6CCS3PRJ Background Specification

Progress Report
Breadboard Computer Architecture

Final Project Report

Author: Luca-Dorin Anton

Supervisor: Christian Urban

Student ID: 1710700

April 17, 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.

<div align="right">

Luca-Dorin Anton

April 17, 2020

</div>

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.

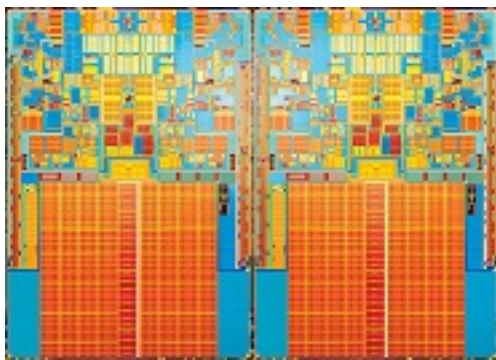Figure 1.1: Intel quad-core 45nm CPU die

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 [6]. 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 [4]. 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. [9]

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!*" [3]. 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* [7].
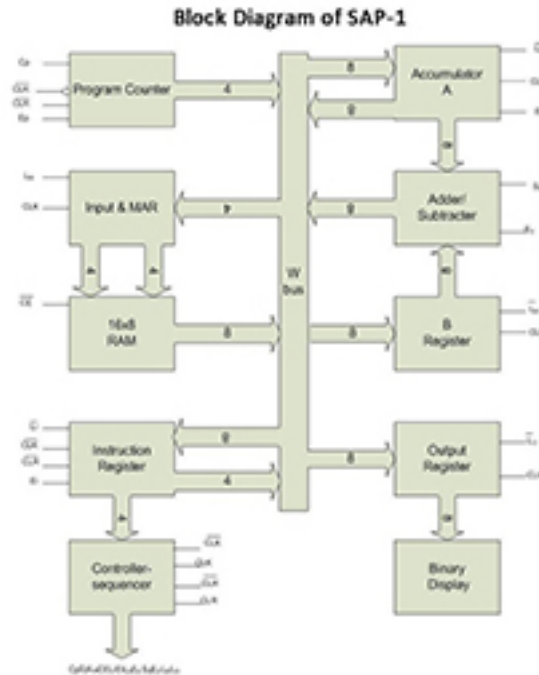

Figure 1.2: Block Diagram of the SAP-1 architecture

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 [3]. 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.
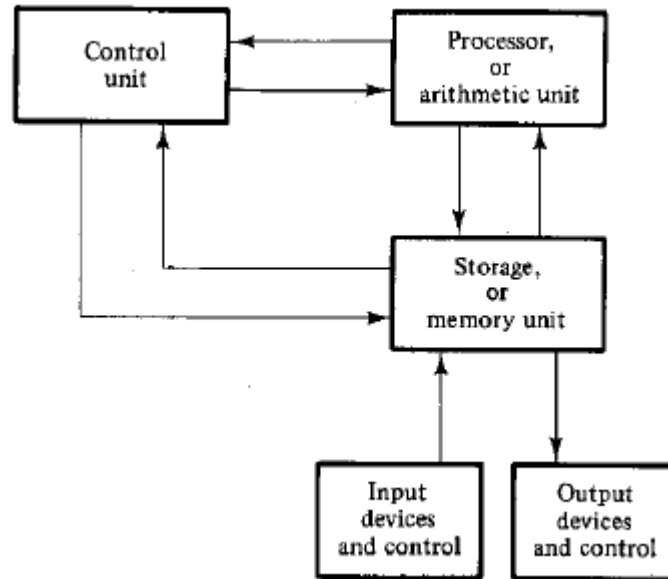
Figure 2.1: Block diagram of a digital computer, adapted from *Digital Logic and Computer Design* by M. Morris Mano [8]

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 [3]

