

B4

The comprehensive,
modern, experimentation-
based course unlocks the
fascinating world of Digital
Technologies.

Computer
Processor
Kit

0010
0000
0010
0010



DIGITAL
TECHNOLOGIES
INSTITUTE

“What I was proud of was that I used very few parts to build a computer that could actually speak words on a screen and type words on a keyboard and run a programming language that could play games. And I did all this myself.”

(Steve Wozniak)

Table of Contents

| | |
|---|-----------|
| Included Parts | 7 |
| Hello Parents and Teachers | 8 |
| Hello Students | 8 |
| B4's Parts | 8 |
| <i>Core Modules</i> | 9 |
| <i>Helper Modules</i> | 12 |
| <i>Wires and Connectors</i> | 13 |
| <i>A Word about Power</i> | 14 |
| <i>Please look after me</i> | 14 |
| Exploration through Experimentation | 15 |
| Experiments | 16 |
| <i>Overview</i> | 16 |
| <i>Experiment 1: One Small Step</i> | 18 |
| <i>Experiment 2: Adding Two Numbers</i> | 22 |
| <i>Experiment 3: What about Subtraction?</i> | 24 |
| <i>Experiment 4: Short Term Memory</i> | 27 |
| <i>Experiment 5: Long Term Memory</i> | 33 |
| <i>Experiment 6: Giving Direction to Data</i> | 37 |
| <i>Experiment 7a: Let's Build a Manual Computer, Part 1</i> | 39 |
| <i>Experiment 7b: Let's Build a Manual Computer, Part 2</i> | 48 |
| <i>Experiment 8: Let's Make a Real Computer</i> | 53 |
| <i>B4 Design</i> | 53 |
| <i>Designing the Program</i> | 55 |
| <i>B4 Assembly</i> | 56 |
| <i>Experiment 9: Programming the B4</i> | 62 |
| 1. <i>Programming the Program RAM</i> | 62 |
| 2. <i>Programming the Data RAM</i> | 64 |
| 3. <i>Executing the Program</i> | 65 |

| | |
|--|------------|
| Experiment 9a: Adding three numbers | 67 |
| <i>Designing the Program.....</i> | <i>67</i> |
| <i>Running the Program.....</i> | <i>68</i> |
| Experiment 10: B4 Learns Subtraction | 70 |
| <i>Designing the Program.....</i> | <i>70</i> |
| <i>Running the Program.....</i> | <i>71</i> |
| Experiment 11: Automatic Programming..... | 73 |
| <i>Step 1 Installing the Automatic Programmer.....</i> | <i>73</i> |
| <i>Step 2: Modules and their Wiring.....</i> | <i>74</i> |
| <i>Step 3: Installing and Configuring the Arduino IDE.....</i> | <i>78</i> |
| <i>Step 4: Installing the B4 Arduino Library.....</i> | <i>78</i> |
| Experiment 12: Program Language Design..... | 83 |
| <i>Simplifying our Program.....</i> | <i>86</i> |
| <i>Summary</i> | <i>86</i> |
| Experiment 13: On the Role of Timing | 87 |
| Experiment 14: So, how does a Computer work ... actually? | 88 |
| <i>Logic and Boolean Logic.....</i> | <i>88</i> |
| <i>A Logical Adding Machine.....</i> | <i>90</i> |
| <i>A Logical Memory Machine</i> | <i>93</i> |
| <i>Engineering.....</i> | <i>97</i> |
| <i>Summary</i> | <i>101</i> |
| Experiment 15: Cyber Security | 102 |
| <i>Software: Understanding the B4's Arduino Library.....</i> | <i>102</i> |
| <i>Hardware: Hacking deeper yet by understanding the Automatic Programmer ..</i> | <i>107</i> |
| <i>Hack 1: Randomly incrementing the Program Counter.....</i> | <i>110</i> |
| <i>Hack 2: Randomly changing the Data RAM</i> | <i>111</i> |
| Further Reading | 113 |
| Troubleshooting..... | 113 |
| Appendix A: Programming Table Template..... | 114 |

| | |
|--|------------|
| Appendix B: Fun Algorithms | 115 |
| Appendix C: Solutions | 116 |
| Appendix D: Extension Kits | 124 |
| Appendix E: Quick Reference Guide | 125 |



WARNING:
CHOKING HAZARD - Small Parts
Not for children under 3 years.
PHOTOSENSITIVE EPILEPSY -
Some of the experiments produce
light flashes that can potentially
trigger seizures in people with
photosensitive epilepsy

Safety instructions

The B4 operates on 5 Volts and only draws a few milliamperes. Nevertheless, it is an electrical device and should be handled as such. We recommend to treat it with care, and to keep it on a non-conductive, dry and level surface. Do not scratch the surface of the printed circuit boards with sharp or metallic instruments, as this might damage the wires.

Acknowledgements

We would like to thank Charles Petzold, the author of 'Code: The Hidden Language of Computer Hardware and Software', published in 1999. His book has both inspired and guided the design of the B4. We recommend it as additional reading material for students.

We would further like thank Henrik Maier from proconX for his guidance and feedback on the electrical engineering design, fabrication and component selection, which has been invaluable to transforming the B4 from a breadboard prototype to a robust design that can be used in the classroom.

Special thanks to Dr. Hayden White for his support and input which have been invaluable to get the B4 off the ground. His regular feedback on the development of the B4 has influenced many of the design decisions.

A big thank you is owed to Martin Levins, Katie Woolston and Michael Schulz for their contributions to this handbook. David Schulz has contributed code that drives the seven segment LEDs of the Program Counter and the Decimal Display. He originally developed this for his Young ICT Explorers project, The Tardis, in 2016. We'd further like to thank the Arduino community: Two of the B4's modules deploy an Atmega processor which run Arduino programs. Keep up the great work!

Mrs. Sharon Singh and her year 8 students (8N and 8W) at St. John's Anglican College in Forest Lake, QLD, have provided very valuable feedback on the B4 and this handbook, which has lead to many improvements. Thank you!

The logic diagrams in this handbook have been designed using the Logicly program. We think it is a great tool to quickly draw and test Boolean logic problems.

Dr. Karsten Schulz, CEO, The Digital Technologies Institute.

Included Parts

1x 2-Line-to-1-Line Selector
1x Adder
1x Automatic Programmer Arduino Shield
1x Data Random Access Memory, 16x4bit
1x Decimal Display
1x Inverter
1x Latch
2x Variable
1x Program Counter
1x Program Random Access Memory, 16x4bit

14 x 4 Pin Wires
10 x 2 Pin Wires
11 x 1 Pin Wires
1x USB Cable

1x Printed Student Handbook
1x B4 Arduino Library (available for download at <http://www.digital-technologies.institute/downloads>)

1x Arduino Uno compatible (required for the full function of the Automatic Programmer Arduino Shield)

Power Consumption:
5V, 200mA (average), 1W DC.

This product complies with the Restriction of Hazardous Substances Directive and is lead free.

The illustrations in this handbook can slightly differ from the actual modules. However, the functionality is the same.

This handbook has been made with great care. Should you find errors or have ideas to improve it, please email us at enquiries@digital-technologies.institute.

Designed and manufactured in Australia
(c) Digital Technologies Institute PTY LTD, 2016-21 AD. All rights reserved.

Hello Parents and Teachers

The B4 is an educational computer support students in their learning about digital systems. It has been designed from the ground up to support the new Australian Curriculum: Digital Technologies. The B4 supports the teaching of the knowledge and understanding of digital systems and the representation of data. The B4's design goes back to the time in the 1970's when early digital computers emerged. It follows similar design principles as some of the famous classic computers, such as the Apple I, the Altair 8800, or the Z-80. These principles are still valid today in modern computers, smartphones and tablets. The B4 illustrates these principles and combines them with modern 21st century Arduino technology to let students, parents and teachers explore the magic of making a computer without needing a university degree.

A virtual version of this hardware kit is available at <https://mycomputerbrain.net/php/courses/bk.php>. Teacher and parent accounts are free and student accounts can be purchased at a reasonable price. We recommend to use the physical kit in conjunction with the virtual kit. In the virtual kit, we have slowed down the data and control flow, which helps students to trace the sequence of steps.

Hello Students

Computers were once bigger than our bedrooms. Their parts were big and you could hear and see them working, or computing as we say. Modern computers have become very tightly integrated and fit into the pockets of our pants. But this means that their parts have become so small that we can hardly see them with our own eyes. Therefore, the B4 has bigger components that you can easily see. Whereas the speed of modern computers is measured in millions of instructions per second (MIPS) the B4 operates at human speed, thus allowing us to see with our own eyes how data flows between each of the B4's modules.

The B4 is a 4 bit Harvard-architecture-style microprocessor. It can store and process numbers and instructions that are 4 bits long, meaning that it can work with positive numbers from 0 to 15. It has one data storage and one program storage module. Each of them can hold 16 of these 4 bit numbers.

This may not sound like much, because you are probably used to 64 bit computers with gigabytes of memory. But the B4 is not meant to compete with these computers. It is simple enough to teach Digital Technologies fundamentals. Still, you will be surprised what can be done with a 4bit computer.

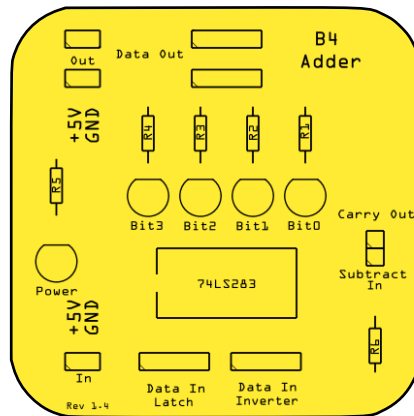
Now let's see what's in the box:

B4's Parts

The B4 consists of seven different modules which represent the most important parts of a Computer's Central Processing Unit (CPU) and Memory, plus four helper modules for programming and data conversion. The B4 can be programmed manually step by step, which is useful to learn coding in detail. For convenience, and to aid with the repetition and expansion of experiments, the B4 also has an Automatic Programmer, which requires an Arduino Uno or compatible.

Generally, each module receives its input through the connectors at the lower end of the module and provides an output through the connectors at the top. All connectors are labeled with **In** or **Out**.

Output



Input

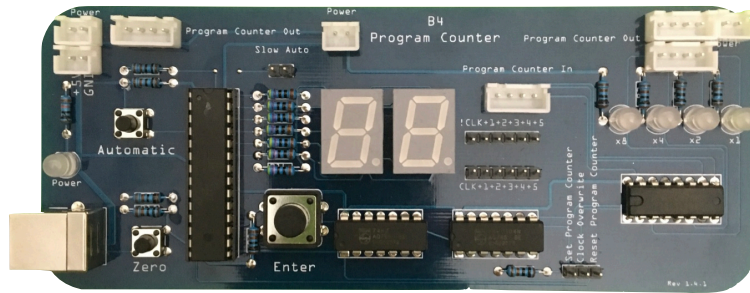
We can further classify the B4 modules into control, arithmetic, memory & storage, programming and miscellaneous. All modules within a category are of the same colour. The control modules are blue and the arithmetic modules yellow. Anything to do with memory & storage is green. The programming modules are red and anything else is black.

| | Function | Color | Modules |
|----------------|------------------|--------|----------------------------------|
| Core Modules | Control | blue | Program Counter, 2-to-1 Selector |
| | Arithmetic | yellow | Adder, Inverter |
| | Memory & Storage | green | Data RAM, Program RAM, Latch |
| Helper Modules | Programming | red | Variables, Automatic Programmer |
| | Miscellaneous | black | Decimal Display |

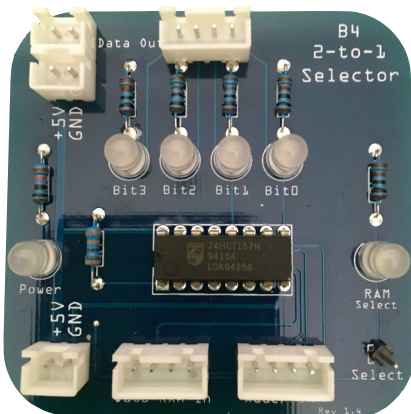
The modules are all labeled. Take them out of the box and find each of the modules as we describe them below. Let's start with the core modules:

Core Modules

The **Program Counter**'s main function is to count from 0 to 15. Every time you press the Enter button, the number on the display will grow by one. When it shows 15 and you press the button, the counter will tick over and start at 0 again. This number is important for the Data and Program RAM modules so that they know which step of the program they should be working on and where the corresponding data is located. Every time you press the

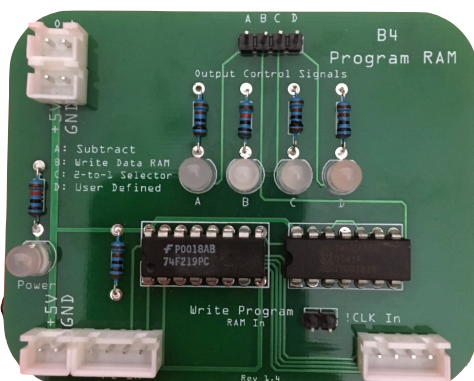
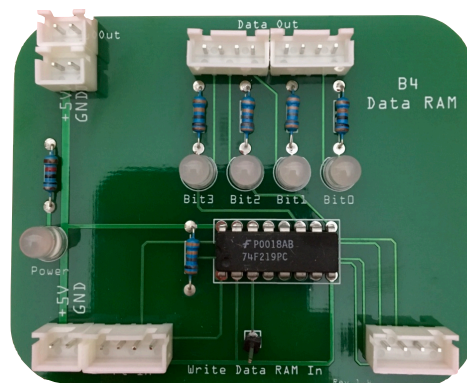


Enter button, the Program Counter will also send a clock (abbreviated as CLK) signal to some of the other modules. We will talk about this later. The program Counter also has a Reset button, which resets the output to 0.



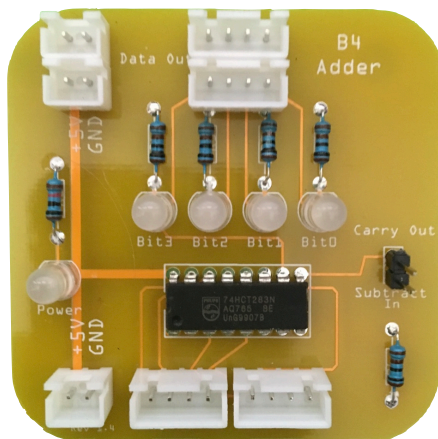
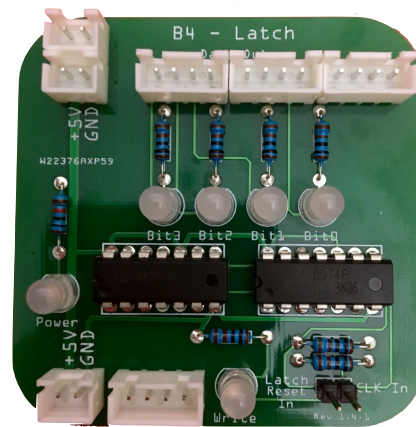
The **2-to-1 Selector** is a switch that can change the data paths in our computer. Precisely, it is used to select data from the Data RAM or from the Adder. This is important when adding two numbers, as we will see later.

The **Data Random Access Memory (RAM)** holds the data that the program is working on. For example, it would hold two numbers that the program will be adding. The Data RAM has room for 16 numbers, which are 4 bit wide, thus representing the decimal numbers of 0 to 15. The Data RAM is a 16 by 4, or 16x4, memory module.



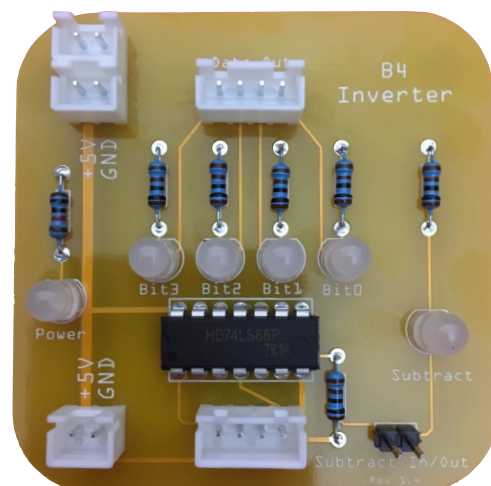
The **Program Random Access Memory (RAM)** holds the program that manipulates the data. For example, it would contain the information so that two numbers from the Data RAM would get subtracted, or that the result of an operation be stored back into the RAM. You will see later that a program is quite different to what you think it is. Like the Data RAM the Program RAM has room for 16 instructions, which are 4 bit wide, thus representing the numbers 0 to 15. Therefore, the Program RAM is also a 16 by 4, or 16x4, memory module.

A **Latch** has the function of short-term memory in a computer. It stores (or 'buffers') some data before that data can be further processed. For example, in the B4, the Latch stores the first number so that a second number can be added to it.

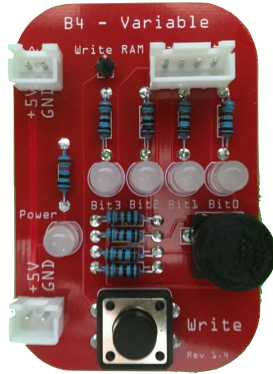


The **Adder** can add exactly two numbers.

The **Inverter's** function is to help the Adder to subtract numbers. In the binary world, subtracting is the same thing as adding the binary complement, and then adding 1 to the result.

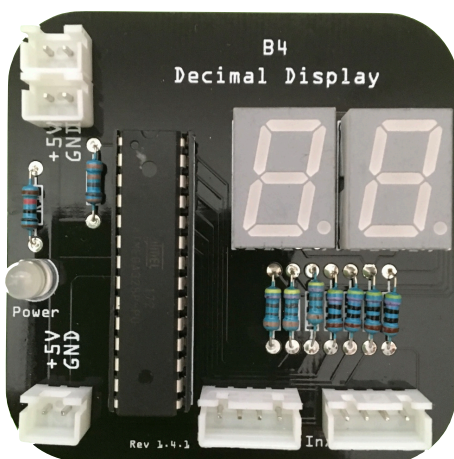
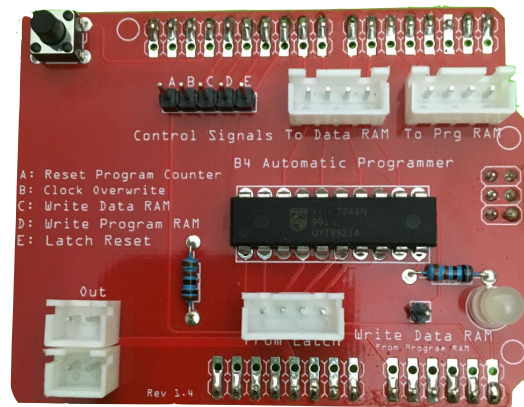


Helper Modules



With the **Variable** we can produce binary data simply by rotating the knob. You can think of it as a variable in a computer program. With its button we store data and program code in the Data and Program RAM modules. The B4 ships with two Variable modules. The knob of the Variable was made on a 3D printer.

The **Automatic Programmer** is the Variable's bigger brother (or sister). The Automatic programmer can be plugged into an Arduino Uno (or compatible). With the Arduino Integrated Development Environment (IDE) we can then write and transfer B4 code from our laptops or PCs. The B4 comes with a handy Arduino library that you can use to write your own B4 programs. We'll talk more about this a little later.



All computers internally work with binary numbers only. However, we humans are more familiar with decimal numbers. As you work with the B4, you will get used to 1's and 0's and you will find it increasingly easy to remember that a 0111 is a decimal 7. The **Decimal Display** is a handy little module that does that binary to decimal conversion for you. You can plug it into any output port of any other module, or insert it between any other two modules.

We now have a basic understanding of the modules of our B4. Don't worry if you haven't understood everything yet. We will revisit each module in more depth during the following experiments.

Wires and Connectors

In order to connect the B4 modules with the computer and with each other, the B4 comes with 4 types of wires. They are:



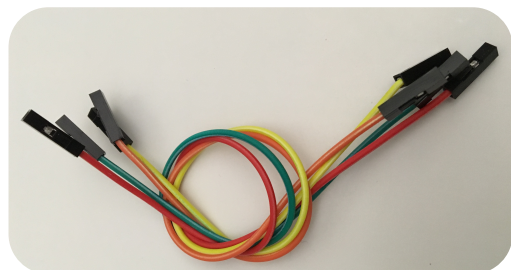
A **USB cable** to provide electricity from a power source to the B4's Program Counter module and from there, to all other modules connected to the Program Counter. You can connect the USB cable to a PC, Laptop, USB Hub, USB battery, or any other suitable 5V power source with a USB port.

2 pin **power wires** with black/white and red wires. They are part of the B4's power distribution system and transport electricity from module to module. Each module has one power input and 1-2 power outputs.





4 pin **data wires**. These transport 4 bit data and program counter signals between the modules.

