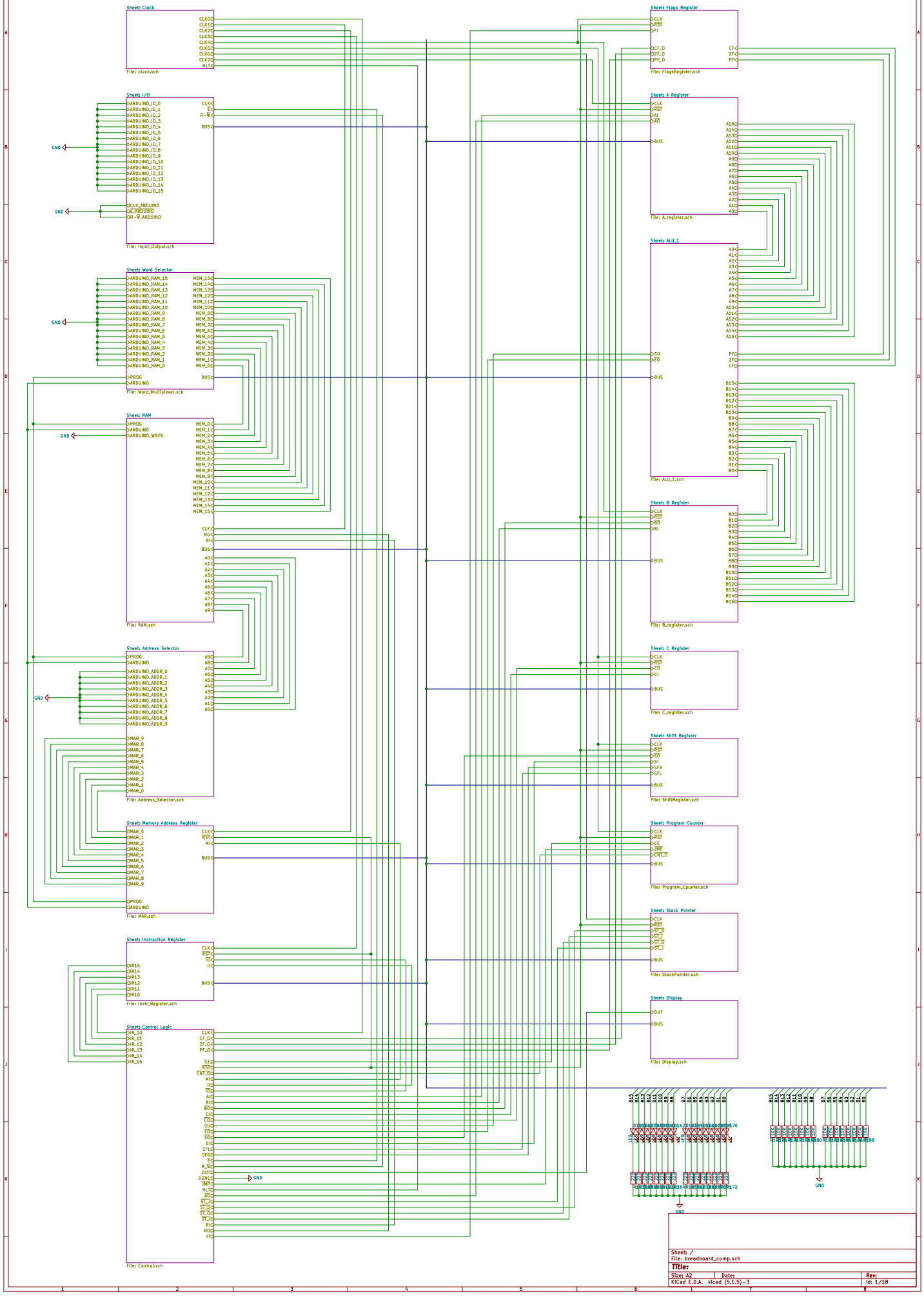
1. Make an adequate chip selection
2. Create a new hierarchial sheet and enter it
3. Use the Symbol Editor to create chip symbols (in case the chips don't exist in the Symbol Library)
4. Place all needed chips in the schematic

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 *Control Signals* module 3.3.20 can also be found.


Sheet: Clock
File: clock.sch
Sheet: I/O
File: Input_Output.sch
Sheet: Word Selector
File: Word_MultSplexer.sch
Sheet: RAM
File: RAM.sch
Sheet: Address Selector
File: Address_Selector.sch
Sheet: Memory Address Register
File: MAR.sch
Sheet: Instruction Register
File: Instr_Register.sch
Sheet: Control Logic
File: Control.sch
Sheet: Flags Register
File: FlagsRegister.sch
Sheet: A Register
File: A_register.sch
Sheet: ALU_1
File: ALU_1.sch
Sheet: B Register
File: B_register.sch
Sheet: C Register
File: C_register.sch
Sheet: Shift Register
File: ShiftRegister.sch
Sheet: Program Counter
File: Program_Counter.sch
Sheet: Stack Pointer
File: StackPointer.sch
Sheet: Display
File: Display.sch
GND
Sheet: /
File: breadboard_comp.sch
Title:
Size: A2
Date:
Rev:
KiCad E.D.A. kicad (5.1.5)-3
Id: 1/18