This section concerns itself with the detailed analysis of the 8-bit computer architecture implemented by Ben Eater in his YouTube tutorial series [3]. 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[3]. 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[1] 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.
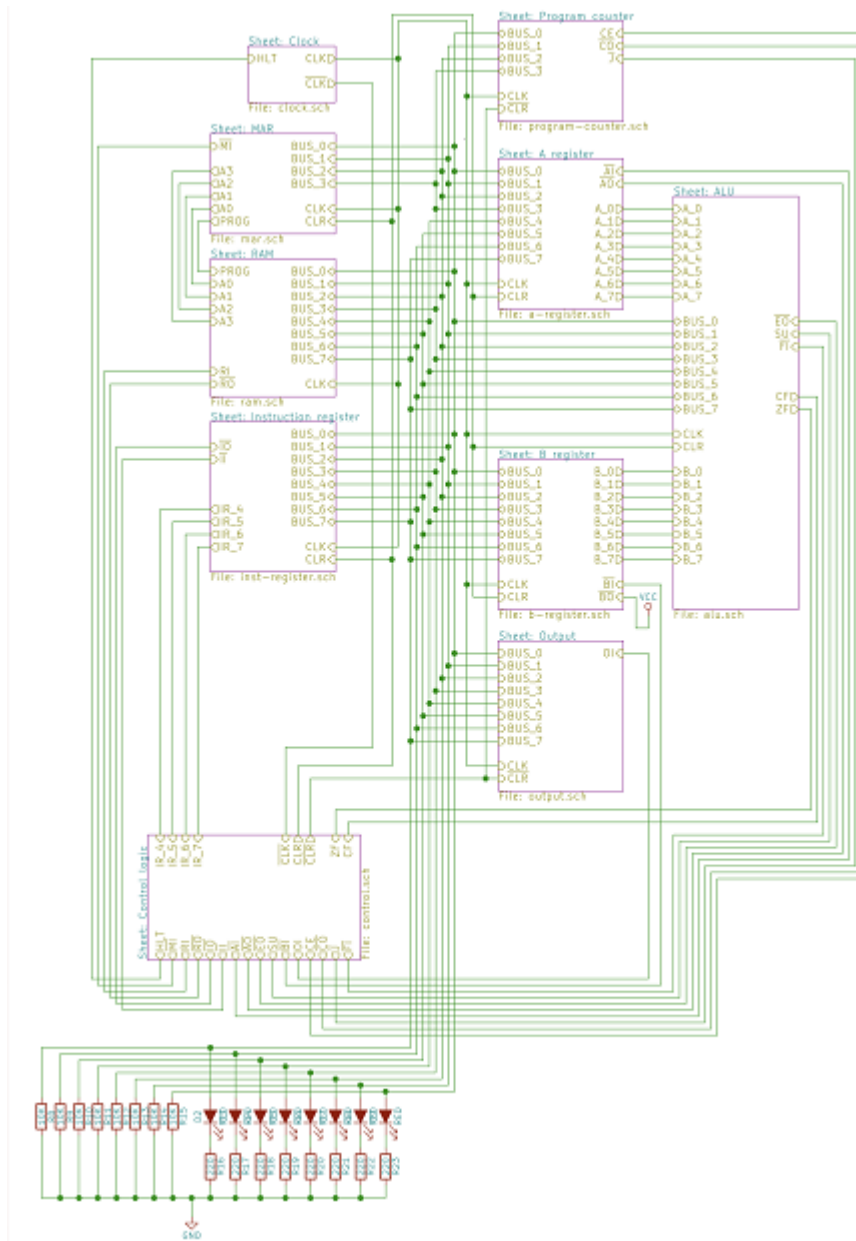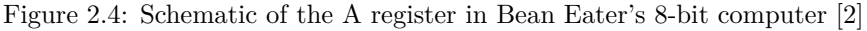
Figure 2.2: Block diagram of the 8-bit computer built by Ben Eater [2]

Figure 2.3: Schematic of the clock module in Bean Eater's 8-bit computer [2]

Figure 2.4: Schematic of the A register in Bean Eater's 8-bit computer [2]