1 pin **control wires**. They transport operation codes and instruct some of the modules of the B4 to do special things, such as storing data. The 1 pin wires come in many different colours. However, they all work the same and their colour has no influence on their function



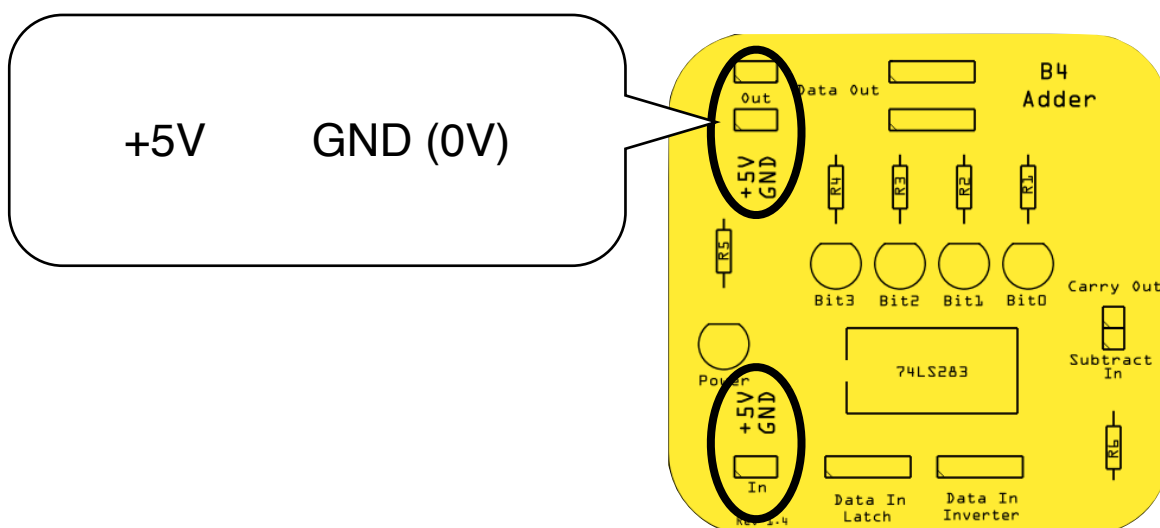
You will find corresponding connectors on the modules. The 2 and 4 pin connectors are directional and the wires will easily click into them. Unless you apply excessive force you should not be able to accidentally plug them in the wrong way.

In the diagrams in this book, we use the following wiring notation. A solid line denotes a power or data wire. A line with two arrows denotes a 1-pin control wire. This is just to make the setup a little bit easier for you.

| Symbol | Meaning |
|---|--------------------|
|  | power or data wire |
|  | control wire |

A Word about Power

Each of the B4's modules has an electric power distribution system on the left hand side of the modules. With the exception of the Program Counter, which connects to a USB port, the other modules have power in and out connectors.



+5V is on the left and GND (Ground, or 0V) is on the right. The power wires will automatically connect in the right way, **but sometimes we will need to connect a single wire to either +5V or GND during some of the experiments.** When asked to connect to +5V, just plug a single wire into the left pin of the power node. If asked to connect to GND, plug a single wire into the right pin of a power node.

Please look after me

The B4 is fairly robust and will last a long time with proper care. As long as you don't plug wires into connectors they are not designed to go in and as long as you don't drop the modules, step on them or use them as a doorstopper, things should be just fine. Always only plug the 2 pin wires into 2 pin connectors. the same applies to 4 pin wires and connectors. **Under no circumstances plug a 2 pin wire into a 4 pin connector.**

Ok, that is enough preparation for now. We will collect more details as we work through the experiments. Let's get started.

Exploration through Experimentation

In this book, we explore through experimentation. Yes, we conduct experiments during which we will be plugging wires into the B4 modules, let them work together and experiment with data and hardware. On occasion, when the bell rings at the end of the lesson, we may not quite be finished with an investigation. So that we don't have to take all our good work apart (and start from scratch next lesson), the B4's packaging also serves as a storage tray. The foam insert contains several cut-outs to keep the B4's modules safe during transport. But now that the B4 has arrived, we no longer need them. Let's turn the packaging into a lab:

Step 1

Remove all the B4 modules and wires from the packaging and place them on your desk.

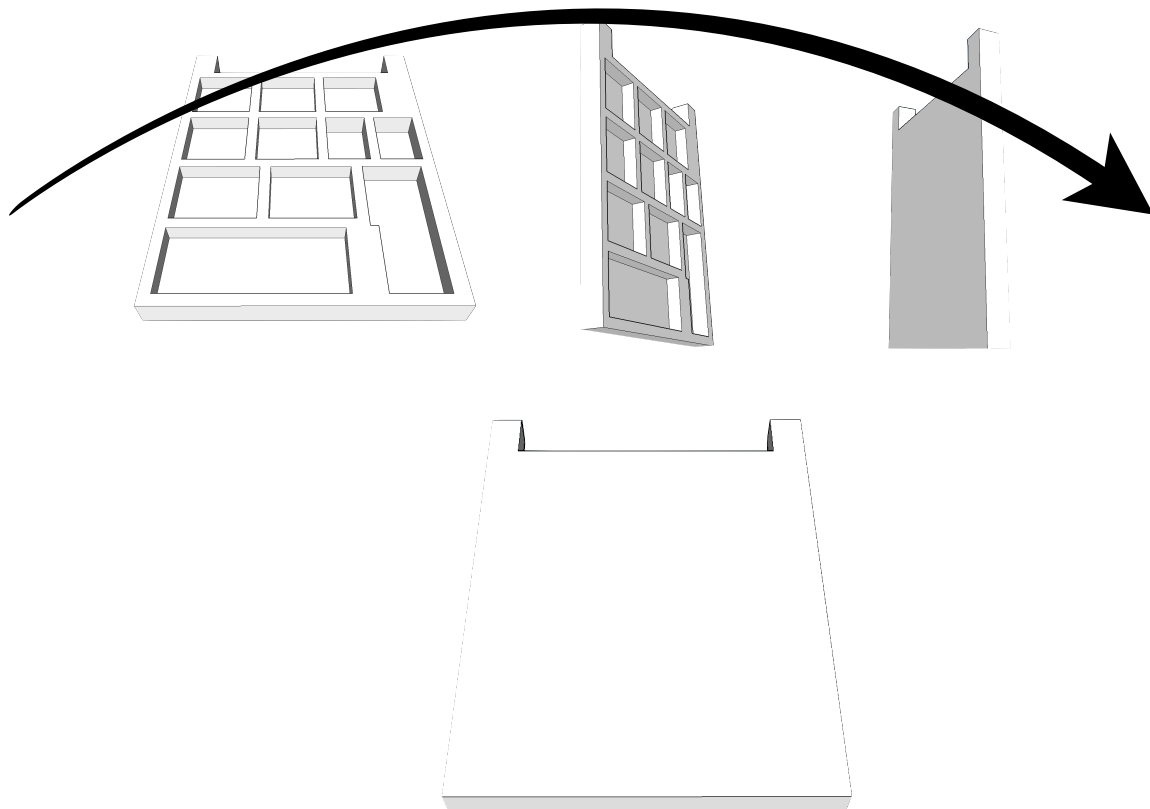
Step 2

Remove the foam insert from the box, flip it over - as shown below - and re-insert it with the flat side up. Reinsert it into the box.

Step 3

Place all modules neatly side-by-side on the foam insert and place all the wires in the cut-out at the top. This will keep the wires that we don't require for an experiment neatly in one place.

In the future, we can simply leave our experiments on the foam insert, close the lid, and place the box on a shelf - or anywhere else your teacher tells you.



Experiments

Overview

In this handbook, we have prepared several experiments that will help you to get to know the modules of the B4, how they are being used and what functions they perform. Most experiments consist of one or more experiments. You will learn how to combine modules to that they perform functions together, which they could not perform individually.

Ultimately, you will develop a computer and learn about coding from the ground up. You will also learn how a computer works internally and what critical role timing plays in the proper function of a computer's internal and external communication.

We recommend that you take the experiments in sequence. But if you are already a computer genius, feel free to jump around. We should mention that the B4 can do much more than what this handbook says. Feel free to explore and try out different things as you like.

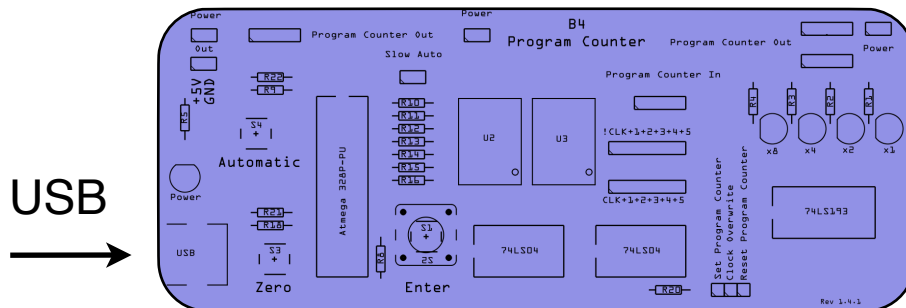
| Experiment | Title | Learning Objectives |
|------------|---|---|
| 1 | One Small Step ... | Function of the Program Counter module. Binary number system. Binary to decimal conversion. Clock signal. |
| 2 | Adding Two Numbers | Function of the Adder module. Binary Addition. Adding as a fundamental computational principle. Extension of the addition principle towards multiplication. |
| 3 | What about Subtraction? | Function of the Inverter module. Binary Complement. Extension of subtraction towards division. |
| 4 | Short Term Memory | Function of the Latch Module. Memory as a fundamental component of a computer to remember. Practical use of the Clock signal |
| 5 | Long Term Memory | Function of the Data RAM module. Stacking of memory. Data pointer. Practical application of the Program Counter. |
| 6 | Giving Direction to Data | Function of the 2-to-1 Data Selector module. Controlling the flow of data through a computer. |
| 7 | Let's Build a Manual Computer, Parts I and II | Extension of Experiment 2 towards addition of 3 numbers. What makes a computer so special? Infinite Loops. |
| 8 | Let's Make a Real Computer | Function of the Program RAM. Assembly of a computer capable of addition, subtraction and data storage. |

| Experiment | Title | Learning Objectives |
|------------|--|---|
| 9 | Programming the B4 | Manually programming of the Data and Program RAM modules. Execution of a pre-designed program that adds two numbers |
| 9a | Addition of three numbers | Extension of the program from experiment 9 to add a third number. |
| 10 | B4 Learns Subtraction | Extension of the program to include subtraction. |
| 11 | Automatic Programming | Function of the Automatic Programmer module for persistent storage of B4 programs on an Arduino and for rapid programming of the B4. Extension of the B4's programming principles. |
| 12 | Higher-Level Programming | Investigating a compact and conceptual notation. Assembly language and the role of an assembler. Shortening of the software design and testing cycle. |
| 13 | On the Role of Timing | Fundamental role of precise timing of the communication of the B4 modules. |
| 14 | So, how does a Computer work ... actually? | How complex logic problems can be expressed by Yes/No. Boolean logic. The role of gates and transistors and how higher-level computer functions are constructed, such as arithmetic units and memory. |
| 15 | Cyber Security | Hacking the library so that it alters data and program code. Making the Automatic Programmer module interfere with the normal operation of the B4 at runtime. |

Experiment 1: One Small Step ...

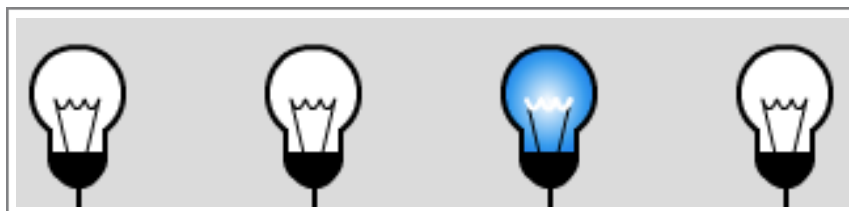
Modules Required: Program Counter

Take the Program Counter out of the box and connect it via a USB cable to your PC or Laptop.



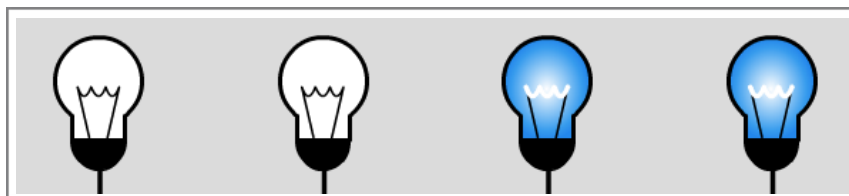
Setup of Experiment 1

The Program Counter only draws a little bit of electricity from your PC or Laptop. With it, it will power itself and all the other B4 modules, as we will see later. The red display will show a 'b4' for B4. On the Program Counter module, you will see a switch labelled *Enter*. Go ahead and press it. What happens? The display in the middle switches to 0. Press the button again and you will see a 1. Every time you press the Enter button, the number increases by 1. On the right hand side of the Program Counter, you will see 4 LEDs. They show exactly the same number as the display, but in binary. If your display shows a 2, then the LEDs will show a 0010, like this:



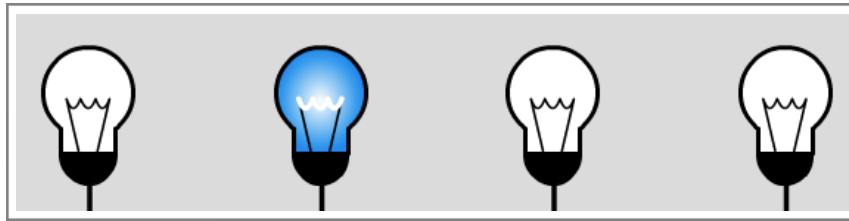
0010 in binary is a decimal 2

If you press the button again, the display will show a 3 and the LEDs will show the following pattern:



0011 in binary is a decimal 3

So, 3 is equal to 2+1. Which pattern will be displayed when you press the button again? The display shows a 4 and the LEDs will look like this: 0100



0100 in binary is a decimal 4

If we keep pressing the button, we will see more light patterns in our LEDs and the corresponding decimal numbers on the display. We can enter these into a table.

| binary | decimal |
|--------|---------|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | 10 |
| 1011 | 11 |
| 1100 | 12 |
| 1101 | 13 |
| 1110 | 14 |
| 1111 | 15 |

That's a lot of binary numbers. We could try to remember them by heart, but let's see if there is an easier way. Can you perhaps see a pattern in the table above?

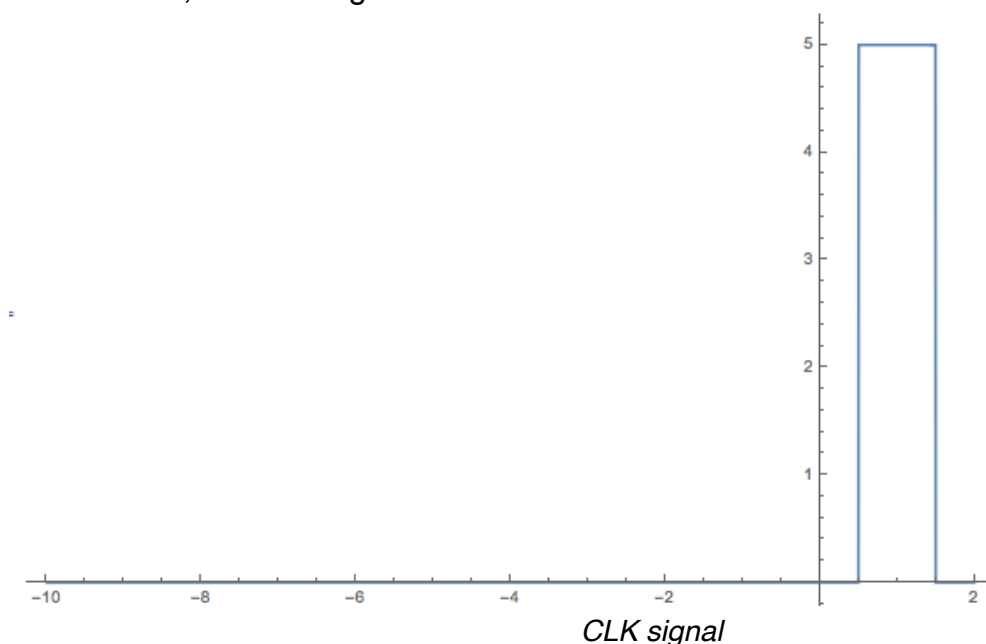
The right LED stands for a 1. The LED on its left stands for a 2, the one next to it for a 4 and the left LED stands for an 8. So, the number we see in the picture above is $8+4+1=13$. So, instead of remembering 16 different binary numbers, we only need to remember the decimal value that each LED represents and we can then easily calculate the number in our heads. What, you think there is an even easier way? Yes, you are right. These numbers double from right to left: 1, 2, 4, 8. **You only need to remember that the right**

LED represents a 1 and that the numbers double as we go from right to left. This means that we only have to remember **two** rules about binary numbers.

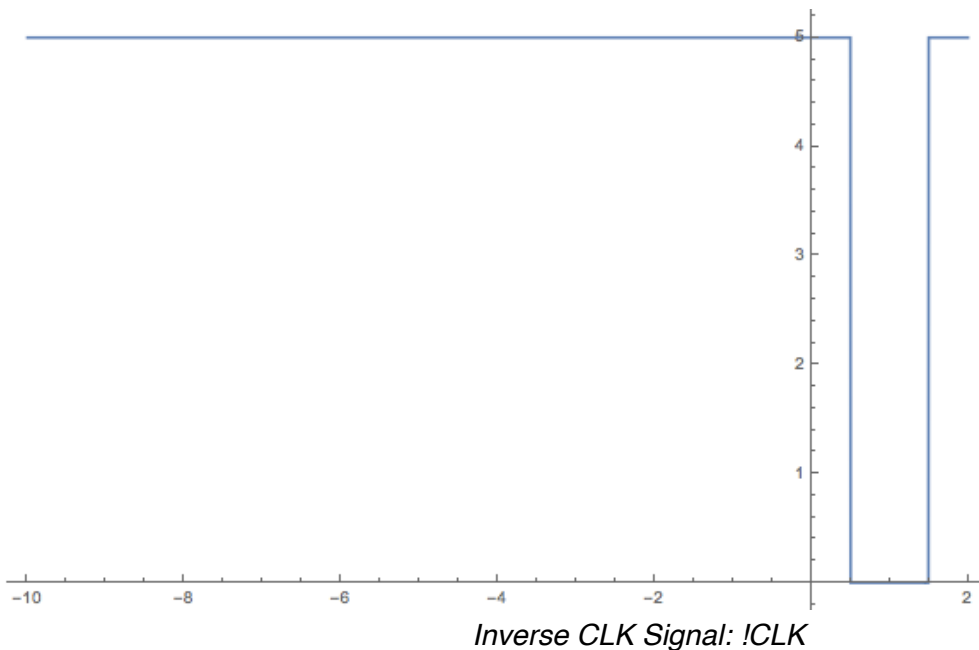
| Exercise 1.1 | |
|--------------|---|
| ? | What is the decimal value of 1111? |
| | What is the decimal value of 0110? |
| | What is the decimal value of 1010? |
| | What is the binary value of decimal 15? |
| | What is the binary value of decimal 12? |
| | What is the binary value of decimal 9? |
| | How can you easily spot an odd binary number? |

We have established that the Program Counter counts from 0 to 15, or, in binary numbers, from 0000 to 1111. By now you have probably discovered that it will tick over to 0000 after 1111. Why is this? Remember that our counter is binary and it is 4 bit only. Adding 1 to 1111 results to a 5 bit number, which is 10000. And because our little counter can't store the 5th bit, it will simply think that 0000 is the new number. This is not a bug, as we will see later.

What else does the Program Counter do? It produces the computer's **clock signal**, which computer scientists like to abbreviate as CLK. The clock signal is like the rhythm of music. You can also think of it as a heartbeat. Every time you press the button, the Program Counter will produce a little pulse. The signal jumps from 0Volt to 5Volt, reside there for a short period, and then drop back to 0V. We call 0V LOW and 5V HIGH. In the B4, the CLK signal is HIGH as long as you hold the Enter button pressed down. When you release the Enter button, the CLK signal returns to LOW



The program Counter also produces the inverse CLK signal, called !CLK. The !CLK signal is the opposite of CLK. It is normally HIGH and drops down to LOW whilst the Enter button is pressed. Our !CLK looks like this.



You might ask why we need both, CLK and !CLK? The answer is that some of the computer circuits require an activation signal to do something. The Latch, for example, requires a positive signal. Other circuits, such as our RAM modules are LOW active and need a LOW signal to store data.

We'll talk more about CLK and !CLK when we cover the Latch and RAM modules in the following experiments, and also at the end of the book in experiment 14, which is about timing.