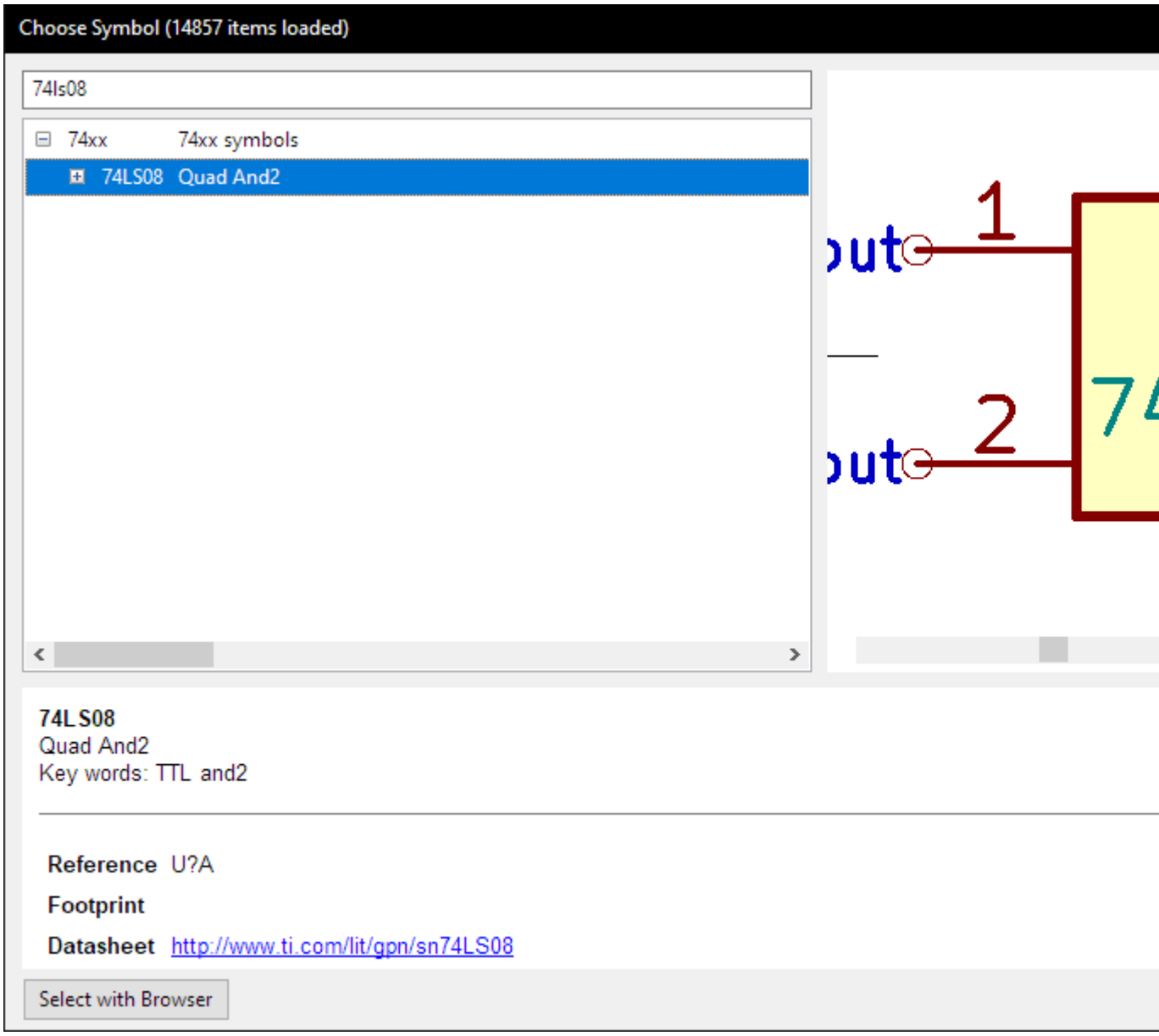


Figure 3.4: Component Database in Eeschema

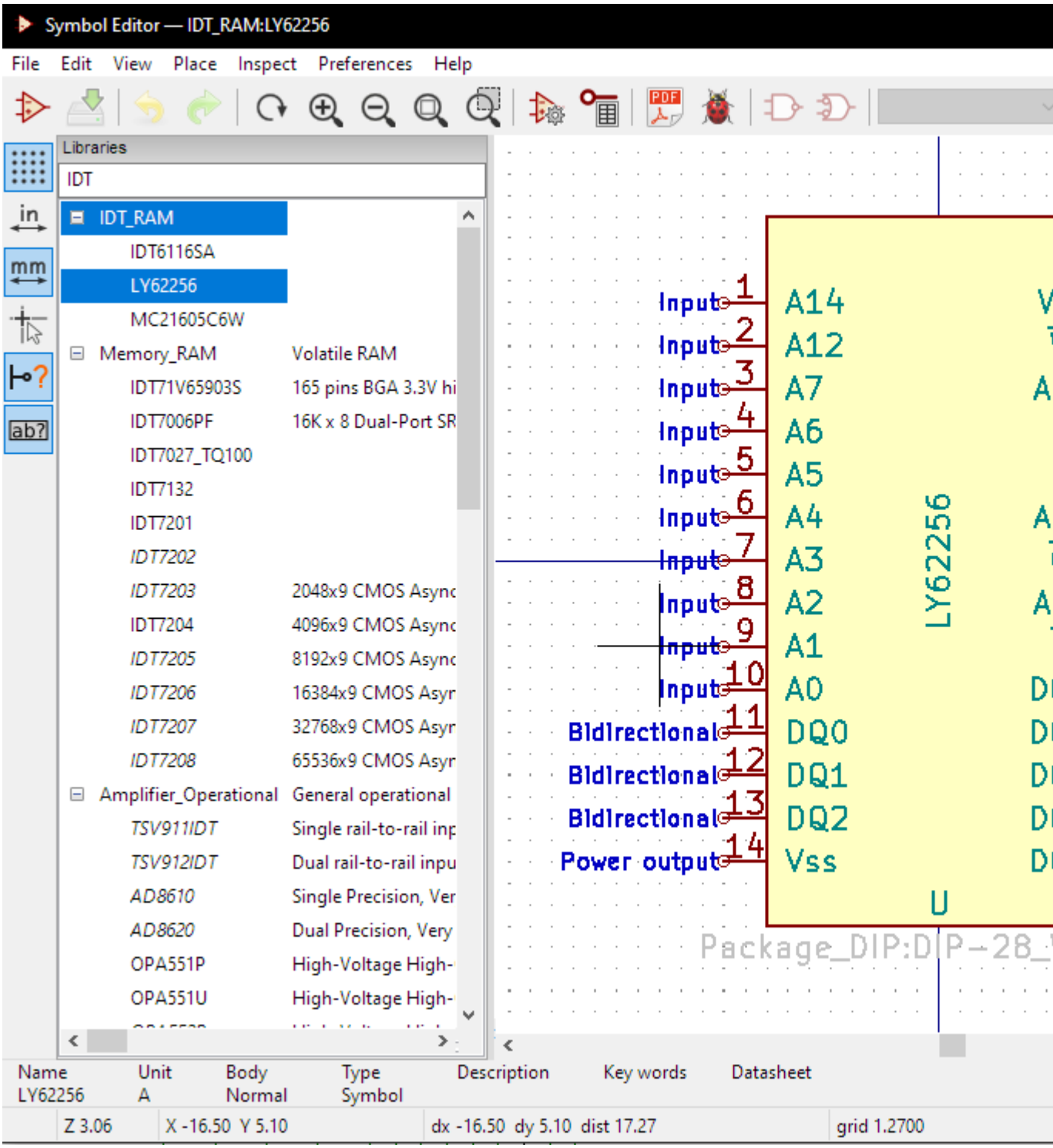


Figure 3.5: Symbol editor in Eeschema

Figure 3.6: The Eeschema Electrical Rules Checker