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 [3], but it will also include parts of SAP-2 and SAP-3 [7]. 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 [7] and built by Bean Eater [3]. 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 function-
ality, 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
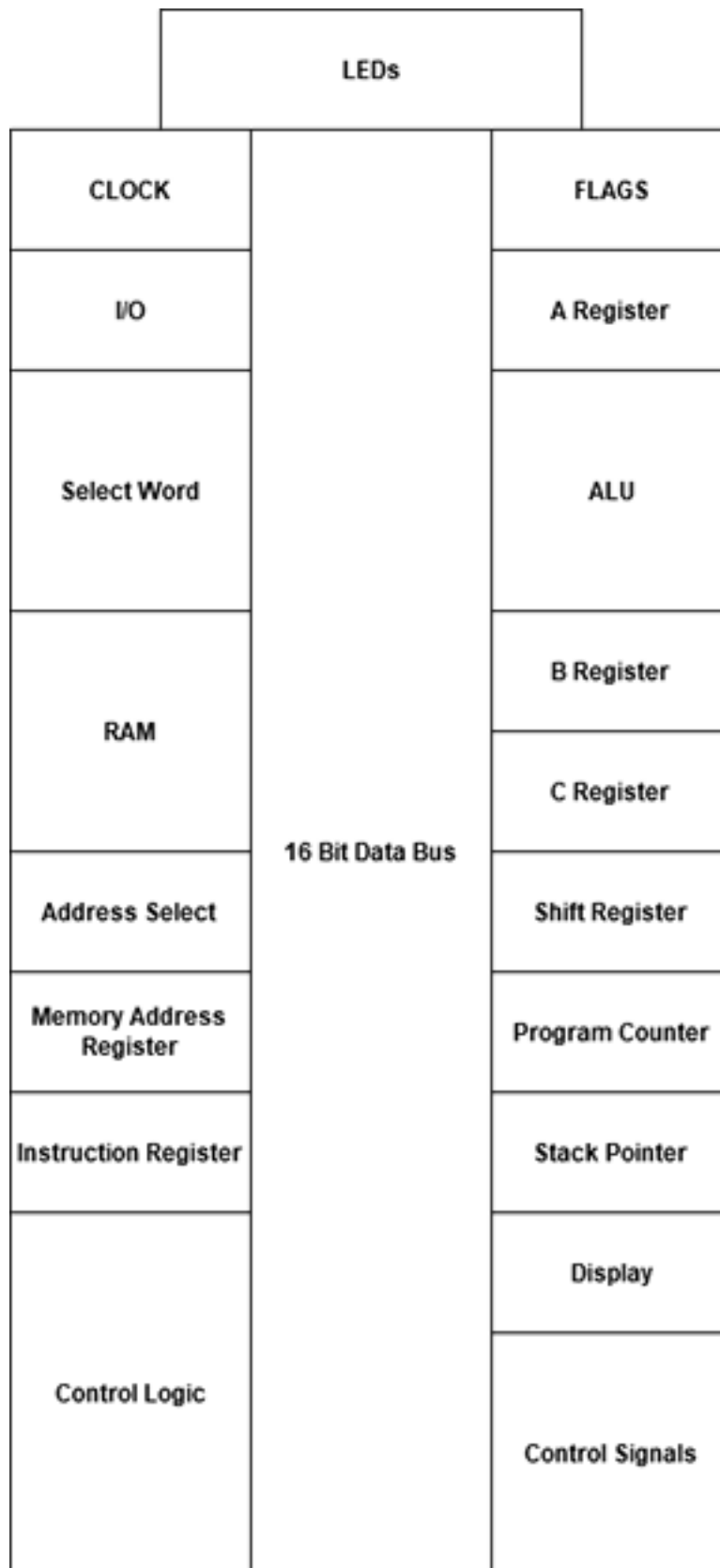following modules and have the following physical layout:

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* [5] controls the writes to memory using a signal called $ARDUINO_W RITE$.

$InputControlSignals : CLK, RI, RO, PROG, ARDUINO, ARDUINO_W RITE$

$DirectConnectionto : WordSelector, AddressSelector, ArduinoMega$

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* [5]. 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 [5]. 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.

References

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

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

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

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

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

[6] 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.

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

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

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