Experiment 2: Adding Two Numbers

Modules Required: Program Counter, Adder, 2x Variable, Decimal Display

Analyse

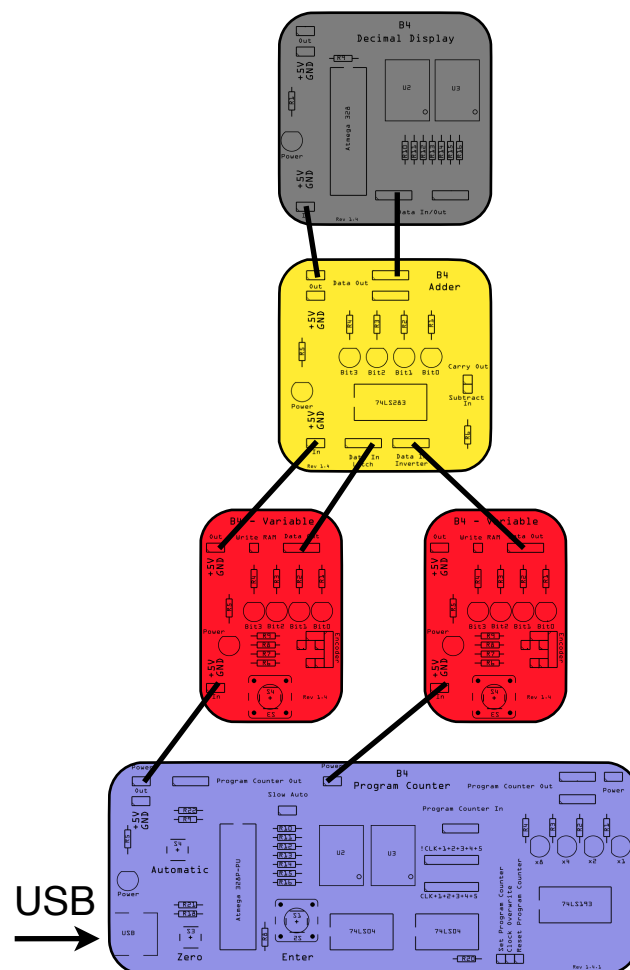
Adding numbers is the most fundamental thing a computer does. Addition is not only fundamental to computers. At school, in mathematics, students learn addition well before subtraction, multiplication and division.

Build

- 1) Connect the two Variables to the input of the Adder Module
- 2) Connect Power and Data cables as shown in the following diagram.

With the Variables, we will set numbers that we want the Adder to add up. In this experiment, we also require the Program Counter, but only to provide electricity from your Computer's USB port to the Variables and to the Adder module.

The following diagram shows the setup of this experiment.



Setup of Experiment 2

| Experiment |
|---|
| 1) Turn the knobs on both Variables until their LEDs are all off. 2) Turn the knob on one of the Variables so that just the right most LED turns on. |

| Observe |
|----------------------------------|
| What is the output of the Adder? |

| Compare |
|---|
| The right LED on the Adder will light up. |

| Understand |
|--|
| <p>That's because $0+1$ is 1.</p> <p>Turn the knob of the other Variable to show the 0001 LED pattern.</p> <p>The Adder will then show 0010. That's because $1+1=2$, which is 0010 in binary. Binary addition works just like the addition you already know with one difference, any number higher than 1 leads to a carry over. In the decimal number system that you already know, any number higher than 9 leads to a carry over. So, in a sense, binary addition is simpler than decimal addition.</p> <pre> 0001 0101 +0001 +0110 ----- 0010 1011 </pre> |

| Exercise 2.1 | Compute with your B4 and also on paper: |
|--------------|---|
| ? | What is $0101 + 1010$? |
| | What is $0010+0010$? |
| | What is $0111+0001$? |
| | What is $1111 + 0001$? Why are all the Adder's LEDs off? |

Experiment 3: What about Subtraction?

Modules Required: Program Counter, Adder, 2x Variable, Inverter, Decimal Display

Analyse

The aim is to subtract two numbers. We call them A and B, and the result is R.

We can express this as a mathematical equation in the form: **$A - B = R$**

Subtraction is addition where the second number is a **binary complement**. A complement is an opposite. The opposite of 0 is 1 and the opposite of 1 is 0. From there follows that the opposite of 10 is 01 and the opposite of 1111 is 0000.

Design

The Inverter module is a little machine that can produce complements of binary numbers. It can turn any 4 bit number into its complement. For this, the Inverter needs to be activated. We do this by connecting a wire from the Inverter's *Subtract In* pin to the +5V Pin. The LEDs on the Inverter Module will then show the opposite of the LEDs of the connected Variable. So, if the Variable displays a 1100, then the Inverter will show 0011.

Build

- 1) Use the setup from experiment 2
- 2) Insert the Inverter module between the right Variable and the Adder
- 3) Connect a wire from the Subtract Out Pin of the Inverter to the Subtract In Pin of the Adder module
- 4) Set the Variable to 0000 (all LEDs are off).

The following diagram shows the setup of this experiment.

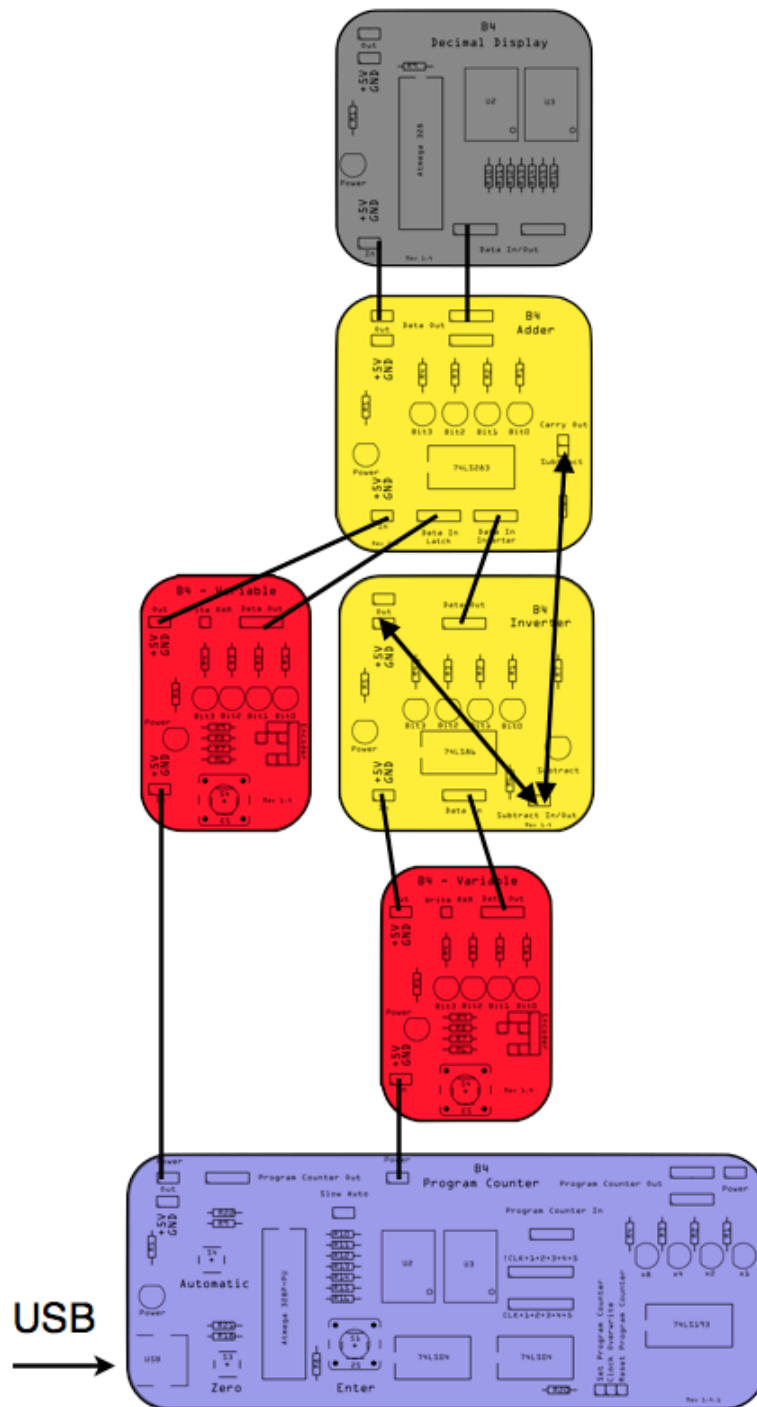
Experiment

We now compute 4 minus 3.

4 is 0100 in binary and 3 is 0011 in binary.

- 1) Enter 0100 into the left Variable
- 2) Enter 0011 into the right Variable.

Connect a wire from the Inverter's *Subtract In* pin to a +5V Pin as shown in the following diagram



Setup Experiment 3

Observe

What is the output of the Inverter? What is the output of the Adder?

Compare

The Inverter Displays the complement of 0011, which is 1100. The Adder shows 0001.

Evaluate

The binary complement of 0011 is 1100,
We can add 4 and the binary complement of 3 as follows:

```

  0100 (4)
+ 1100 (Binary Complement of 3)
-----
 10000
+   1
=====
 10001

```

This is a 5 bit number, but the B4 can only store 4 bit numbers, so the B4 cuts off the leading 1 and the final result of 4 minus 3 is 0001.

In general terms: $A - B = A + \text{Binary Complement of } B + 1$

This works for any numbers A and B. Try it out !

Evaluate Deeper

What about Division?

Division is a series of subtractions. For example the result of 15 divided by 5 can be computed by subtracting 5 from 15 until the result is less than 5, so

```

15-5=10
10-5=5
5-5=0.

```

3 steps were required until the result was less than 5, so we conclude that 15 divided by 5 is 3.

| Exercise 3.1 | Compute with your B4 and also on paper: |
|--------------|---|
| ? | 3 minus 0 |
| | 5 minus 2 |
| | 10 minus 0 |
| | 15 minus 15 |
| | 2 minus 3. What do you see? |

Experiment 4: Short Term Memory

Modules Required: Program Counter, Latch, Variable

Analyse

In our brains we have short term memory that helps us to remember things that are important for us right now. For example, we remember that we are holding something in our hands, such as a pencil. Imagine how funny it would be if we forgot this and then tried to scratch our nose because it was itchy. Short term memory is also useful when we want to calculate $3+8+1$. Because this calculation we usually carry out as $3+8=11$ and then add 1 to reach the final result of 12. If we couldn't remember the intermediate result, we would never be able to add three numbers. You could even argue that we would not be able to add two numbers because by the time we looked at the second number we would have forgotten about the first one.

Design

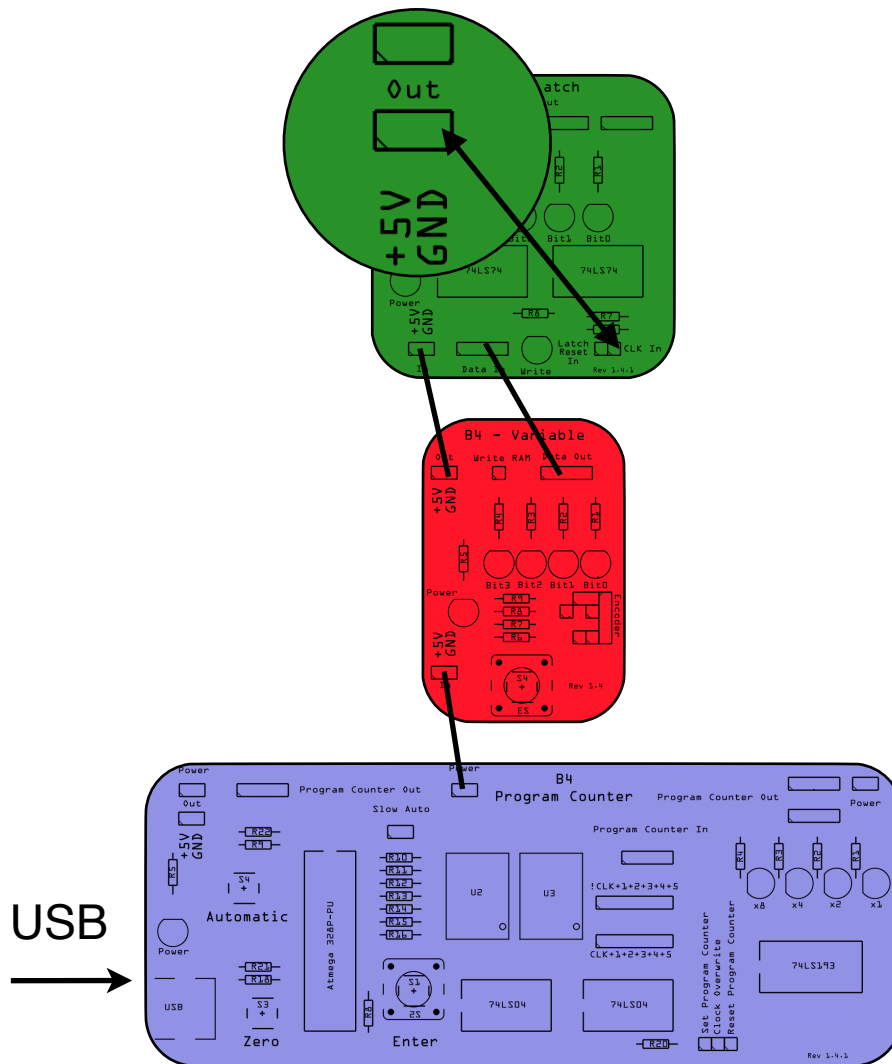
Like us, computers have short term memory. In the B4, the Latch module performs this function. The Latch 'latches' on the last computation and remembers its result. It has 4 bit of storage capacity and could be described as a house with 4 rooms. In each room, it can store exactly 1 bit.

Build 4.1

- 1) Use the Program Counter, Variable and Latch and arrange the modules as shown in the following diagram
- 2) Connect the modules with wires as shown.
- 3) Connect a 1-pin control wire to the Latch's CLK In pin and from there to GND as shown.

With the Variables, we will set numbers that we want the Latch to remember. In this experiment, we also require the Program Counter, but only to provide electricity from your Computer's USB port to the Variables and to the Latch module.

The following diagram shows the setup of this experiment.



Setup of Experiment 4, Part 1

Experiment 4.1

Set a number, for example 0011, on the Variable

Observe 4.1

What does the Latch do?

Compare 4.1

Nothing. The LEDs on the Latch do not change.

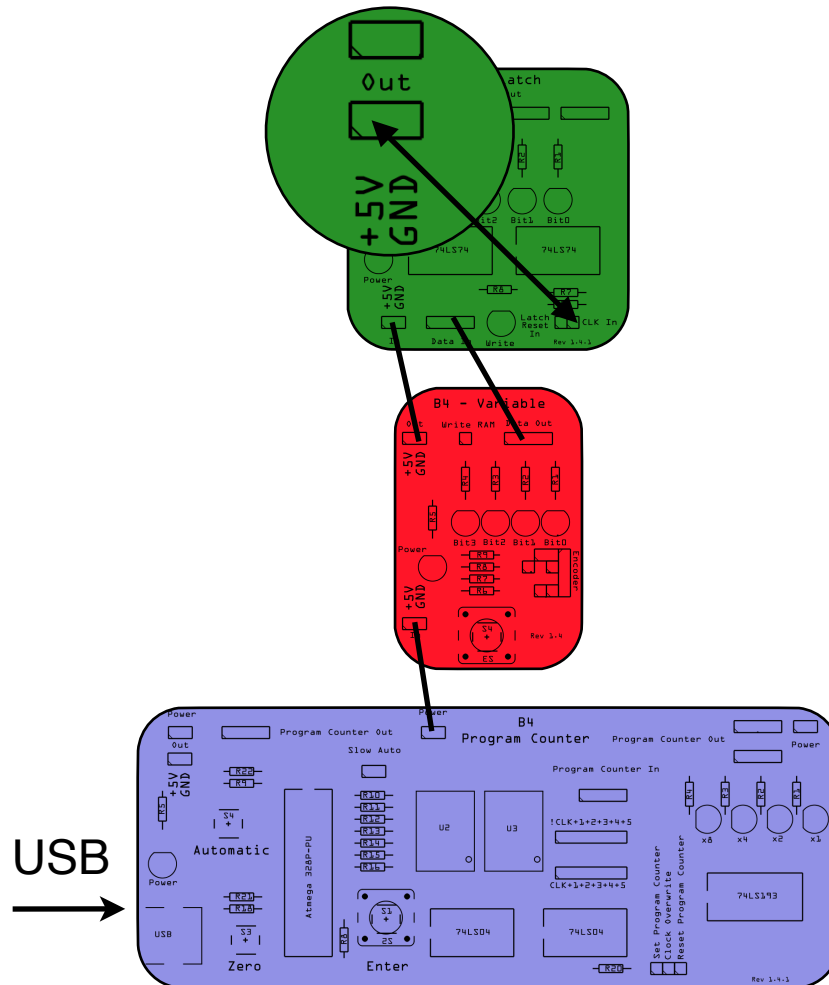
Evaluate 4.1

The Latch is waiting for an activation signal. This is really important, as we need to tell the Latch *when* it should remember something. In the B4, this signal is the CLOCK signal that we have encountered in experiment 1.

Experiment 4.2

We now want to send a CLOCK signal to the Latch.

1) Using a 1-pin control wire, connect the CLK In pin of the Latch to a +5V pin.



Setup of Experiment 4, Part 2

Observe 4.2

What does the Latch do?

Compare 4.2

We observe that the Latch now also shows 0011, which is the value at the output of the Variable.

Evaluate 4.2

The Latch has received an activation signal. This causes it to remember the data at its input port.

Experiment 4.3

We now want to explore what happens when the data changes, whilst the Latch still receives the activation signal. Leave the control wire connected and change the number on the Variable to, let's say, 0100.

Observe 4.3

What does the Latch do? Will the LEDs on the Latch change too?

Compare 4.3

No. The LEDs on the Latch do not change.

Evaluate 4.3

The Latch will only look at the data on its input side when CLK changes from LOW to HIGH, or from 0 Volt to 5 Volt.

Experiment 4.4

What happens to the Latch when the CLK signal changes?
1) Disconnect the CLK wire from +5 and then re-connect it to +5V again

Observe 4.4

What does the Latch do? Will the LEDs on the Latch change?

Compare 4.4

Yes, the value of the Latch is now 0011

Evaluate 4.4

By disconnecting and re-connecting the wire we have made our own CLK signal. This is nice, but a bit impractical for a real computer as we don't want to always plug wires in or out. Do you remember from experiment 1, that one of the functions of the Program Counter is the production of the CLK Signal?

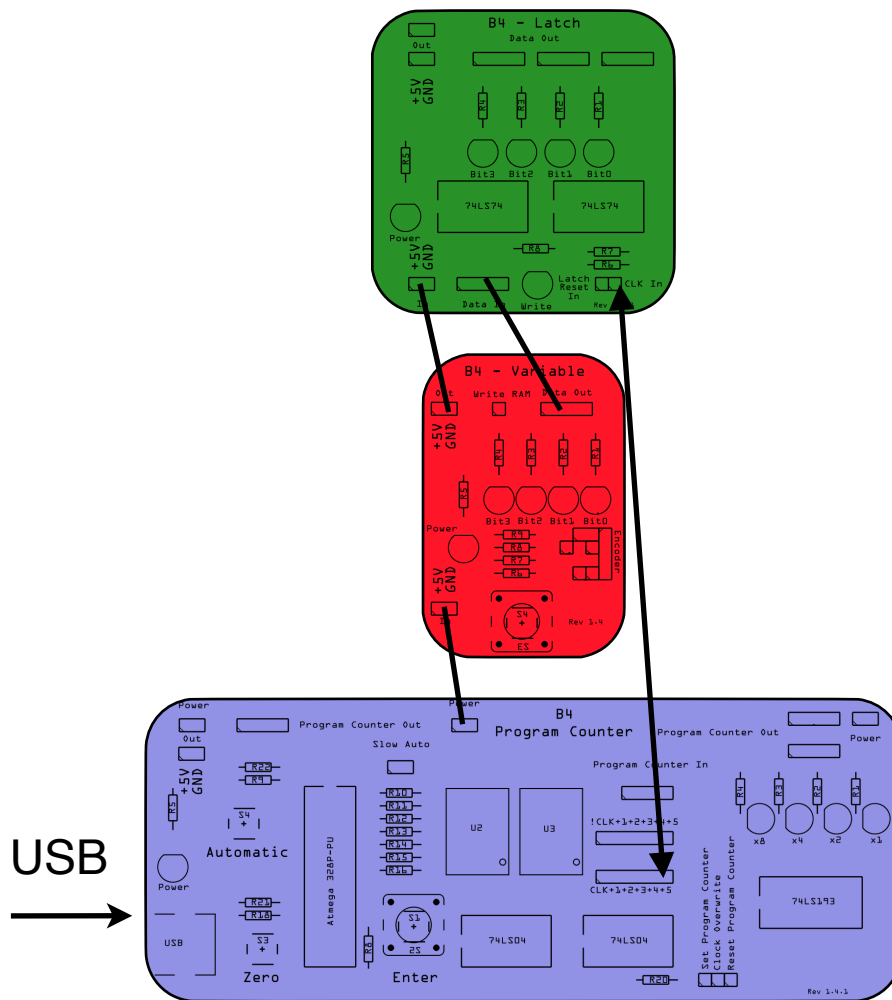
Design 4.5

We want the Latch to work without us having to change wires around. We do this by connecting the Latch to the CLK signal that the Program Counter generates.

Build 4.5

We re-wire our experiment a little, as shown in the next diagram:

Move the control wire from +5V to the CLK+5 Pin on the Program Counter module as shown in the following diagram



Setup of Experiment 4, Part 5

Experiment 4.5

- 1) Change the value on the Variable to, let's say, 1000
- 2) Press the Enter button on the Program Counter module

Observe 4.5

What does the Latch do? Will the LEDs on the Latch change?

Compare 4.5

Yes, the value of the Latch changes to 1000.

Evaluate 4.5

Pressing the enter button on the Program Counter produces the CLK signal, which is sent to the Latch. The Latch will then store 1000.

We have just found an automatic way to activate the Latch. This will become very useful, as you will see in experiment 7B.