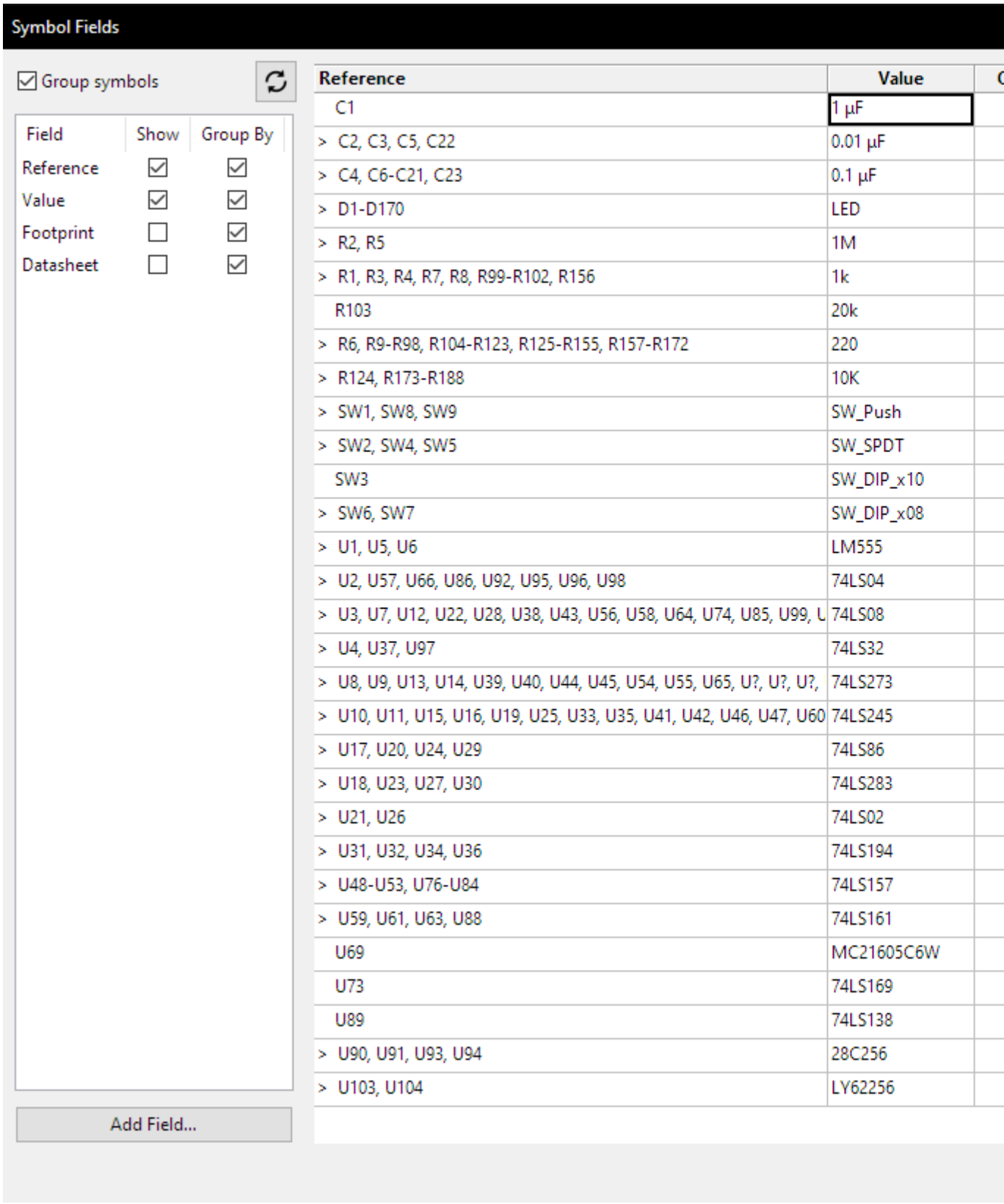
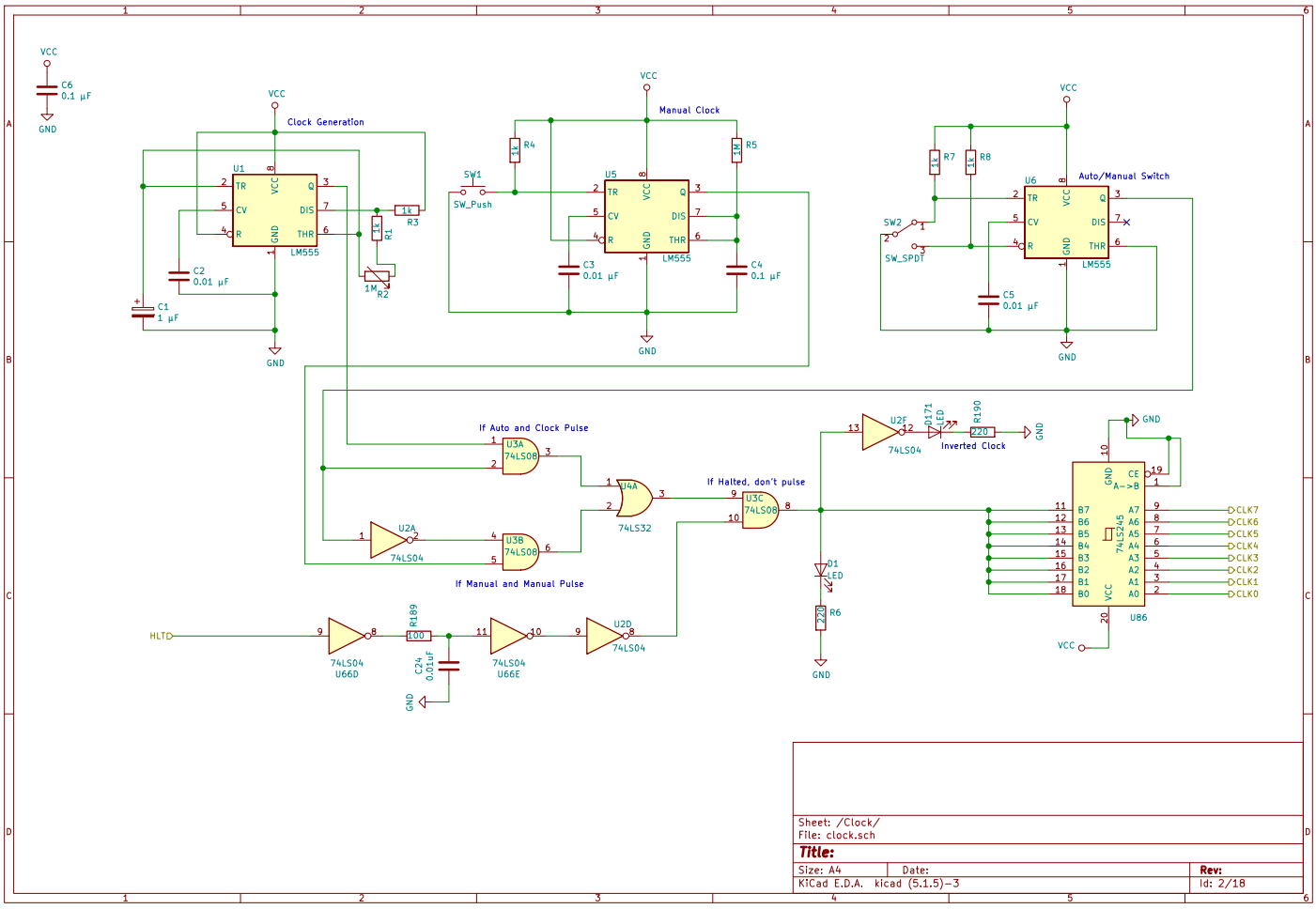


Figure 3.7: Simple Bill of Materials generated by Eeschema

Clock 3.3.3 The clock module is very similar to the clock designed and built by Ben Eater [7]. It uses *555 timers* [13] to generate a square wave which acts as the clock for the system. The *555* is presented in all three most commonly used configurations:

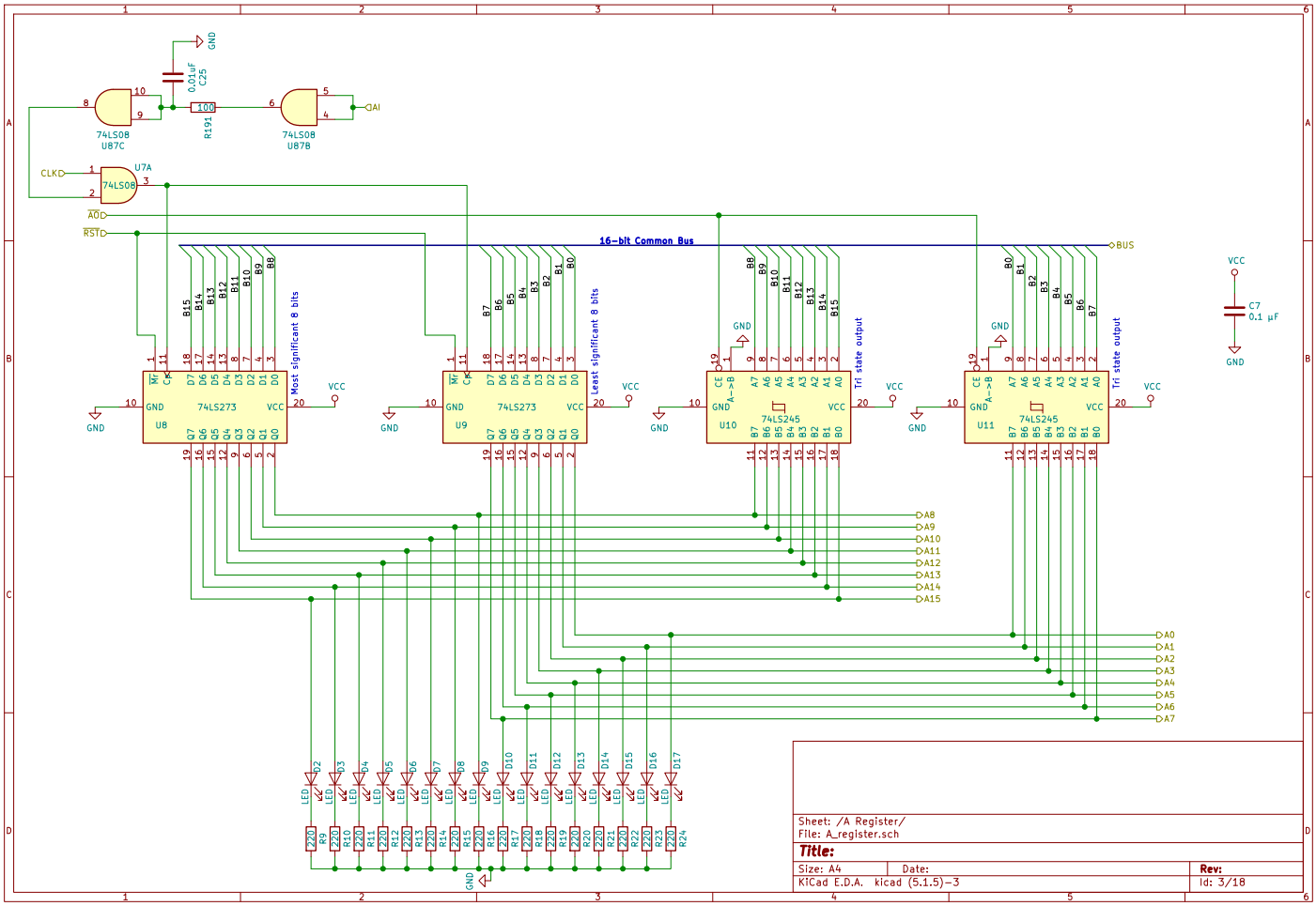
- *Astable:* it oscillates between to states (This 555 generates the main clock)
- *Monostable:* it always „settles down“ to one state after a short period of time (used for the manual push-button clock mode)
- *Bistable:* it has two states between which it can toggle (used for the toggle switch which 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* [14] used to amplify and isolate clock signals going to each module.