Experiment 5: Long Term Memory

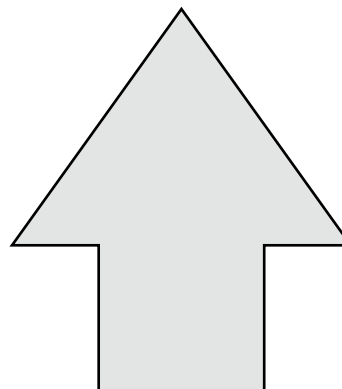
Modules Required: Program Counter, Data RAM, Variable

Analyse

In experiment 4, we looked at the B4's short term memory, which we likened to a house with 4 rooms. The Data RAM is, in this analogy, a 16-floor high-rise with 4 rooms on each floor.

To store data into the Data RAM Module, we first want to tell it on which floor we want our data to be stored. Then, we give it 4 bits of data and finally tell it to actually store it.

| | | | | |
|----------|--|--|--|--|
| Floor 15 | | | | |
| Floor 14 | | | | |
| Floor 13 | | | | |
| Floor 12 | | | | |
| Floor 11 | | | | |
| Floor 10 | | | | |
| Floor 9 | | | | |
| Floor 8 | | | | |
| Floor 7 | | | | |
| Floor 6 | | | | |
| Floor 5 | | | | |
| Floor 4 | | | | |
| Floor 3 | | | | |
| Floor 2 | | | | |
| Floor 1 | | | | |
| Floor 0 | | | | |

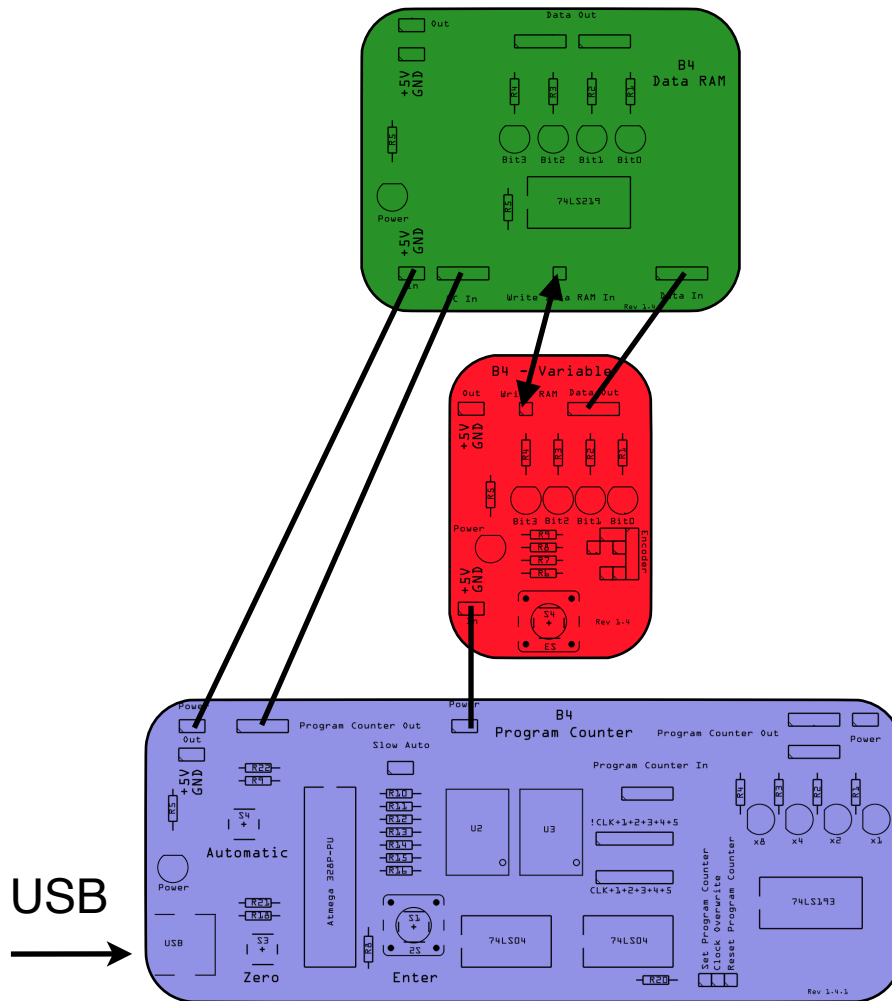


Program Direction

Data RAM: 16x 4 bit.

Build 5.1

- 1) Connect the Program Counter Out Pin to the PC In pins of the Data RAM module.
- 2) Connect the Output of the Variable to the Data In port of the Data RAM
- 3) Connect the Write RAM Pin of the Variable to the Write Data RAM In pin of the Data RAM with a 1-pin control wire.



Setup of Experiment 5

Experiment 5.1

- 1) Press the Reset button on the Program Counter module until it is at step 0000.
- 2) Set the data on the Variable to 1010.
- 3) Press the button on the Variable to store the data. The Data RAM's LEDs will now display 1010. Congratulations, you have just stored 4 bit of data in the Data RAM!!
- 4) Press the Enter button on the Program Counter until it is at step 0001.
- 5) Change the data on the Variable to 0101.
- 6) Press the button on the Variable to store the data. You have just stored your second 4 bit of data in the Data RAM

Change the data on the Variable.

Observe 5.1

What happens to the LEDs on the Data RAM module? Do they change as well?

Compare 5.1

You will notice that the LEDs on the Data RAM will remain unchanged unless you press the button on the Variable.

Evaluate 5.1

You will remember from experiment 1 that the Program Counter can generate numbers from 0 to 15. As it happens, this is exactly the number of floors in our Data RAM high-rise. **The Program Counter will therefore tell the Data RAM module which of its floors we want to access.** You can imagine it as an elevator that moves upwards, one floor at a time, starting on floor 0 and finishing on floor 15. It will then drop, like a stone, back to floor 0.

The Variable can do two things:

- a) On it, we will generate the data we want to store in the Data RAM module, and
- b) Send a 'Store' command to the Data RAM module when we press the button on the Variable. The Data RAM then stores the data from the Variable, to the floor that the program counter indicates.

During programming and operation of the B4, ensure that the Program Counter remains powered and that the Data RAM and the Program RAM modules remain connected to power, too. This ensures that the RAM modules don't forget their data.

Experiment 5.2

Fast-forward again to 0000 on the Program Counter by pressing its the Reset button.

Observe 5.2

Which value do you see on the Data RAM?

Compare 5.2

Once you arrive at step 0000, the Data RAM will show 1010 again.

Random Data

You have probably noticed that there is all sorts of data in the Data RAM that you have not stored there. Where does it come from? The Data RAM consists of hundreds of tiny little switches. When the Data RAM is plugged into power, then some of them are randomly open and some of them are randomly closed. That's not a problem.

Experiment 5.3

You can manually clear the RAM by doing the following: First, set the Variable to 0000. Then, set the Program Counter to 0000 as well.

- 1) Repeat
- 2) Press the Button on Variable to store 0000 into Data RAM
- 3) Press the Enter Button on Program Counter
- 4) Until Program Counter displays 0000 (that's 1111+1)

Evaluate 5.3

Did you notice? You have just completed your **first algorithm** on the B4. Specifically, it is a Repeat Loop.

Experiment 6: Giving Direction to Data

Modules Required: Program Counter, 2-to-1 Selector, 2x Variable

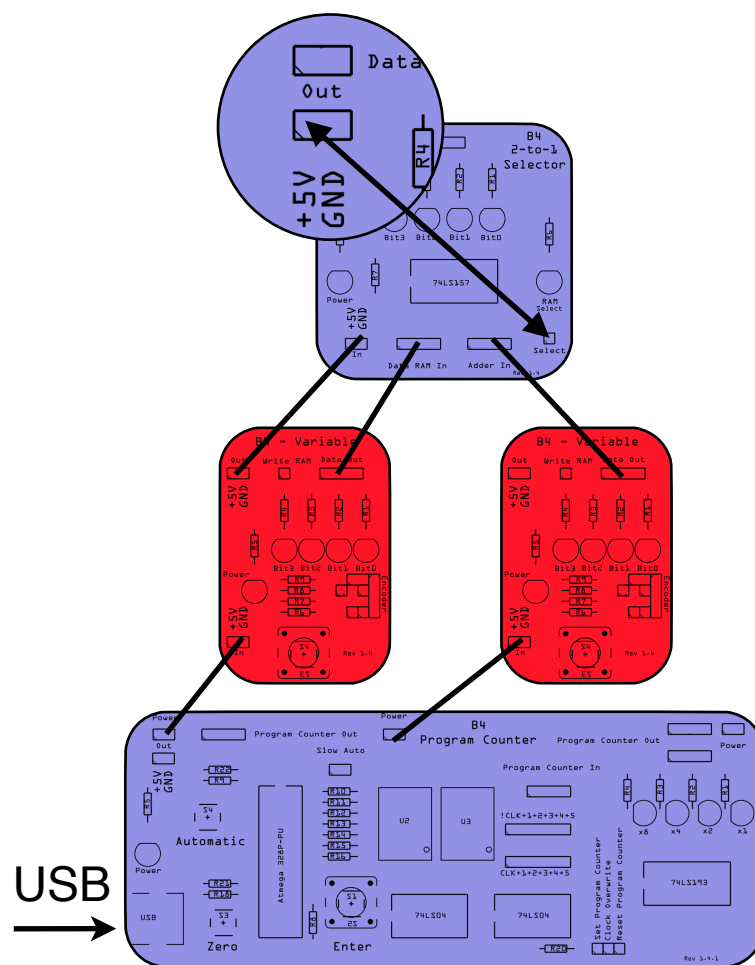
Analyse

Imagine, if you will, a motorway that goes straight through the countryside. However, someone forgot to build the on-ramps. If you wanted to drive on this motorway, you could only get on it at the beginning. This would be quite a silly motorway. In a computer we also need on-ramps, or, as we call them, selectors, to influence the path that data travels through the computer.

The B4 has one 2-to-1 Selector. It is comparable to an on-ramp on a motorway. As we will later see, the B4 uses its 2-to-1 Selector to choose whether it wants data from the Data RAM or from the Adder to reach the Latch (we learned about the Latch in experiment 4).

Build 6.1

Connect the two Variables to the 2-to-1 Selector as shown. The control wire goes from the Select pin to +5V. The blue RAM Select LED lights up.



Setup of Experiment 6

Experiment 6.1

Set the left Variable to 1010 and the right Variable to 0101.

Observe 6.1

What do you see on the LEDs of the 2-to-1 Selector?

Compare 6.1

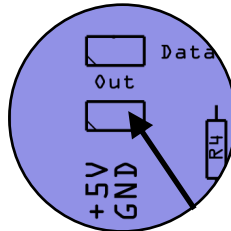
The 2-to-1 Selector will show 0101, which is the data from the left Variable.

Evaluate 6.1

By default it will show the data from the left Variable.

Experiment 6.2

Connect a control wire from the Select pin to GND.



Observe 6.2

What do you see on the LEDs of the 2-to-1 Selector?

Compare 6.2

The 2-to-1 Selector will show 0101, which is the data from the right Variable. The RAM Select LED is off.

Evaluate 6.2

The control wire influences from which input port the 2-to-1 Selector chooses the data that it feeds to its output port. We have just learned how we can influence the direction of data. As we will see later, the B4 uses its 2-to-1 selector to choose whether to route data from the data RAM or from the adder to the cache (we met the cache in Experiment 4).

Experiment 7a: Let's Build a Manual Computer, Part 1

Modules Required: Program Counter, 2-to-1 Selector, Adder, 2x Variable

During the first six experiments we have explored the parts that make a computer. In this experiment, we will explore what makes a computer so special, namely its ability to add a series of numbers. In experiment 2 we learned how we can add two numbers. We did this simply by sending them to the input ports of our Adder modules. We know that our Adder limits us to the processing of two numbers at any given time.

But how can a computer add three numbers?

Analyse

The aim is to add three numbers. We call them A, B, and C and the result is R.

We can express this as a mathematical equation in the form: **$A + B + C = R$**

$A + B + C$ is the same as first calculating $A+B$ and then adding C to the result of $A + B$. $A + B = D$ which we call the intermediate result. We therefore **decompose** the addition of three numbers into two additions of two numbers each:

Step 1: $A + B = D$

Step 2: $D + C = R$

For this to work, we require a mechanism to remember the intermediate result D. We will explore this in experiment 7b. Let's now investigate where the data comes from and where it needs to go to:

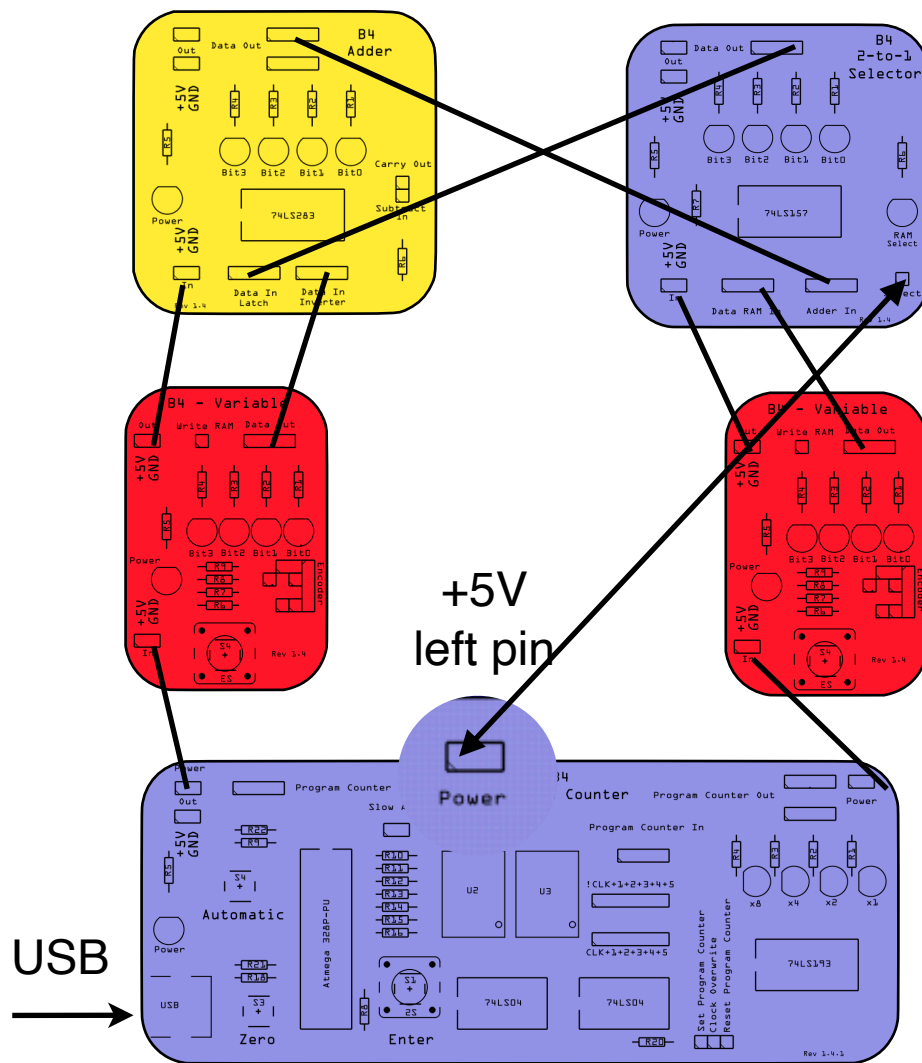
| Data | Origin | Destination |
|-----------------------|------------|-------------|
| A (our first number) | Variable 1 | Adder |
| B (our second number) | Variable 2 | Adder |
| C (our third number) | Variable 1 | Adder |
| D (the sum of A+B) | Adder | Adder |

Analyse & Design

A, B and C come from one of two Variables, but D comes from the Adder. Because it would be a bit impractical to set one of the Variables to D, we need to find an automatic way for one of the Adder input ports to receive data from **either** the Variable **or** the Adder's output port. Does this remind you of something? Yes, the 2-to-1 Selector can do this because it can select one of two data sources and feed the data to its output port.

Build

- 1) Insert the 2-to-1 Selector into the wiring from experiment 2 as illustrated in the following diagram.
- 2) Then, connect its output to the input of the Adder
- 3) Then, connect the output of the Adder to the right input of the 2-to-1 Selector
- 4) Connect the right Variable to the remaining input port of the 2-to-1 Selector
- 5) Connect the Select pin of the 2-to-1 Selector with +5V as shown



Setup of Experiment 7a

Let's try to add $1+1+4$.

Experiment

Set both Variables to 0001

Observe

What value is at the output of the Adder?

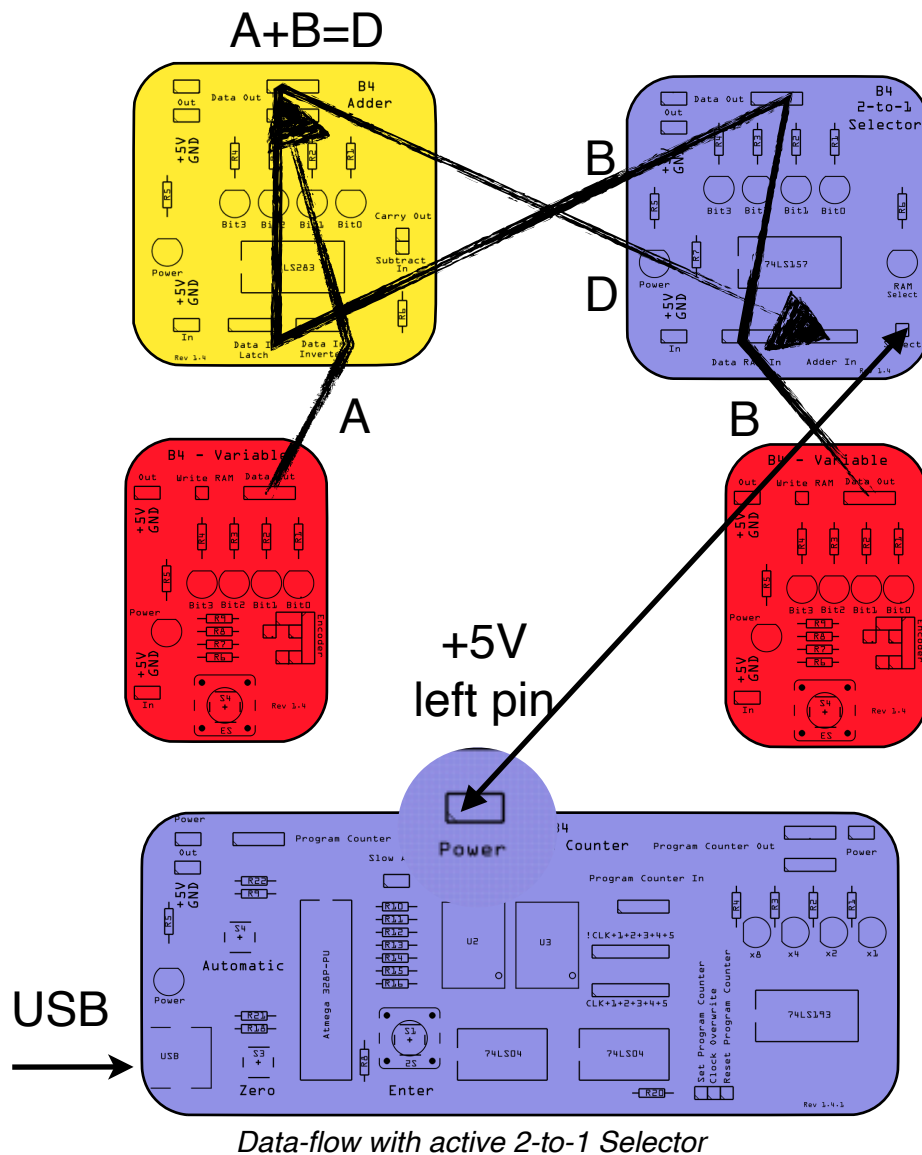
Compare

The Adder displays 0010 (decimal 2)

Evaluate

To understand this, we explore the flow of data. Data flow is the movement of the data through and between our modules. We have drawn this in the next diagram. To make it easier to observe, the power and data wires have been omitted from the diagram.

The number B from the right Variable goes into the 2-to-1 Selector and from there into the Adder. There, it meets the data from the left Variable, A, and the adder computes $A+B=D$, which is B0010, which the Adder sends to the 2-to-1 Selector, But because the Selector listens to data from the right Variable, it simply ignores D. The data flow stops there. So the Adder now shows the sum of our addition, which is B0010. We have successfully added two numbers.



But our goal is to add three numbers. We continue with our experiment.

| Experiment |
|---|
| Set the left Variable to C, which is B0100 (decimal 4). |
| Observe |
| What value is at the output of the Adder? |
| Compare |
| Immediately, the output of the Adder changes to 4+1=5. |

That's not good. But let's ignore this for a moment. We will solve this later, in the next experiment.

Experiment

Deactivate the 2-to-1 Selector to feed D back into the Adder. We do this by changing the control wire from +5V to GND on the Program Counter. The 2-to-1 Selector's right blue LED goes out.

Observe

What value is at the output of the Adder?

Compare

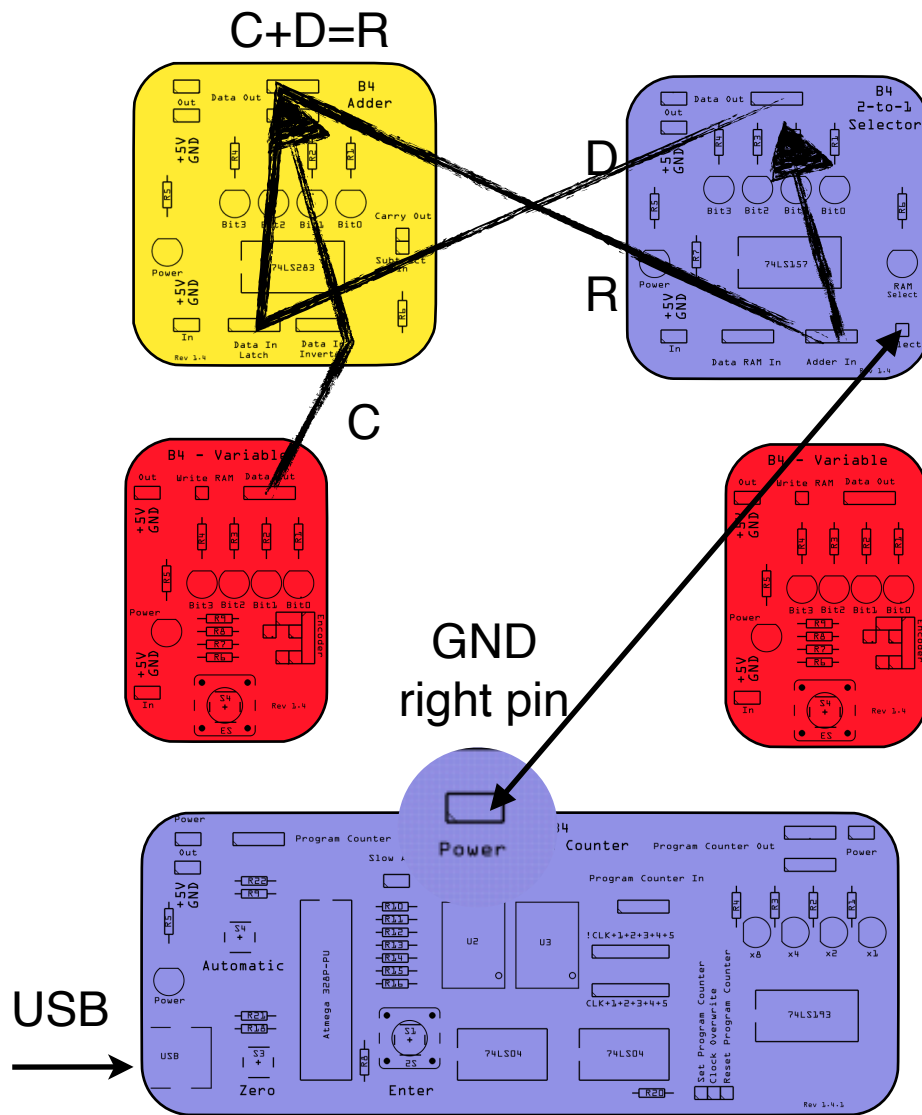
The LEDs of the Adder suddenly switch on. But some of them are brighter than the others. The Adder is not showing the expected result 6 (0110), which would have been the result of the addition of 2 (from the previous step) and 4 from the left Variable.

Evaluate

To explain this unexpected behaviour, we need to take a look at the data flow, which you can see in the following diagram. In the beginning, D reaches the output of the 2-to-1 Selector (because the Selector is listening to the Adder). C and D are at the input of the Adder, which computes $C+D=R$.

So far so good, but what happens then? The result R gets from the output of the Adder to the input of the 2-to-1 Selector, which again feeds it through to the input of the Adder. The Adder performs $C+R=R'$, which it again provides to the input of the 2-to-1 Selector. This results in R'' , then R''' and so forth. So R never stands still, it keeps moving. In fact, we have just made a loop. This is the famous infinite loop that you might have heard of. Infinite means that it does not have an end.

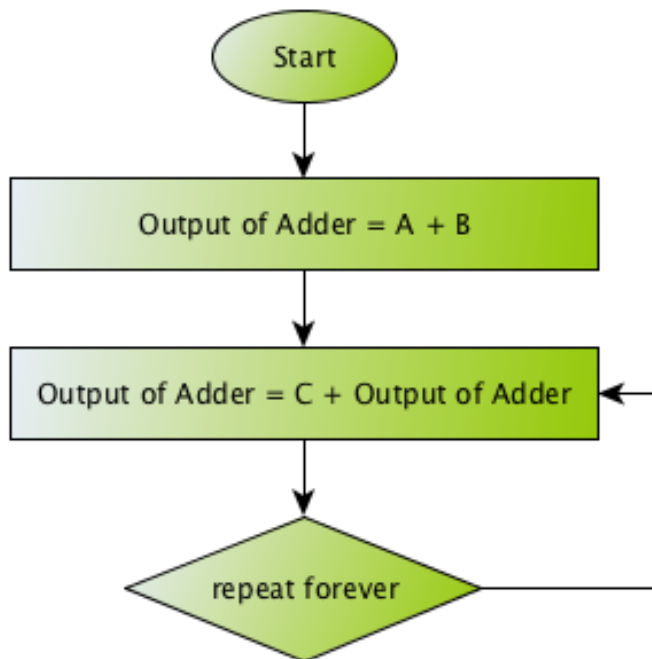
The infinite loop is so famous that there is even a road named after it in Cupertino, California, USA, where the Apple offices are.



Data-flow with inactive 2-to-1 Selector

Evaluate deeper

We can visualise this in a flowchart:



Infinite Loop as Flowchart

We begin with :

A=1

B=1

C=4

$A+B=D=2$

Round 1: $C+D=R=4+2=6$

Round 2: $C+R=R'=4+6=10$

Round 3: $C+R'=R''=4+10=14$

Round 4: $C+R''=R'''=4+14=18$, but that's a 2 in a 4-bit system

Round 5: $C+R'''=R''''=4+2=6$

So this produces the series 2, 6, 10, 14. We can write this in binary:

| Decimal Value | Binary Value |
|---------------|--------------|
| 2 | 0010 |
| 6 | 0110 |
| 10 | 1010 |
| 14 | 1110 |

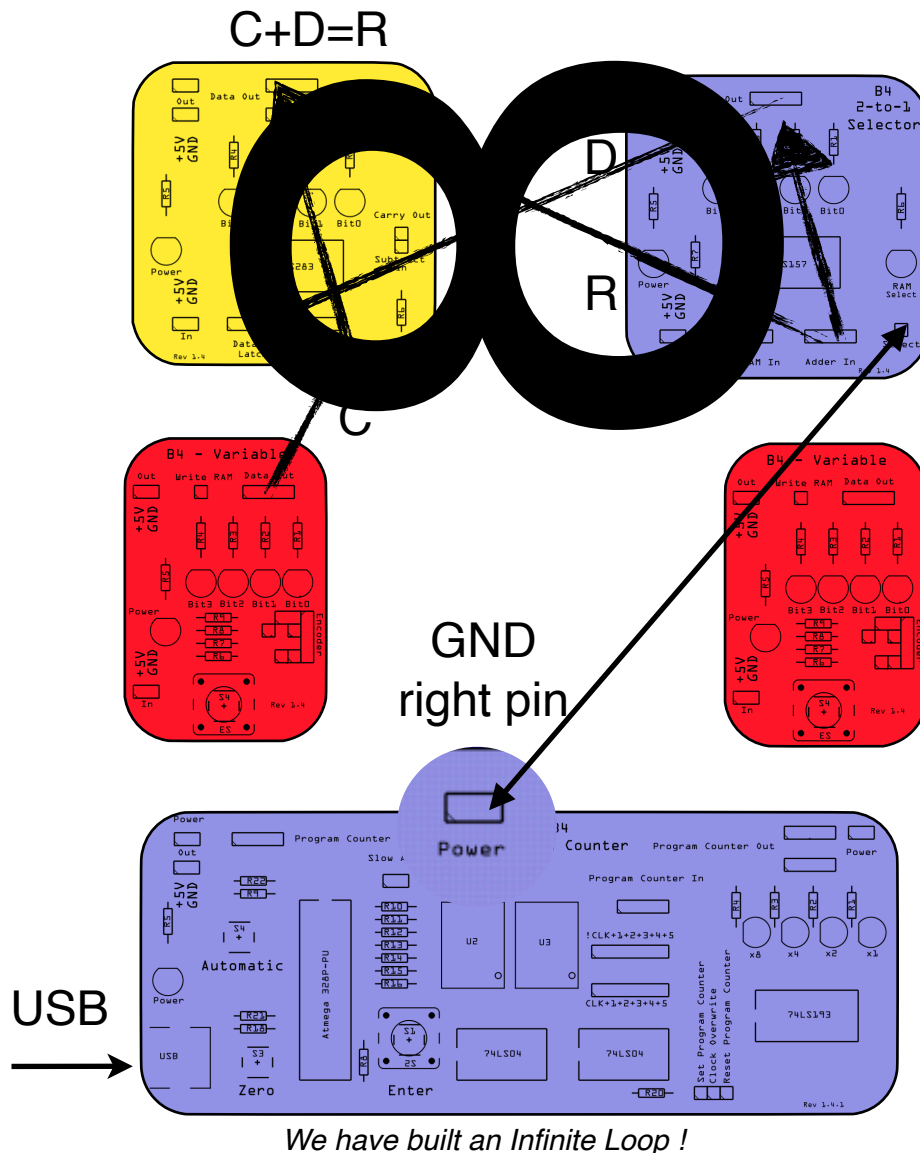
Let's look at the individual bits more closely

| | Decimal Value | Binary Value | bit3 | bit2 | bit1 | bit0 |
|-----|---------------|--------------|------|------|------|------|
| | 2 | 0010 | 0 | 0 | 1 | 0 |
| | 6 | 0110 | 0 | 1 | 1 | 0 |
| | 10 | 1010 | 1 | 0 | 1 | 0 |
| | 14 | 1110 | 1 | 1 | 1 | 0 |
| Sum | | | 2 | 2 | 4 | 0 |

So, bit1 is always active, but bit2 and bit3 are lit only every second time and bit0 is not used at all. So we would expect that the LEDs 3, 2, and 1 on the Adder are lit, but LED 0 stays off. And this is what we see.

In fact, R moves so fast that we cannot see it changing with our naked eye. In our lab, we have measured the change of R with an oscilloscope and found that it changes 16 million times per second. That's how fast our chips are.

You can also think of this as a figure of eight, representing an endless loop of data flow. We have drawn it in the following diagram.



We seem to be on the right track by using the 2-to-1 Selector, but we have two big problems:

- 1) The Adder changes its output as soon as we enter the third number C.
- 2) Once we activate the 2-to-1 Selector, the output of the Adder keeps changing 16 million times per seconds.

We need something that stops the infinite loop in a controlled way. This module needs to remember the result of A+B. This is our challenge for the following experiment.

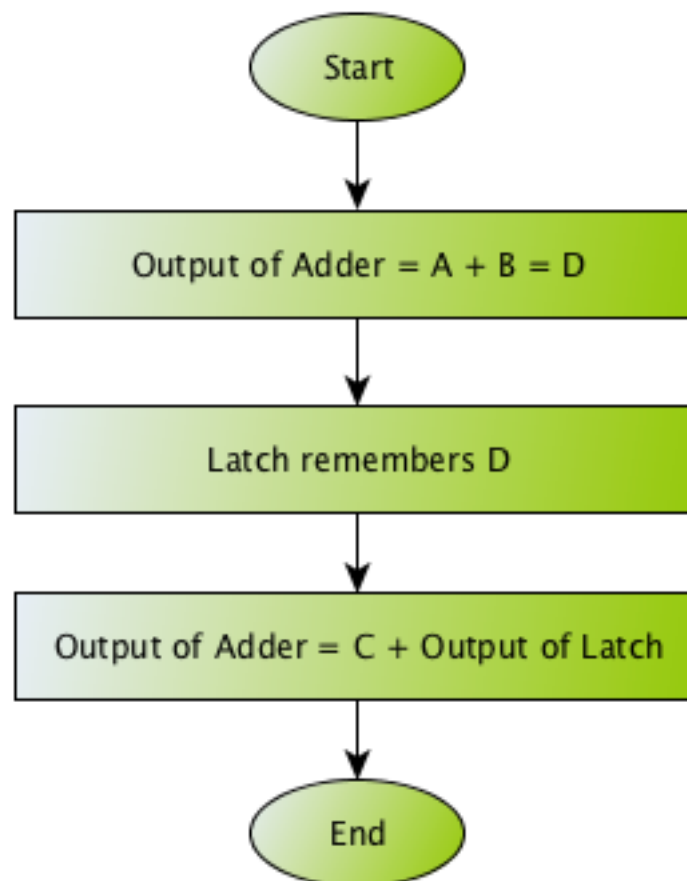
Experiment 7b: Let's Build a Manual Computer, Part 2

Modules Required: Program Counter, 2-to-1 Selector, Adder, 2x Variable, Latch

In the previous experiment, we tried to build a manual computer, but encountered a problem with an infinite loop. In this experiment, we will solve this problem and develop a simple manual computer that already has a number of the characteristics of a 'real' computer.

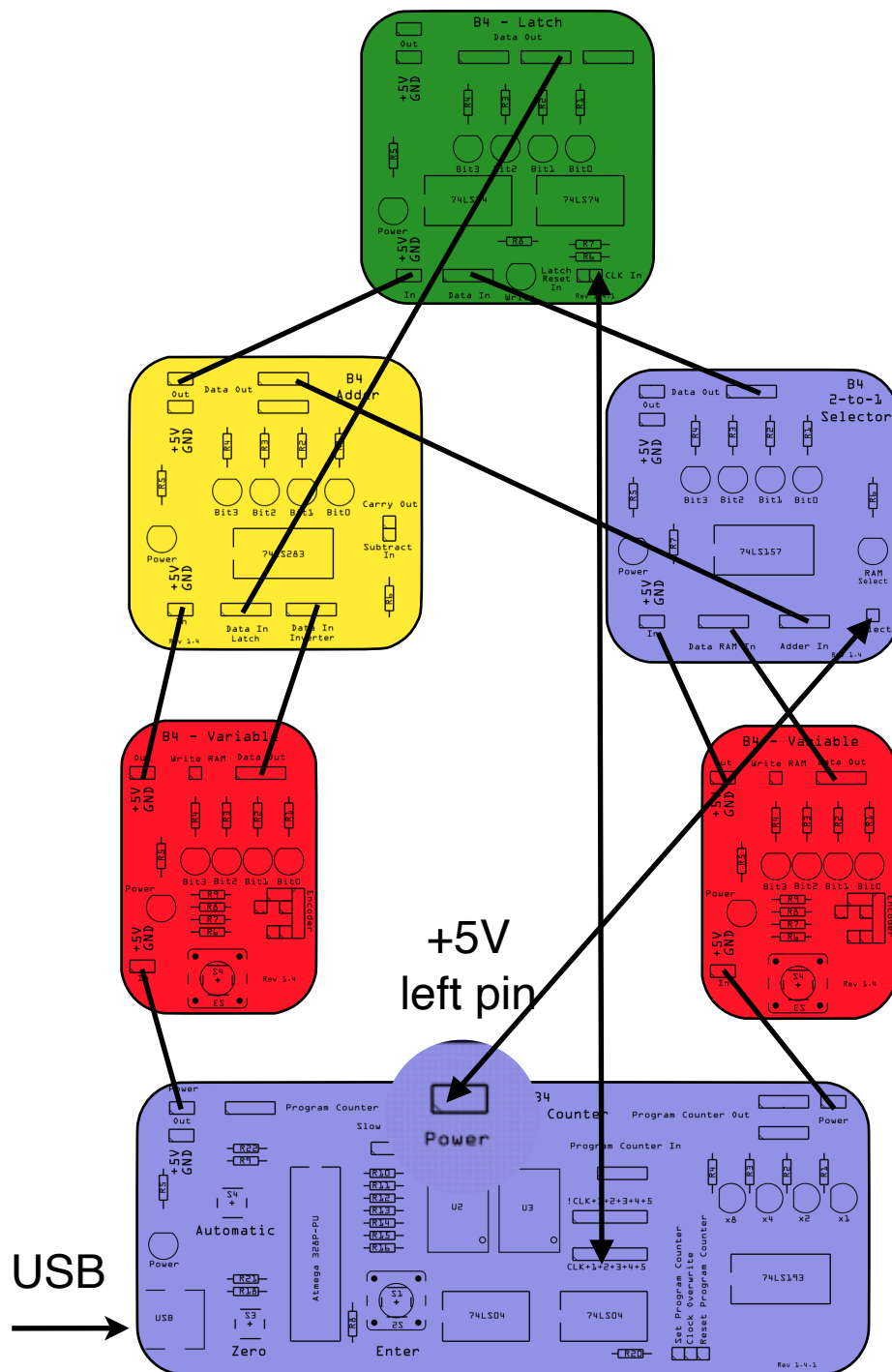
In experiment 4, we learned about the Latch. It can remember 4 bits, but only when it receives an activation signal. This sounds like a promising solution to our infinite loop problem.

With the Latch, we can change our data flow to the following:



The Latch breaks the Infinite Loop

As shown in the following diagram, insert a Latch between the output of the 2-to-1 Selector and the input of the Adder (We could just as well place it between the output of the Adder and the input of the 2-to-1 Selector). To control the Latch, we connect the Latch CLK Input to any of the the Program Counter's CLK outputs. They are the bottom row of pins as shown. Make sure you don't accidentally connect to the pins in the row above (!CLK).



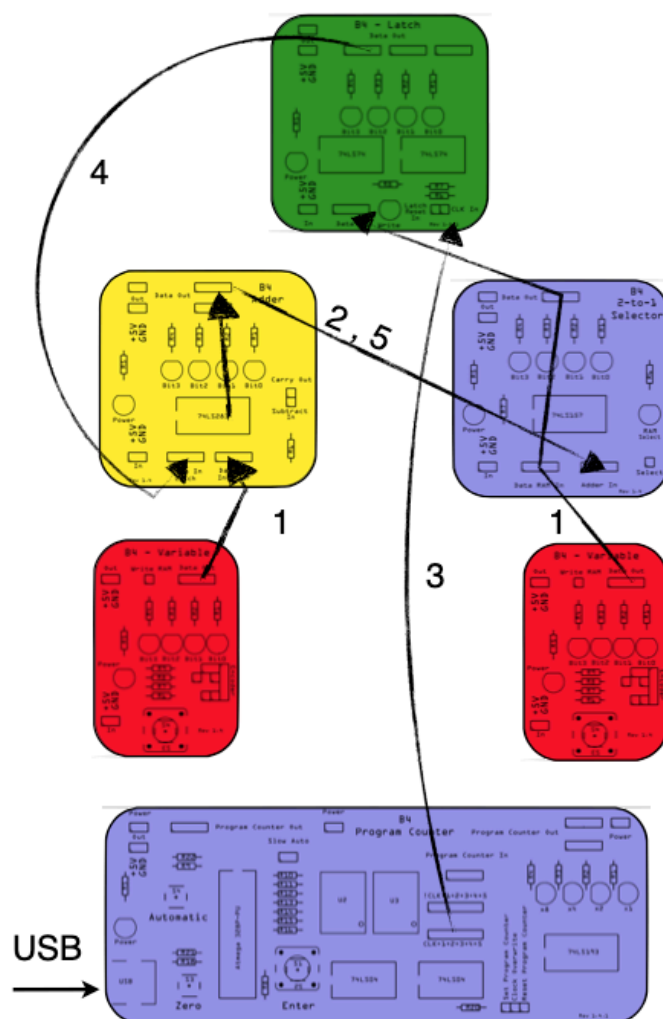
Setup of Experiment 7b

The following two diagrams illustrate the data flow during steps 1-5 and steps 6-9. Note how the 2-to-1 Selector changes the data flow and how the Latch always interrupts the infinite loop. Only when the CLK signal is sent from the Program Counter will the Latch briefly allow the data to flow from its input to its output.

Again, we try to compute $1+1+4$.