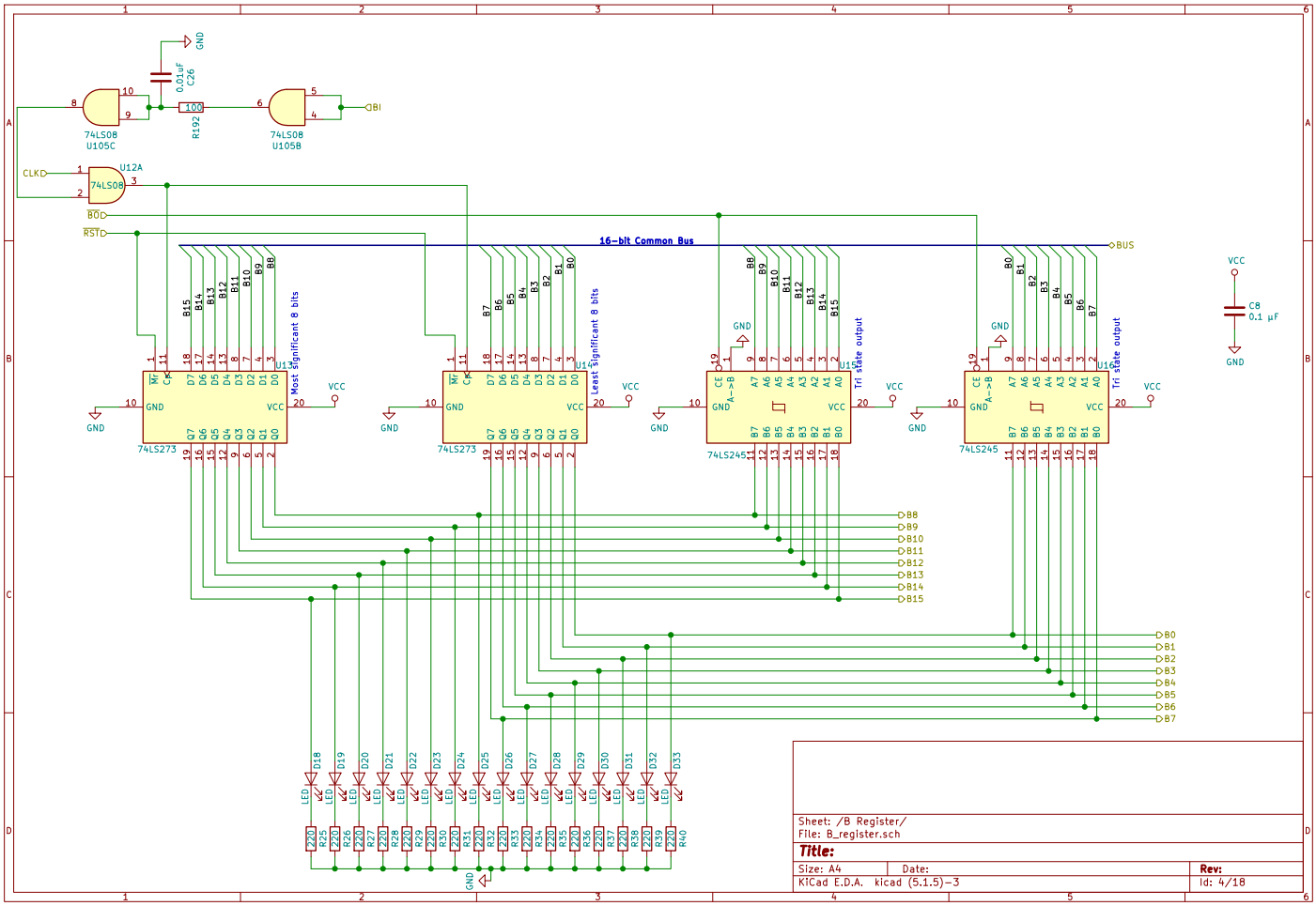


Sheet: /Clock/		
File: clock.sch		
Title:		
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad (5.1.5)-3		Id: 2/18

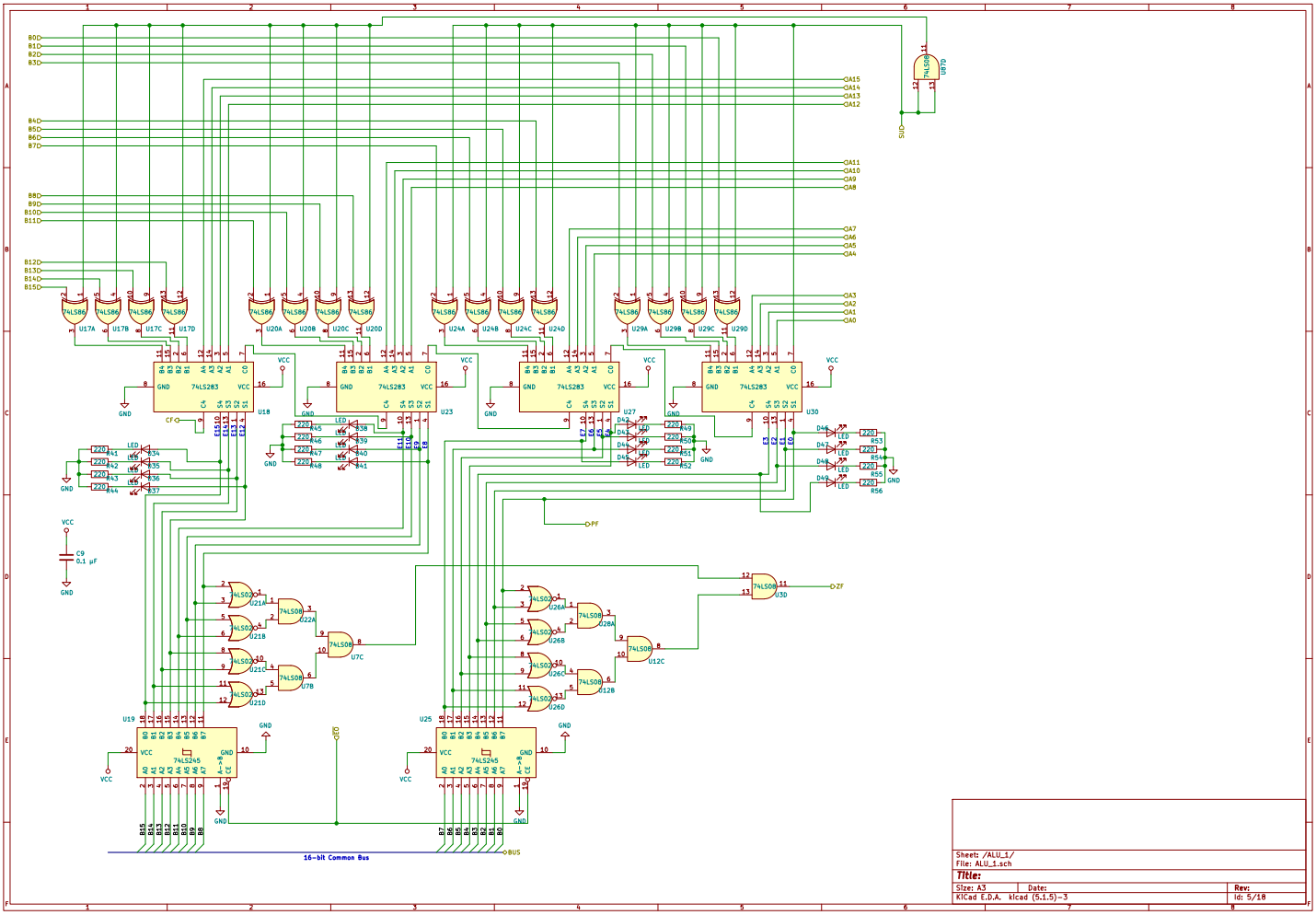
A Register 3.3.4 The A register consists of two *74ls273* [15] 8-bit flip-flop chips and two *74ls245* [14] buffer interfaces to the bus. The buffers are *tri-state* buffers, which means that they can pass through their input, either high or low, or they can disconnect their input from their output by 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 **??**.