We start by resetting the Latch to 0000 by briefly connecting a control wire from Latch-reset to GND. Then, remove the control wire. All LEDs on the Latch should be off.

| Step | What's happening | Type |
|------|---|--------------|
| 1 | Set both variables on B0001. The value of the left Variable goes directly to the adder. The value of the right Variable travels via the 2-to-1 Selector to the input of the Latch | Data flow |
| 2 | Observe the output of the Adder. It shows B0001. Why? Because the Latch is still at zero. The Adder adds $1 + 0 = 1$ | Data flow |
| 3 | Press the ENTER button on the Program Counter. This sends a CLK signal to the Latch. The Latch stores the B0001 from the 2-to-1 Selector | Control flow |
| 4 | The Latch gives B0001 to the Adder. | Data flow |
| 5 | The Adder adds the 1 from the Latch to the 1 from the left Variable and displays B0010. Hooray, we have computed $1+1=2$. | Data flow |



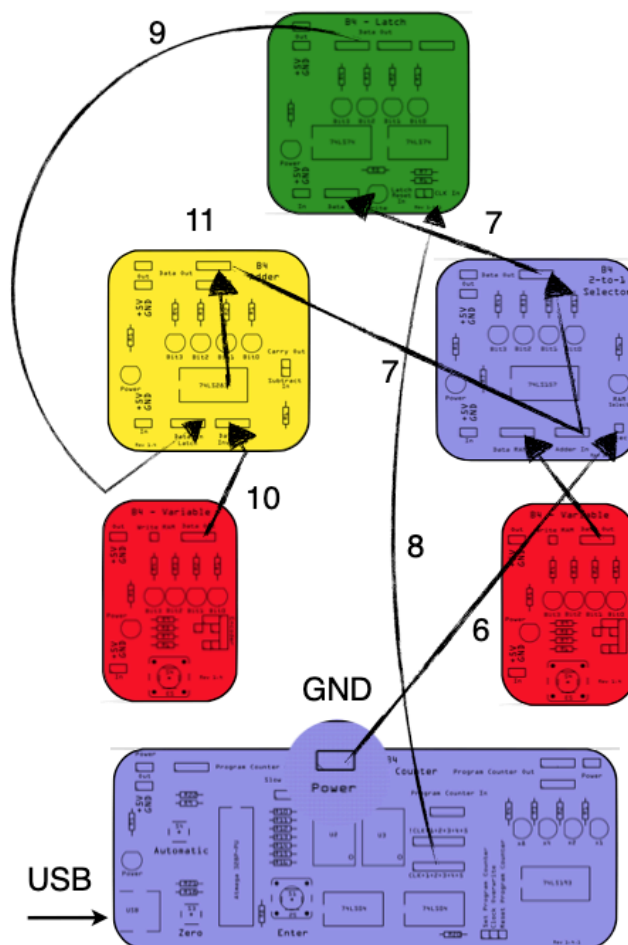
Data and control flow during steps 1-5

Next, we need the Latch to remember this result of the addition.

| Step | What's happening | Type |
|------|---|--------------|
| 6 | Move the 2-1-Selector's control wire from +5V to GND. The 2-1-Selector is now interested in any data from the Adder ... | Control flow |
| 7 | ... that is B0010, which the 2-1-Selector provides to the input of the Latch. | Data flow |
| 8 | Click the Enter button on the Program Counter again. This generates a CLK signal for the Latch. The Latch then remembers 0010. The intermediate result was saved. | Control flow |
| 9 | Then the intermediate result flows to the left input of the Adder | Data flow |

Finally, we add the third number, C, which is 4, to D. $2+4=6$

| Step | What's happening | Type |
|------|--|-----------|
| 10 | Set the left Variable to B0100 (4 in decimal). This value flows to the right input of the Adder. | Data flow |
| 11 | The Adder displays the final result, which is B0110 (6 in decimal) | Data flow |



Data and control flow during steps 6-11

Mission accomplished!

It is interesting to note that after step 8, the Adder displays a 3, which is 1 (from the left Variable) plus 2, which is D. But that doesn't concern us because we set the left Variable to a new value in step 10.

The steps you just run are a **program**. Would you have thought that a simple addition task of three numbers causes so much work for a computer? Eleven steps in exactly the correct order were necessary to calculate $1 + 1 + 4$.

As we will see later, the Latch remembers the output of the Adder at every clock cycle. The programs that we will write can load data from RAM, write Data back and control the flow of data with the 2-to-1 Selector. Adding and latching will be done automatically. You can compare this to your body. Your cells also work automatically - your brain does not need to instruct them. That's a bit of a generalisation, but you get the picture.

Controlling the flow of data is at the heart of every computer. In this experiment, we have learned that we can add more than two numbers by using the 2-to-1 Selector and the Latch. The 2-to-1 Selector helps us to switch the output of the Adder into the next addition cycle, whilst the Latch remembers the intermediate result and prevents an infinite loop. This works not only for three numbers, but also for five, six, or any number of numbers we want to add.

For example the addition of 4 numbers can be broken down into three additions of two numbers each:

$A+B+C+D$

Step 1: $A+B=E$

Step 2: $E+C=F$

Step 3: $F+D=R$

Any addition of n numbers can be broken into $n-1$ additions of two numbers.

We are making good progress towards a real computer. In the next experiment, we will learn how to store data and program information so that we no longer have to set data with Variables and no longer have to change the wiring of the control signals during calculation. The solution to both problems is, surprisingly, more memory.

Experiment 8: Let's Make a Real Computer

Modules Required: All modules except for the Automatic Programmer

During the first seven experiments we have explored the parts that make a computer. We have learned about the Program Counter, the Adder, the Inverter, the Latch the Data RAM and the 2-to-1 Selector. As we experimented with them, we learned about binary numbers, the CLK signal, how to store data in the Data RAM, how to avoid an infinite loop, and how to give data direction through the 2-to-1 Selector.

B4 Design

Did you notice that the only difference between adding and subtracting two numbers is the activation of the Inverter module? Connecting just one wire to +5V activated it, and as a result, the Adder didn't add, but subtract. Similarly, storing data in the Latch required the CLK signal. Storing data in the Data RAM needed the press of a button from the Variable, which produced also a signal. In the B4, a binary 1 corresponds to about +5V and a binary 0 to about 0V. If we could store some of these 1's and 0's and connect them to the Inverter, Latch and Data RAM, then we could control our computer. And this is where the Program RAM comes into play. Like the Data RAM, the Program RAM can store 16 sets of 4 bit of information.

If we put the Data RAM and the Program RAM side by side and label their respective bits, we can draw a table like the one below.

| | Data RAM | | | | Program RAM | | | |
|---------|----------|---|---|---|-------------|---|---|---|
| bit # | 3 | 2 | 1 | 0 | A | B | C | D |
| Step 15 | | | | | | | | |
| Step 14 | | | | | | | | |
| Step 13 | | | | | | | | |
| Step 12 | | | | | | | | |
| Step 11 | | | | | | | | |
| Step 10 | | | | | | | | |
| Step 9 | | | | | | | | |
| Step 8 | | | | | | | | |
| Step 7 | | | | | | | | |
| Step 6 | | | | | | | | |
| Step 5 | | | | | | | | |
| Step 4 | | | | | | | | |
| Step 3 | | | | | | | | |
| Step 2 | | | | | 0 | 1 | 0 | 0 |
| Step 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| Step 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

In the Data RAM we store the data that we want the B4 to work with and in the Program RAM we store the commands required to switch parts of the B4 on and off to compute. Does this look like a code? Yes, absolutely. Now you know where the term ‘coding’ comes from.

We need to decide what each bit in the Program RAM is supposed to do.

Let’s decide that:

- Bit 3 activates the Inverter when 1. The Adder then subtracts, instead of adding. Let’s therefore call it SUB (for subtract)
- Bit 2 activates the writing into the Data RAM when 1. Let’s call it WRT (for write)
- Bit 1 points the 2-to-1 Selector to the Data RAM when 1, or to the Adder when 0. We call it SEL (for select)
- Bit 0 is reserved for you, the students, to make extensions to the B4. It is not important at the moment.

We call these codes, instruction codes or operation codes, or, in short, **opcodes**. They instruct circuitry to perform certain operations. The following table summarises the B4 opcodes:

| | Opcodes | | | |
|---------------------------|-------------|--|--|--------------|
| Output Program RAM Module | A | B | C | D |
| Name | Subtract | Write to Data RAM | Select | user defined |
| Abbreviation | SUB | WRT | SEL | USR |
| Function when 1 | Subtracting | Write content of Latch to Data RAM | Connect input of Latch to output of Data RAM | no function |
| Function when 0 | Adding | Don’t write content of Latch to Data RAM | Connect input of Latch to output of Adder | no function |

You might wonder why there is no opcode to activate the Latch. The Latch operates on the CLK signal, which is automatically generated when the Program Counter ticks.

Do we need an opcode to activate the Adder? No. The Adder will always add whatever data is provided to it. All we need to do is to decide whether we want to use the Adder’s output or not. The SEL signal performs this job.

With the opcodes settled, we can re-draw our table as follows. We also give it a little description field for each program step to write down in our own language what the B4 is expected to do.

Designing the Program

| | Data RAM | | | | Program RAM | | | | Description |
|---------|----------|---|---|---|-------------|-----|-----|-----|--|
| Step # | 3 | 2 | 1 | 0 | SUB | WRT | SEL | USR | |
| Step 15 | | | | | | | | | |
| Step 14 | | | | | | | | | |
| Step 13 | | | | | | | | | |
| Step 12 | | | | | | | | | |
| Step 11 | | | | | | | | | |
| Step 10 | | | | | | | | | |
| Step 9 | | | | | | | | | |
| Step 8 | | | | | | | | | |
| Step 7 | | | | | | | | | |
| Step 6 | | | | | | | | | |
| Step 5 | | | | | | | | | |
| Step 4 | | | | | | | | | |
| Step 3 | | | | | | | | | |
| Step 2 | | | | | 0 | 1 | 0 | 0 | Store the result (0011) back into the Data RAM |
| Step 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Send 0010 to the Adder, which adds it to the 0001 stored in the Latch. |
| Step 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | Load 0001 from the Data RAM into the Latch. This is the first number for the adder |

Our program should calculate $1 + 2$.

In step 0 the B4 loads 0001 from the Data RAM and latches on to it. In Step 1, it loads the second data (0010), adds it and holds on to it on the output of the Adder. In Step 2, the B4 does two things. It first latches the result from the Adder and then writes it into the Data RAM.

Let's now put theory into practice by connecting all core components together to build this machine. We will then program it and finally run the program.

B4 Assembly

This will be our biggest build so far for which we require all 7 core components of the B4:

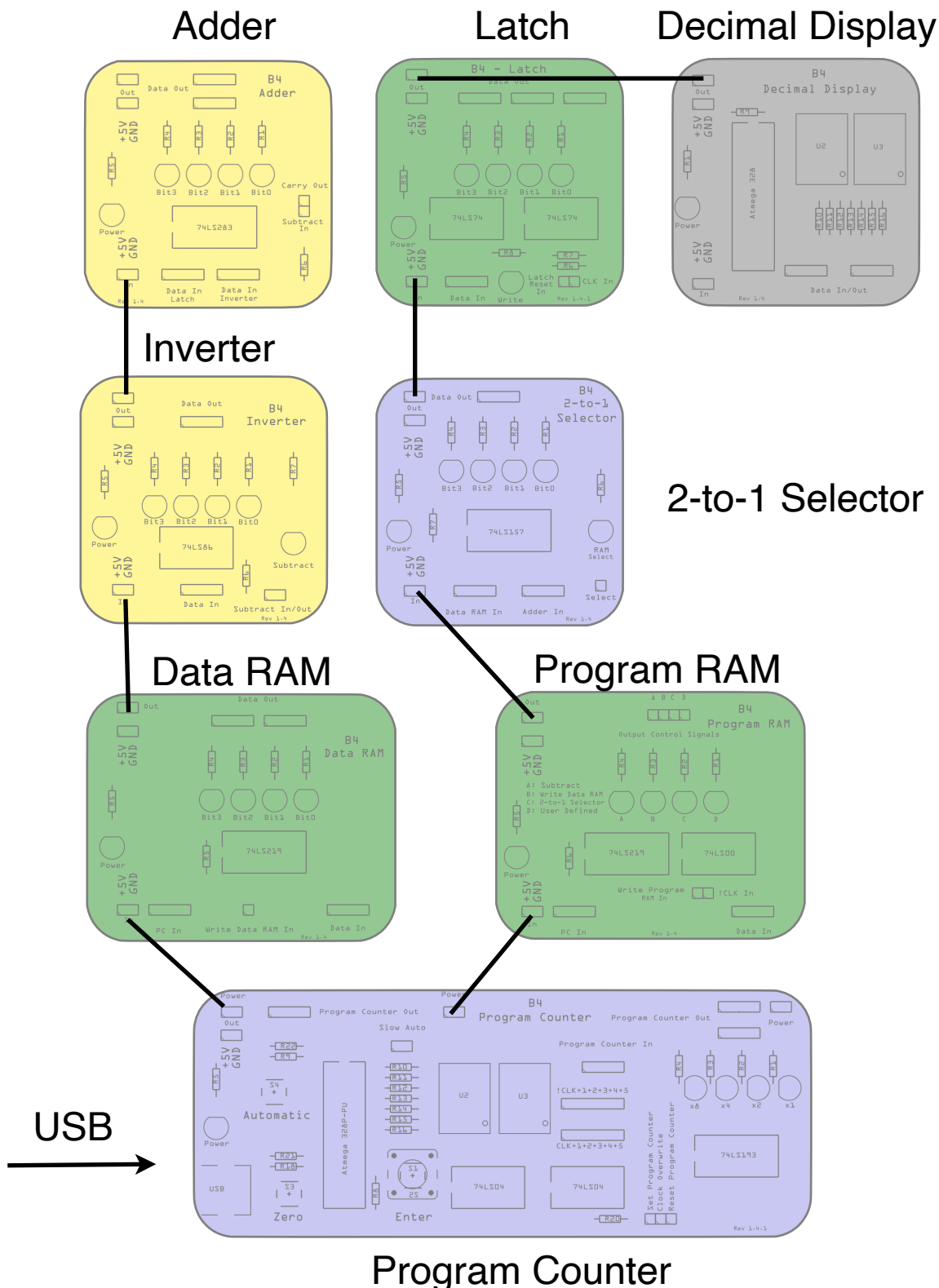
- 1x 2-Line-to-1-Line Selector
- 1x Adder
- 1x Data RAM
- 1x Inverter
- 1x Latch
- 1x Program Counter
- 1x Program RAM

We place them on the desk in front of us as shown on the next pages and then connect the wires as shown. You can tick the wires off one by one with a soft pencil on the paper of this handbook to ensure that you haven't forgotten one.



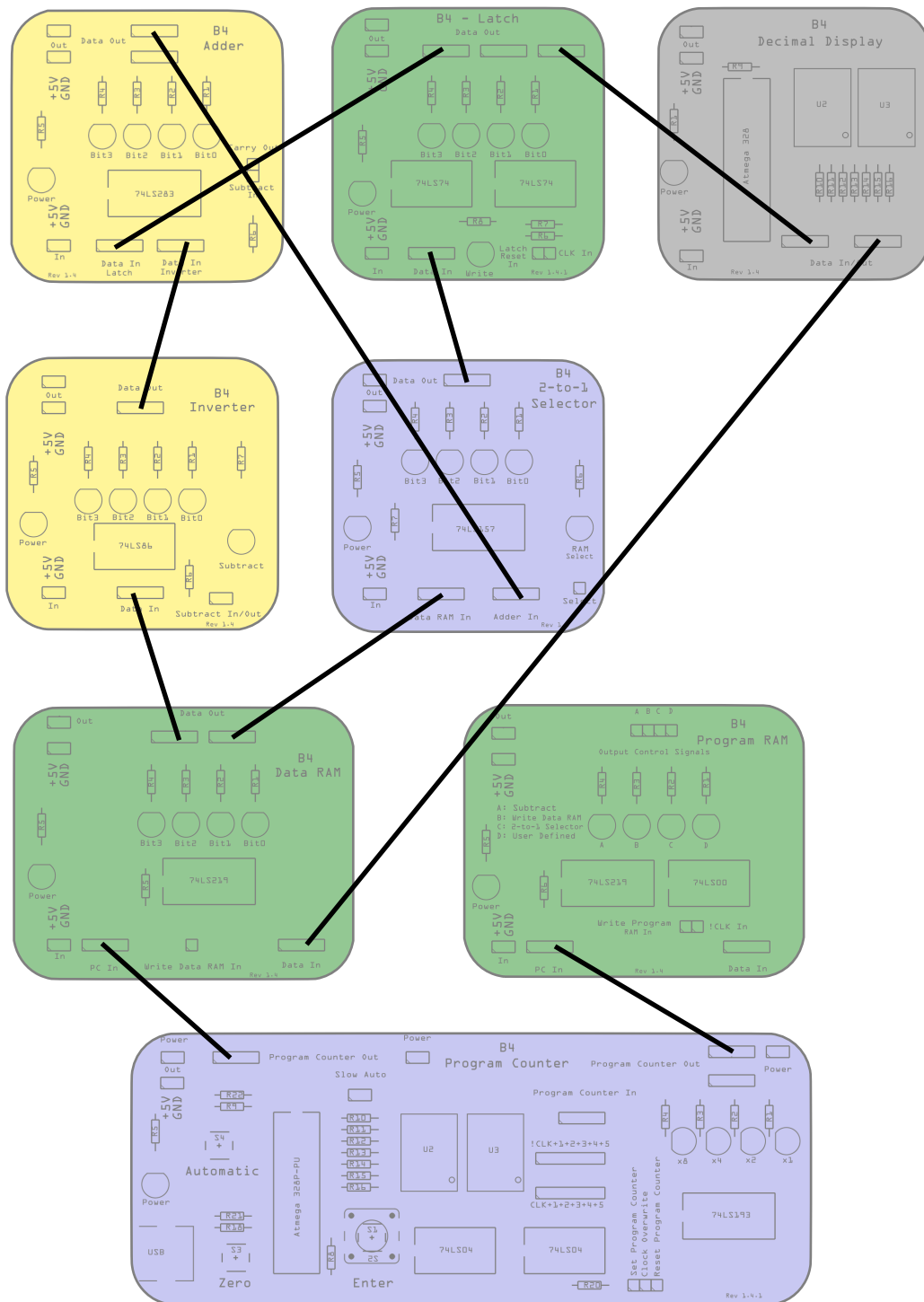
If you like, you can set this experiment up by placing the modules in the foam insert that came with the B4 box. We designed it specifically with experiments 8-12 in mind. This way the modules neatly stay in place.

We begin by connecting all the **power wires**, as shown in the following diagram:



Setup of Experiment 8: **Power Wiring only**

Then, we add the **data wires**:

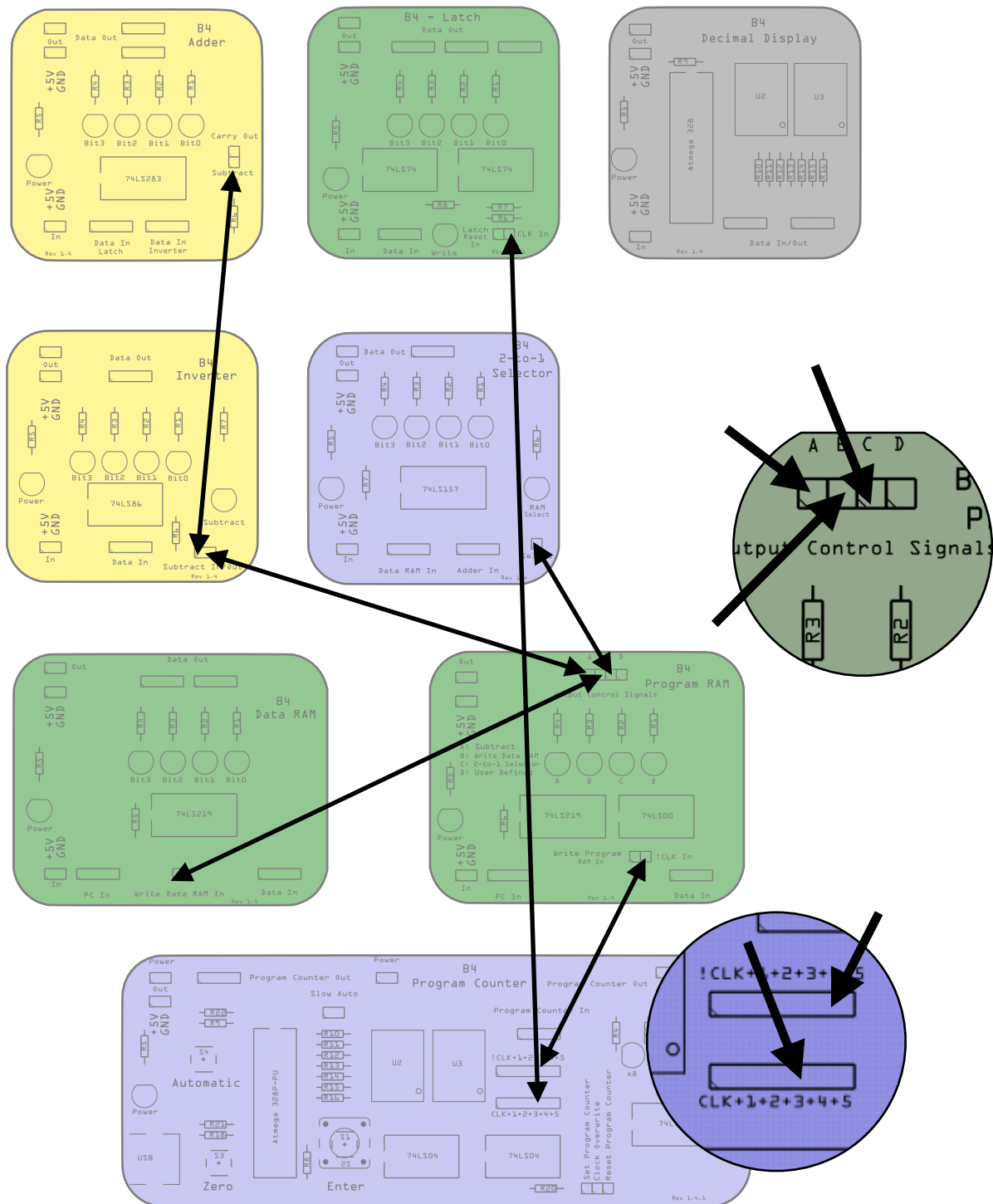


Setup of Experiment 8: Data Wiring only

And finally, we add the 1-pin **control wires**.

Take a closer look at the control wires on the Program Counter and on the Program RAM modules. To connect them to the correct pins is really important and we have therefore magnified the sections of the Program Counter and Program RAM in the figure below.

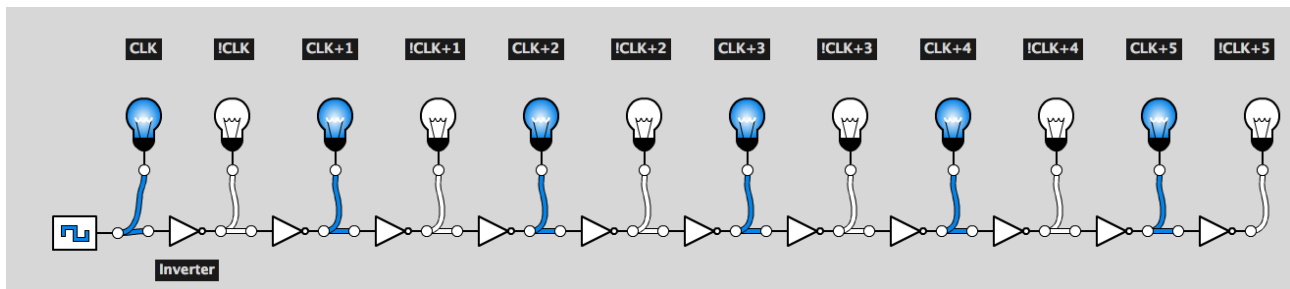
The Latch's CLK In pin is connected to CLK+3 and the Program RAM's !CLK In pin gets wired to !CLK+4.



Setup of Experiment 8:1-Pin Control Wiring only

The Program Counter has one row of pins dedicated to the CLK signals and another row of pins for the !CLK signals. They are labeled as CLK+1+2+3+4+5 and !CLK+1+2+3+4+5 respectively. Because some the modules in the B4 need input from other modules, before they can perform their function, we need to activate (or trigger) them in the proper sequence. For example, before we can store data in the Latch, it has to be made available by the Data RAM first. And before we can write the result of an arithmetic operation back into the Data RAM, it also has to be stored in the Latch first.

And this is where the B4's CLK and !CLK signals are required. When you press the Enter button on the Program Counter, the CLK signal will be generated, just as shown in experiment 1.



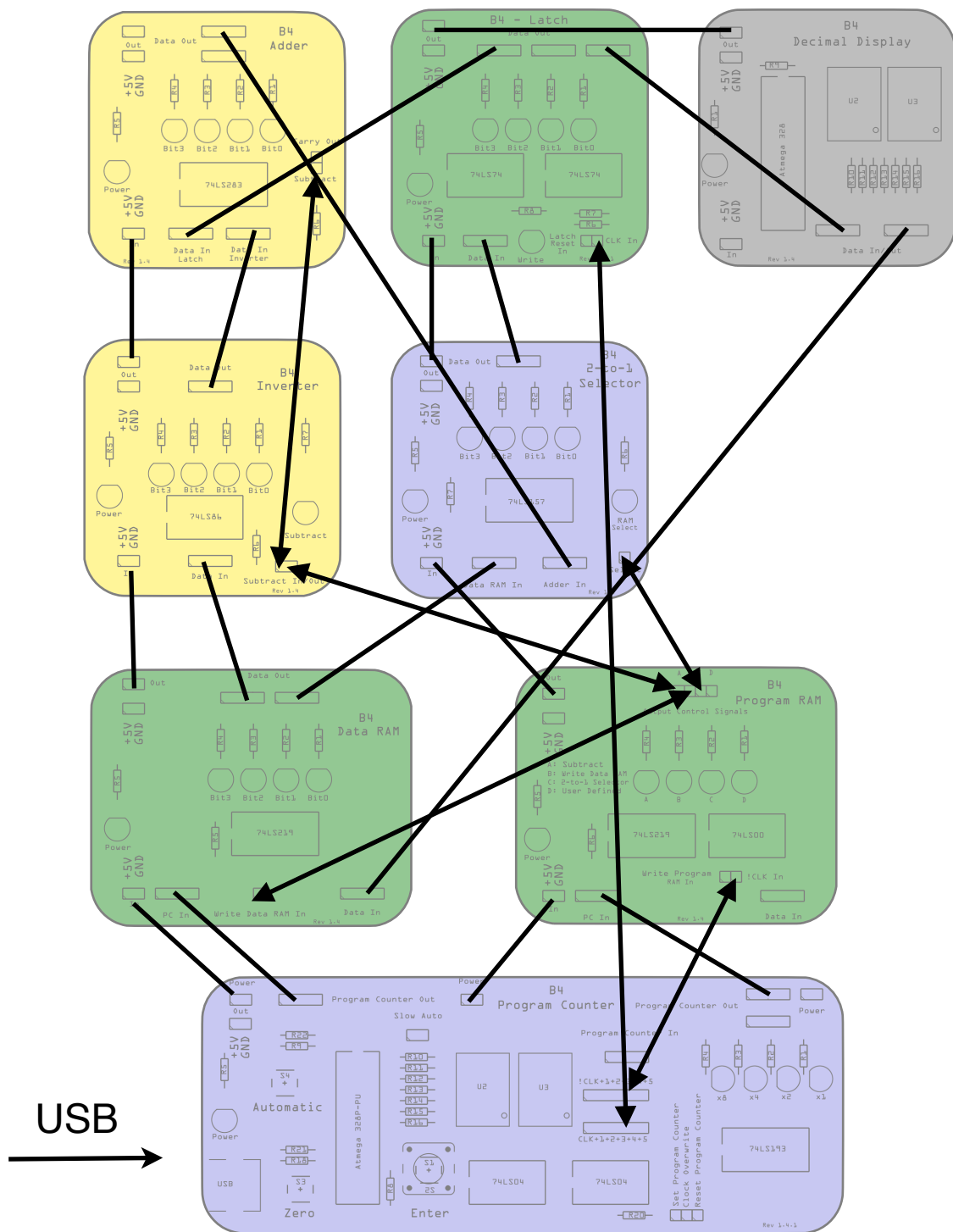
CLK and !CLK Delay Chain

To delay and invert a signal, we require a circuit. An electronic circuit that can perform this function is called an inverter. It inverts a 1 signal into a 0 signal and a 0 signal into 1.

Although inverters are very fast and operate nearly at light speed, they still need a little bit of time to do this - about 20 billionth of a second, which is 5 nano seconds, 5ns. This means that the !CLK signal changes 5ns after the CLK signal. The !CLK+5 signal occurs $11 \times 5\text{ns} = 55\text{ns}$ after the original CLK signal. With the CLK+1+2+3+4+5 and !CLK+1+2+3+4+5 we can finely tune the B4 so that it operates just in the right order. This involves a little bit of trial and error. **We have found that the B4 operates well with the Latch connected to CLK+3 and the Program RAM wired to !CLK+4.** We encourage you to experiment with these settings a little bit later, once we have programmed the B4.

Fun fact: The speed of light is approximately 300,000km per second. In 5ns, light travels about 1.5m.

Your B4's hardware is now complete and should look like in the figure below. Next, we will develop the software.



Completed Setup of Experiment 8

Experiment 9: Programming the B4

Required modules: Setup from experiment 8, 2x Variable

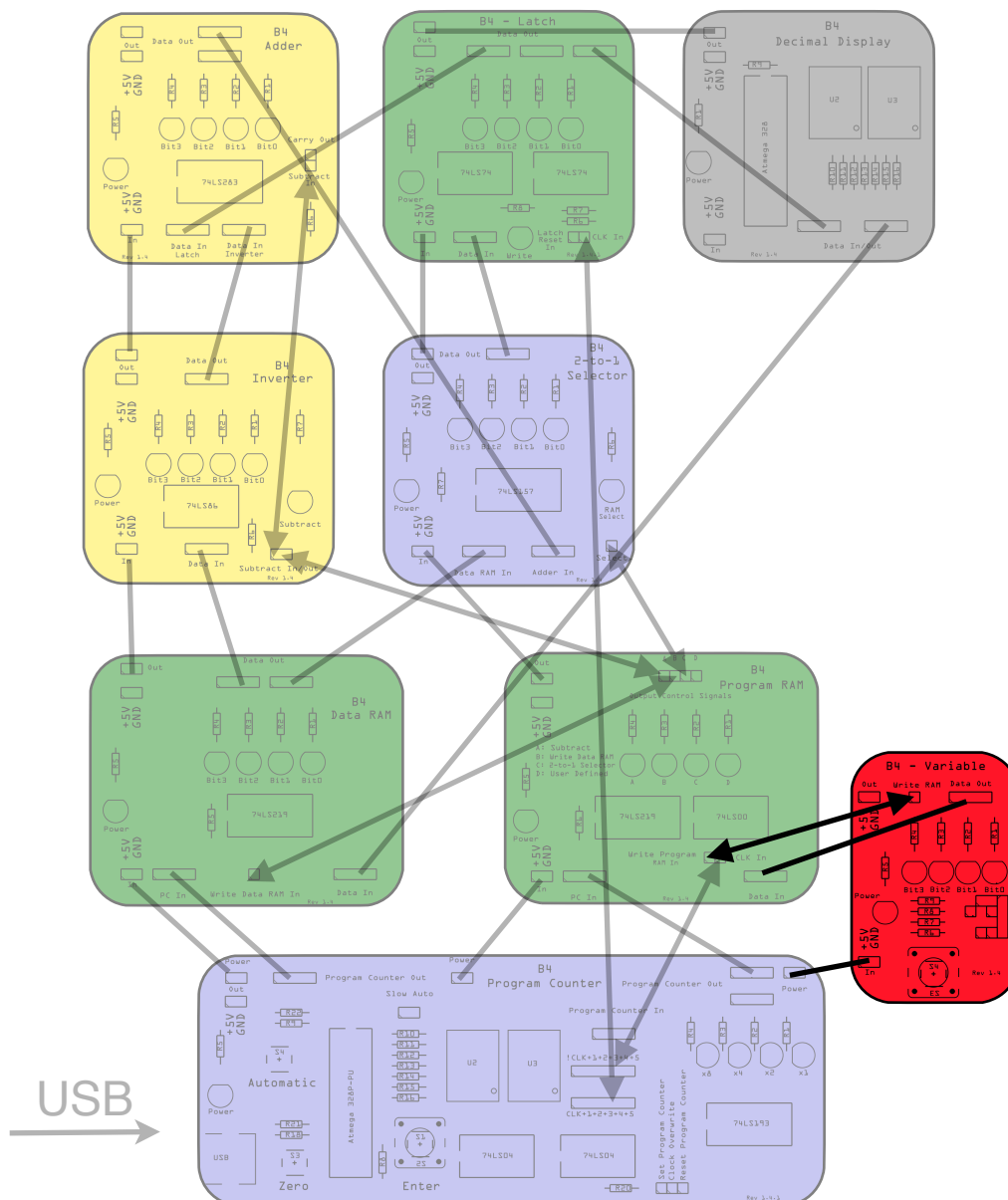
We will program the B4 in two steps:

1. Enter the program code into the Program RAM. These are the instructions that tell the B4 what it should do with the data.
2. Enter the data into the Data RAM.

For both, we require a Variable.

1. Programming the Program RAM

Connect the Variable Module to the Program RAM as shown in the following figure. The Variable draws power from any available 2 pin power connector. Its output is connected to the Data In Pins on the Program RAM and the Write RAM Pin of the Variable connects to the Write Program RAM In Pin on the Program RAM.



Setup of Experiment 9: Programming the Program RAM.

Set the Program Counter to 0000. That's step 0 of our program.

Step 0:

We now enter our first command. Set the Variable to 0010 and then press the button on the Variable. Congratulations, you have just programmed the B4 to Load data from the Data RAM into the Latch.

Step 1:

For our next command, we need to progress the program counter to 0001, which is step 1. Then, on the Variable, enter 0000 and press the button on the Variable. Congratulations, you have just programmed the B4 to send a second set of data to the Adder. The adder will add to the data already stored in the Latch.

Step 2:

Progress the Program Counter to 0010 and enter 0100 into the Variable. Click its button to store this command into the Program RAM. The instruction is to store the result of the addition back into the Data RAM.

Steps 3 to 15:

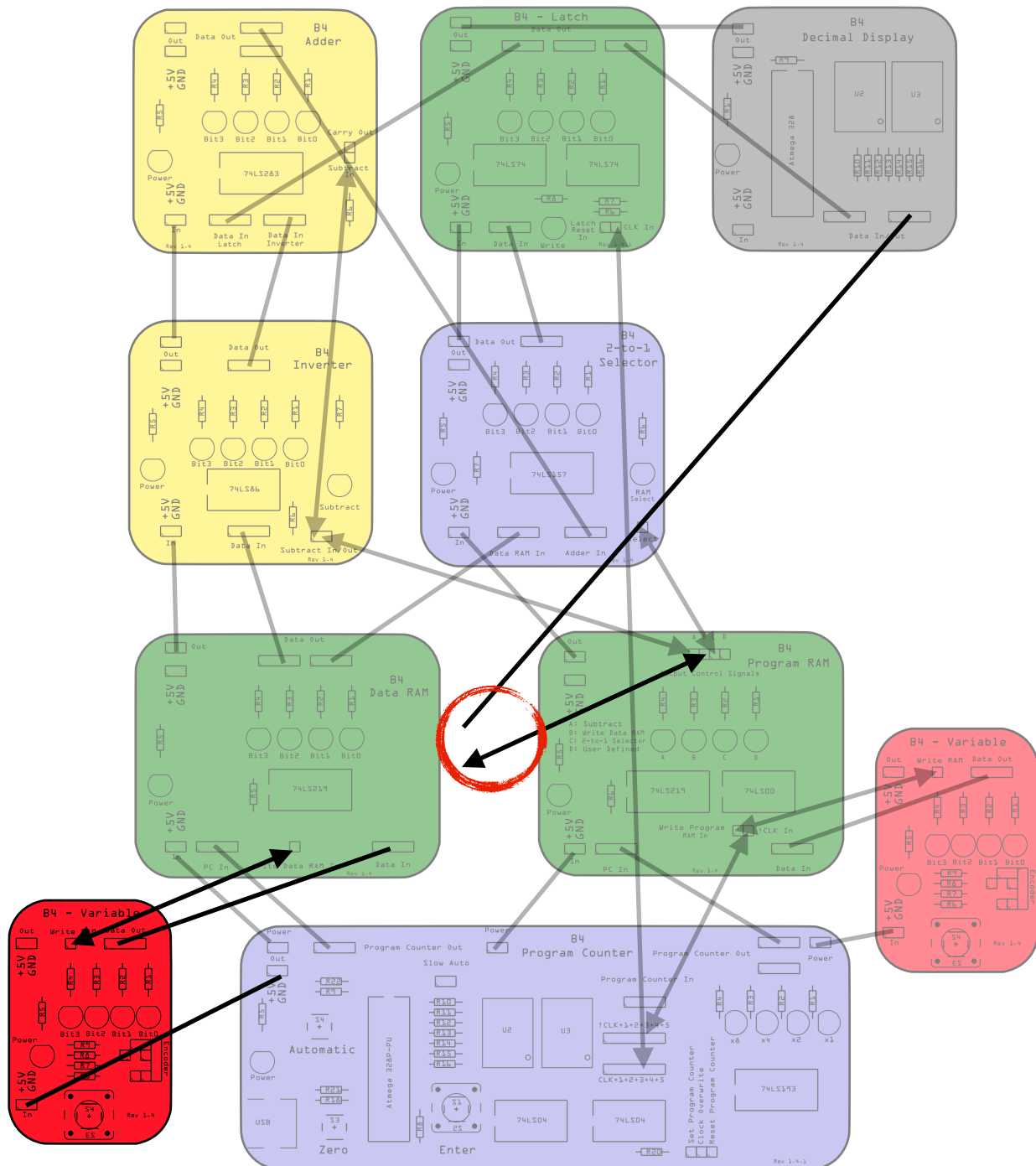
We enter neutral commands into the Program RAM. In the B4, the neutral command code is 0000. Technically it loads data from the RAM and adds it to whatever is in the Latch, but since we will set the corresponding Data RAM to 0000 shortly, the effect of the activity is neutral. So let's set the Program Counter to 0011, set the Variable to 0000 and press the button on the Variable. Repeat this for Steps 4 to 15. In the end, your Program RAM should look like in the table below. You can check this by pressing the Enter button on the Program Counter repeatedly. If everything is ok, move on the next section.

| | Data RAM | | | | Program RAM | | | | Description |
|------------|----------|---|---|---|-------------|-----|-----|-----|--|
| Step # | 3 | 2 | 1 | 0 | SUB | WRT | SEL | USR | |
| Steps 3-15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | do nothing |
| Step 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Store the result back into the Data RAM |
| Step 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Add 0010 to the contents of the Latch. |
| Step 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | Load 0001 from the Data RAM into the Latch. This is the first number for the Adder |

Program for Experiment 9

2. Programming the Data RAM

To program the Data RAM we follow a similar approach to programming the Program RAM. Connect the second Variable to the Data RAM as shown in the following image. **To connect the Variable, you will need to temporarily disconnect the 4 pin wire from the Dot Matrix Display to the Data RAM, and also the Write Data In wire from the Program RAM.** We will reconnect them again once we have programmed the Data RAM.



Setup of Experiment 9: Setting the Data RAM

We now set the data with the newly added Variable.

Set the Program Counter to 0000. That's step 0 of our program.

Step 0:

We enter our first data. Set the Variable to 0001 and then press the button on the Variable.

Step 1:

For our next data, we progress the Program Counter to 0001, which is step 1.

Then, on the Variable, enter 0010 and then press the button on the Variable.

Step 2:

Progress the Program Counter to 0010 and enter 0000 into the Variable. Click its button to store this data into the Data RAM. We store 0000 here, as our program will store the result of the addition into the memory. This isn't really required, but it will make debugging easier if something doesn't work as expected.

Steps 3 to 15:

We clear the remaining Data RAM. Set the Program Counter to 0011, set the Variable to 0000 and press the button on the Variable. Repeat this for Steps 4 to 15. In the end, your Data RAM should look like in the table above. You can check this by pressing the Enter button on the Program Counter repeatedly. If everything is ok, move on.

Finally, remove the Variable, and reconnect the two parked wires. That's the data wire from the Display to the Data RAM and the control wire from the Program RAM's Pin B to the Write Data RAM In on the Data RAM module. B4 is now ready to compute.

3. Executing the Program

In the previous two steps, we have entered code and data into the B4's RAM modules. Now it is time to run the program and observe the B4's operation.

Set the Program Counter to zero by pressing its *Zero* button. Press its Enter button twice and you will see the result of the addition of $1+2$ on the display, which is 3. If the output is different, you may have made a mistake with the wiring or programming. Check your setup with the wiring diagrams and check your code against the program table.

The next table shows the output your B4 produces at every individual step if everything works correctly:

Because we have connected the display to the Latch and because the Latch receives the CLK signal always at the beginning of a program step, the display output always lags by a step. Alternatively, you can connect the Display to the output of the Adder to read the results of the additions and subtraction directly.

| Program Step | Visual | Module | Output of the module |
|--------------|--------|-------------|--|
| 0 | | Data RAM | B0001 |
| | | Program RAM | B0010 |
| | | Selector | B0001 |
| | | Adder | Numbers that depend on which value the Latch had. Not important. |
| | | Latch | |
| | | Display | |
| 1 | | Data RAM | B0010 |
| | | Program RAM | B0000 |
| | | Selector | B0011 |
| | | Adder | B0011 |
| | | Latch | B0001 |
| | | Display | 1 |
| 2 | | Data RAM | B0011 |
| | | Program RAM | B0100 |
| | | Selector | B0110 |
| | | Adder | B0110 |
| | | Latch | B0011 |
| | | Display | 3 |

Outputs of the modules of Experiment 9

Experiment 9a: Adding three numbers

We started our quest with the aim to add three numbers. In the previous experiment, we added two numbers. We will now see how we simply extend our code to let B4 add a third number.

We want B4 to compute $1+2+4$

Designing the Program

In experiment 9, we already calculated $1+2$, so we will just need to extend this program with an additional addition step. We leave steps 0 and 1 unchanged. Step 2 (writing data into the Data RAM) becomes Step 3 and we need to enter a new addition Step 2. The complete new program is listed in the table below. Follow the instructions on how to program the Program RAM and Data RAM modules from the previous experiment, including connecting and disconnecting wires to enter the new program and data into the RAM modules for this experiment.

| | Data RAM | | | | Program RAM | | | | Description |
|------------|----------|---|---|---|-------------|-----|-----|-----|--|
| Step # | 3 | 2 | 1 | 0 | SUB | WRT | SEL | USR | |
| Steps 4-15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | do nothing |
| Step 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Store the result back into the Data RAM |
| Step 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Add 0100 (4) to the contents of the Latch. |
| Step 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Add 0010 (2) to the contents of the Latch. |
| Step 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | Load 0001 (1) from the Data RAM into the Latch. This is the first number for the Adder |

Program for Experiment 9a

Running the Program

Reset the program Counter to 0 and press its Enter button three times. The Display then shows the result of the addition of 1+2+4, which is 7. The following table lists the detailed outputs of the various modules to help you trace the function of the program.

| Program Step | Visual | Module | Output of the module |
|--------------|---|-------------|--|
| 0 |  | Data RAM | B0001 |
| | | Program RAM | B0010 |
| | | Selector | B0001 |
| | | Adder | Numbers that depend on which value the Latch had. Not important. |
| | | Latch | |
| | | Display | |
| 1 |  | Data RAM | B0010 |
| | | Program RAM | B0000 |
| | | Selector | B0011 |
| | | Adder | B0011 |
| | | Latch | B0001 |
| | | Display | 1 |

Outputs of the modules of Experiment 9a, Steps 1-2

| Program Step | Visual | Module | Output of the module |
|--------------|--------|-------------|----------------------|
| 2 | | Data RAM | B0100 |
| | | Program RAM | B0000 |
| | | Selector | B0111 |
| | | Adder | B0111 |
| | | Latch | B0011 |
| | | Display | 3 |
| 3 | | Data RAM | B0111 |
| | | Program RAM | B0100 |
| | | Selector | B1110 |
| | | Adder | B1110 |
| | | Latch | B0111 |
| | | Display | 7 |

Outputs of the modules of Experiment 9a, Steps 2-3

Experiment 10: B4 Learns Subtraction

In this experiment, we use the same setup from the previous experiment. We now want to program the B4 to calculate the result of $3+5-2$ and store the result back into memory.