74LS08
U87C
74LS08
U87B
R191
100
0.01uF
C25
GND
CLK
RST
A0
AI
U7A
74LS08
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
LED
R220
A8
A9
A10
A11
A12
A13
A14
A15
A0
A1
A2
A3
A4
A5
A6
A7
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
74LS08
U105B
0.01uF
C26
100
R192
GND
BI
CLK
U12A
74LS08
B0
RST
16-bit Common Bus
BUS
VCC
C8
0.1 µF
74LS273
Most significant 8 bits
Least Significant 8 bits
74LS245
Tri state output
B8
B9
B10
B11
B12
B13
B14
B15
B0
B1
B2
B3
B4
B5
B6
B7
LED
220
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 ?? The *ALU* has been designed around the *74LS283* [12] 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* [14] 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* [16] , 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* [18] 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* [17] 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.

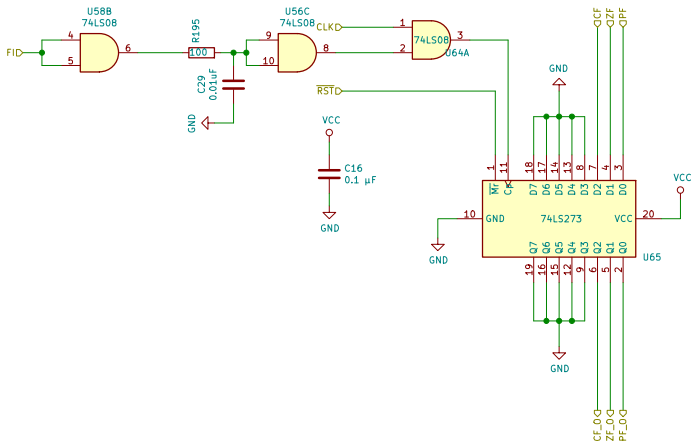


Sheet: /ALU_1/
File: ALU_1.sch

Title:

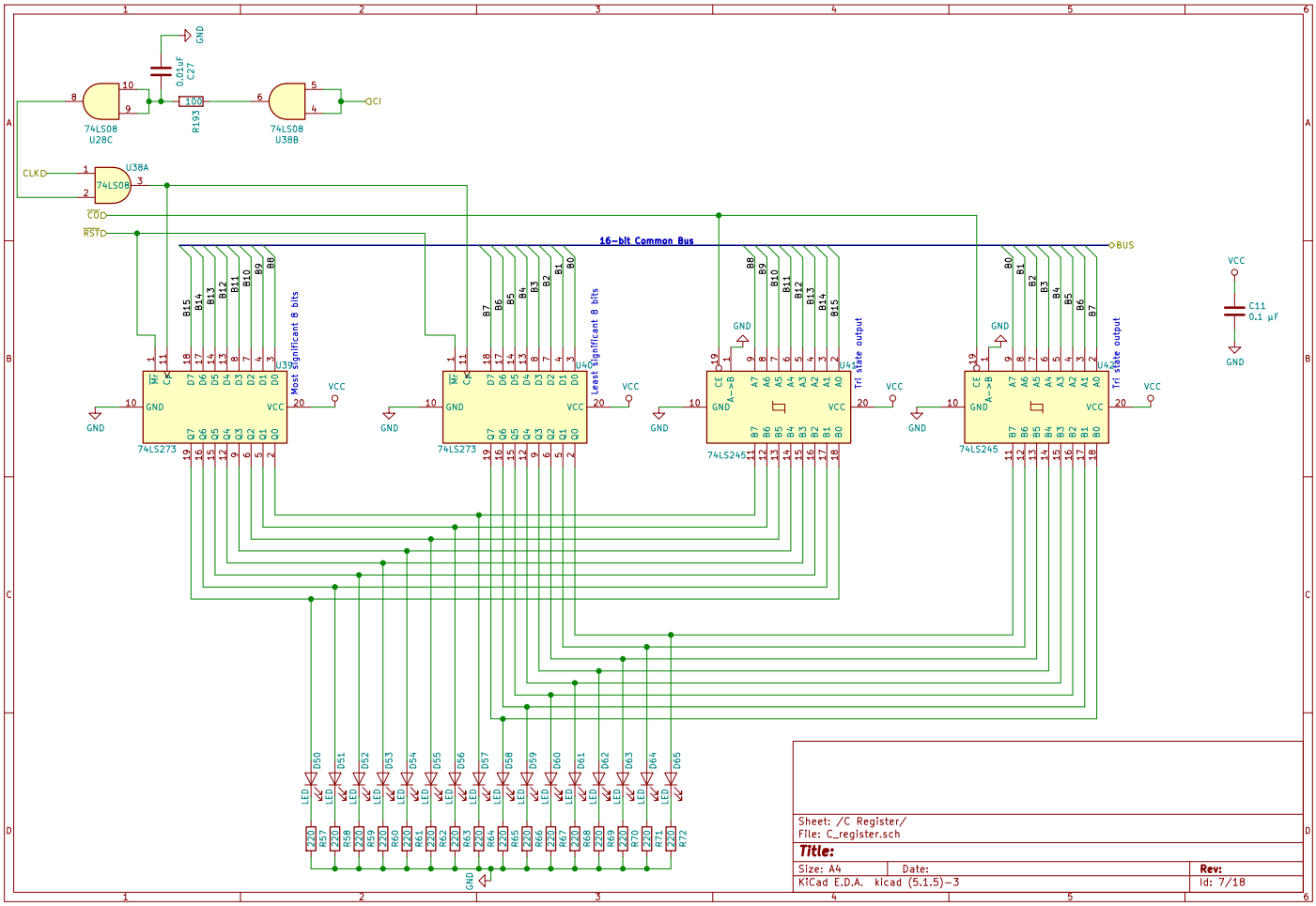
Size: A3	Date:	Rev:
KiCad E.D.A. kicad (5.1.5)-3		Id: 5/18

Flags Register 3.3.7 The Flags register consists of just one *74LS273* [15] 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.

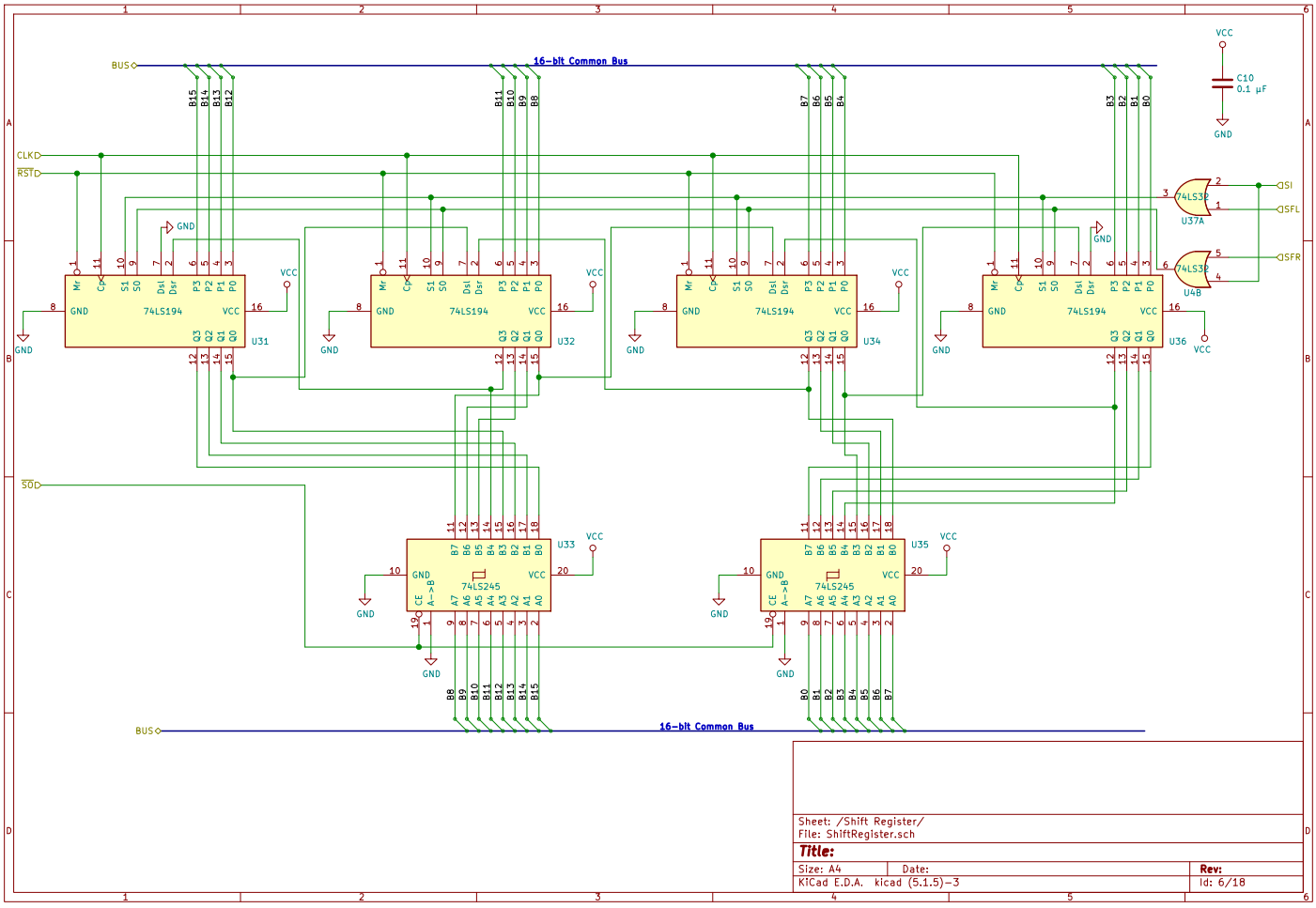


Sheet: /C Register/
File: C_register.sch

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* [11] 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* [14] 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.


BUS
16-bit Common Bus
CLK
RST
SO
SI
SFL
SFR
VCC
GND
C10
0.1 µF
74LS194
74LS245
74LS32
U31
U32
U33
U34
U35
U36
U37A
U4B
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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] 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.

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

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

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

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

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

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

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