Designing the Program

We remember that in the previous experiment we had already added three numbers together. We just have to change our previous program. The complete program is listed in the following table.

The contents of the Data RAM shows the new numbers 3,5, and 2 that we want to work with. The Program RAM content is almost identical to the previous program. You notice that in the subtraction step (# 2) we only need to set the SUB bit to subtract the computer to subtract. The program RAM activates the inverter.

Follow the instructions on how to program the Program RAM and Data RAM modules from the previous experiment, including connecting and disconnecting wires to enter the new program and data into the RAM modules for this experiment.

| | Data RAM | | | | Program RAM | | | | Description |
|------------|----------|---|---|---|-------------|-----|-----|-----|--|
| Step # | 3 | 2 | 1 | 0 | SUB | WRT | SEL | USR | |
| Steps 4-15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | do nothing |
| Step 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Store the result back into the Data RAM |
| Step 2 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | Subtract 0010 (2) from the contents of the Latch by activating the Inverter. |
| Step 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | Add 0101 (5) to the contents of the Latch. |
| Step 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Load 0011 (3) from the Data RAM into the Latch. This is the first number for the Adder |

Program for Experiment 10

Running the Program

Reset the program Counter to 0 and press its Enter button three times. The Display then shows the result of $3+5-2$. The following table lists the detailed outputs of the various modules to help you trace the function of the program.

| Program Step | Visual | Module | Output of the module |
|-----------------------------------|--------|-------------|--|
| 0 Loading the first number | | Data RAM | B0011 |
| | | Program RAM | B0010 |
| | | Selector | B0011 |
| | | Adder | B0011 |
| | | Latch | Numbers that depend on which value the Latch had. Not important. |
| | | Inverter | |
| | | Display | |
| 1 Adding the second number | | Data RAM | B0101 |
| | | Program RAM | B0000 |
| | | Selector | B1010 |
| | | Adder | B0101 |
| | | Latch | B1000 |
| | | Inverter | B0011 |
| | | Display | 3 |

Outputs of the modules of Experiment , Steps 0-1

| Program Step | Visual | Module | Output of the module |
|---|--------|-------------|----------------------|
| 2 Subtracting the third number | | Data RAM | B0010 |
| | | Program RAM | B1000 |
| | | Selector | B0110 |
| | | Adder | B1101 |
| | | Latch | B0110 |
| | | Inverter | B1001 |
| | | Display | 8 |
| 3 Storing the result in the Data RAM | | Data RAM | B0110 |
| | | Program RAM | B0100 |
| | | Selector | B1100 |
| | | Adder | B0110 |
| | | Latch | B1100 |
| | | Inverter | B0110 |
| | | Display | 6 |

Outputs of the modules of Experiment 10, Steps 2-3

Experiment 11: Automatic Programming

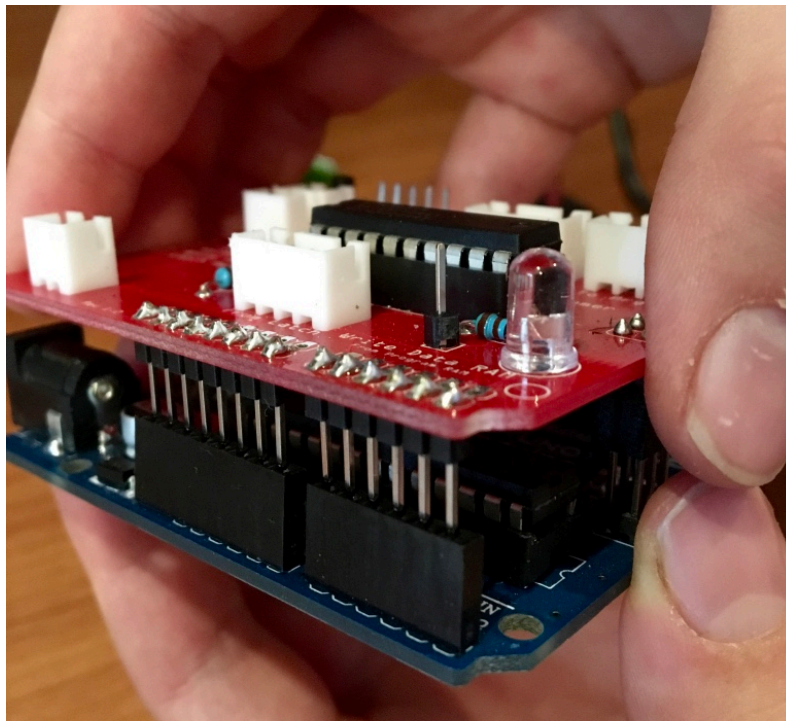
You will probably agree that entering data and program code into the B4 isn't very convenient. In the previous experiments, we have used the Variable modules to get an understanding of coding on the lowest possible level. To make the programming process more elegant, we will introduce the Automatic Programming (AP) shield. The AP can take full control of the B4 during the programming phase, as to avoid that, for example, data from the Latch gets written to the Data RAM accidentally. However, the AP will sit quietly in the background and not interfere with the B4 while the B4 runs a program. In a sense, the AP is a hacking device. All this has to be achieved without moving a single wire between programming mode and runtime mode.

To get it to work, you need:

- 1) An Arduino Uno or compatible and a USB cable that fits into the Arduino. You find both in the kit.
- 2) The B4 Automatic Programmer Shield
- 3) The setup from experiments 8, 9, 10.
- 4) A Laptop or PC with the Arduino IDE
- 5) The B4 Arduino Library, available from <http://www.digital-technologies.institute/downloads>

Step 1 Installing the Automatic Programmer

Plug the Automatic Programmer shield into the Arduino as shown in the following picture:



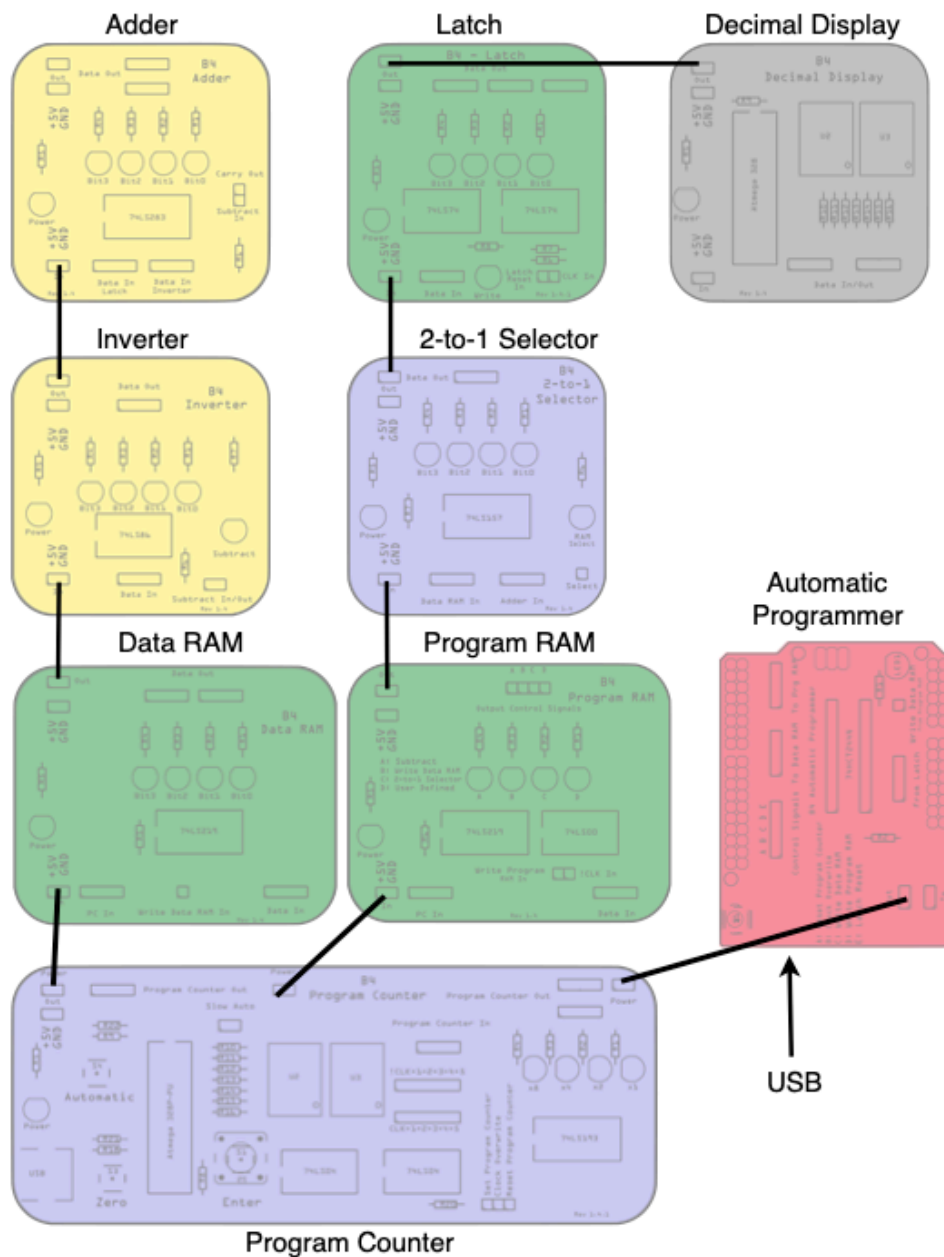
Installing the Automatic Programmer Shield on an Arduino

Step 2: Modules and their Wiring

Insert the Automatic Programmer into the setup of our circuit from experiment 8. A good place for the Automatic Programmer is on the right side of the Program RAM. Because of the complexity of the Setup of this experiment, we do it in multiple stages:

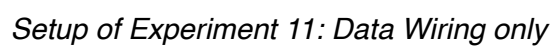
Stage 1: Modules and Power Wires:

First, let's arrange the modules as shown below. Then, connect the power wires as shown. Most of them would already be in place from the previous experiment, but you will need to run a wire to the Automatic Programmer



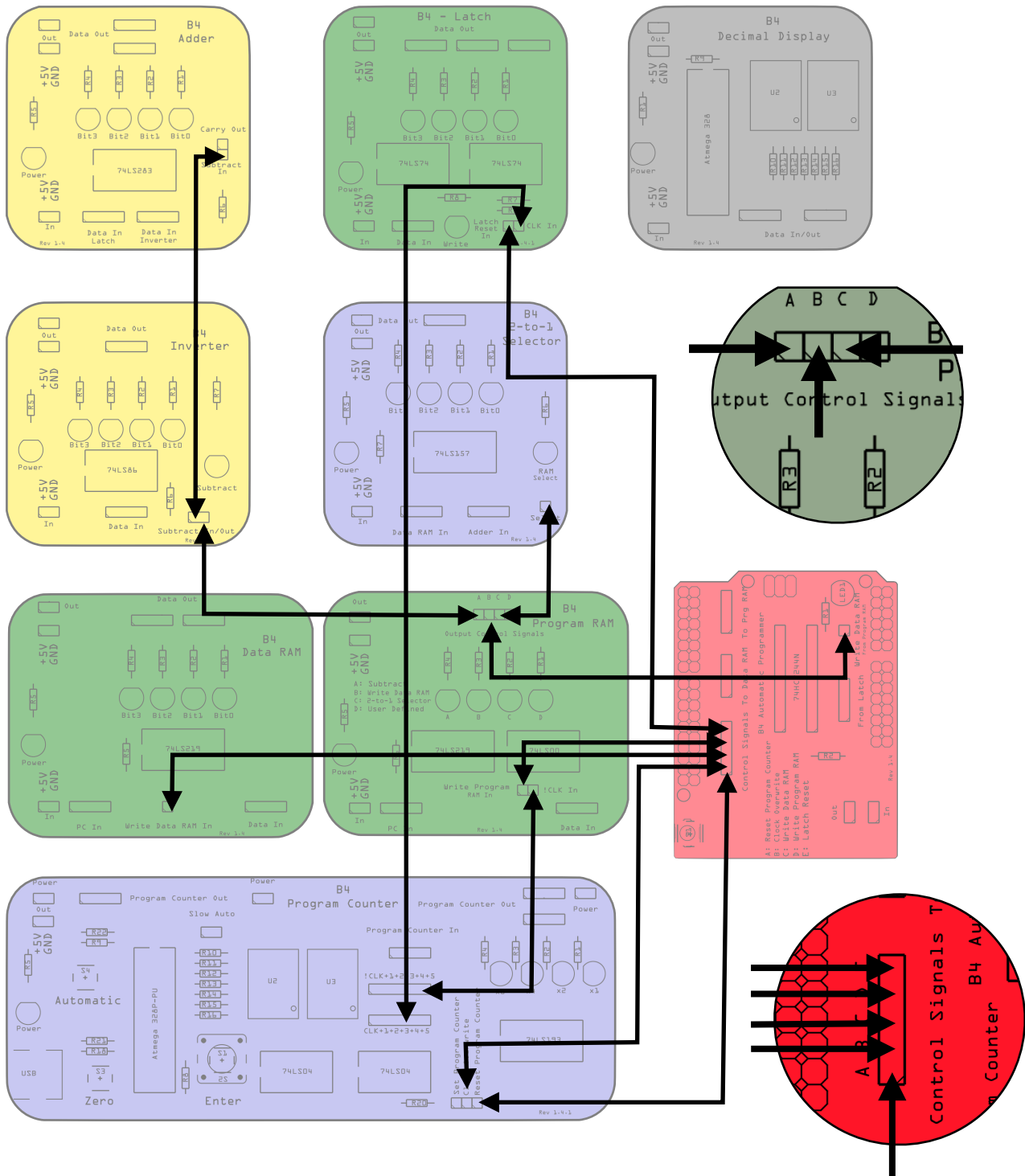
Setup of Experiment 11: Power Wiring only

Then, we connect the data wires as shown in the following Figure.



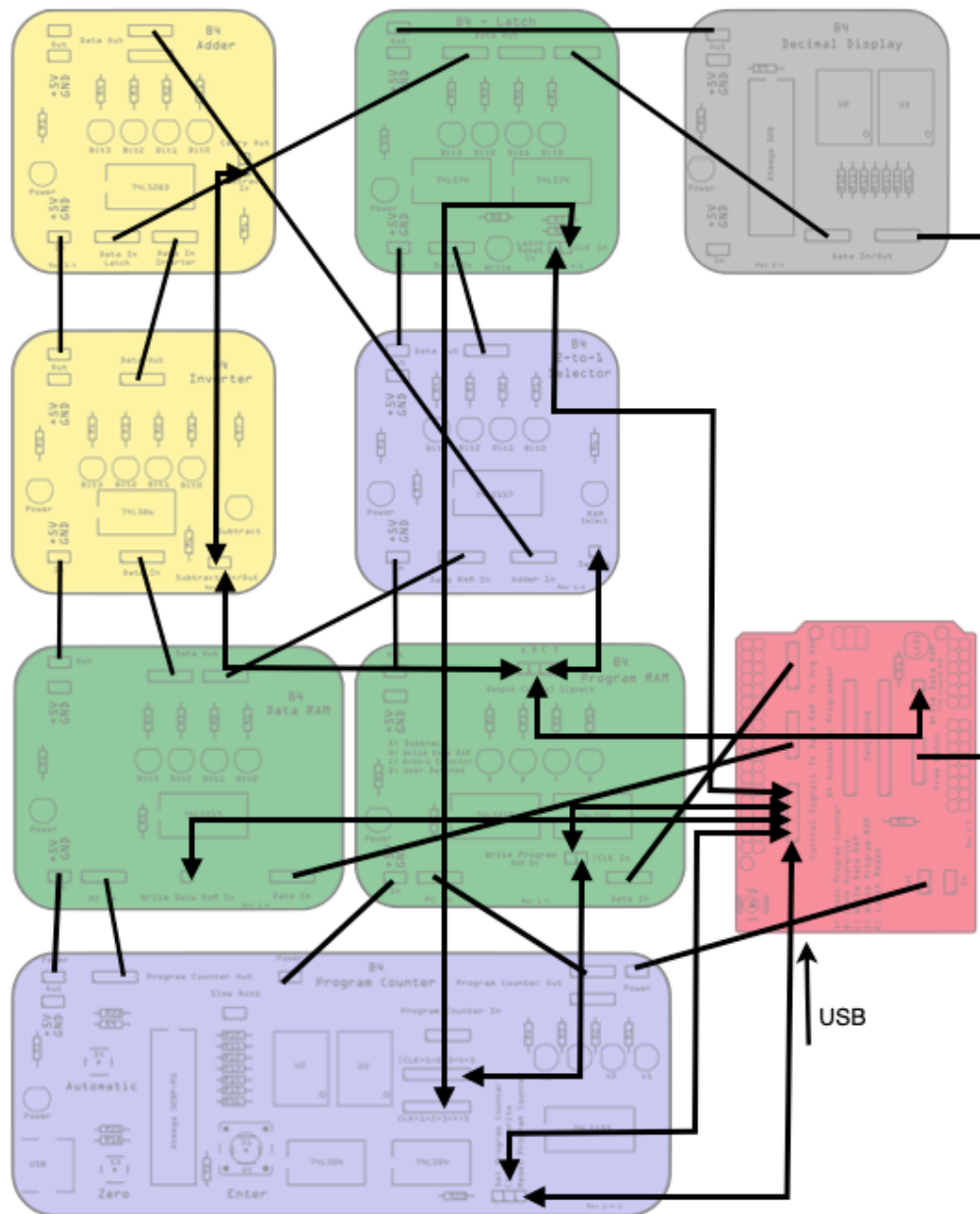
Stage 3: 1-Pin Control Wires:

Finally, we connect the one-pin control wires as shown in the following figure. The wiring of the Automatic Programmer can be a bit tricky. Each control wire has a name, such as Reset Program Counter. You will find a pin with the same name on the corresponding board.



Setup of Experiment 11:1-Pin Control Wiring only

Congratulations, we are done. The final setup looks like in the following Figure: **Make sure you connect the USB cable to the Automatic Programmer.**

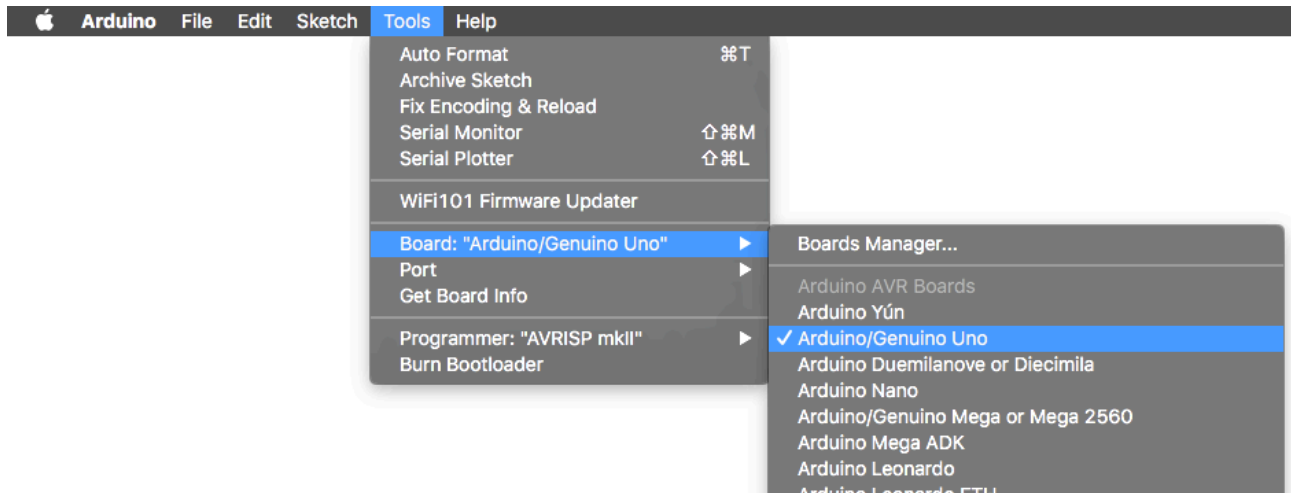


Setup of Experiment 11: All Wires

This completes our hardware setup. Let's continue with software.

Step 3: Installing and Configuring the Arduino IDE

Install the Arduino IDE on your laptop. If it is already installed, check the version number, which should be 1.6.8 or higher. If you need to download the Arduino IDE, head to <https://www.arduino.cc/en/Main/Software> and follow the download and installation instructions. Once the IDE is installed, go to the Tools Menu and select Arduino/Genuino Uno as Board.

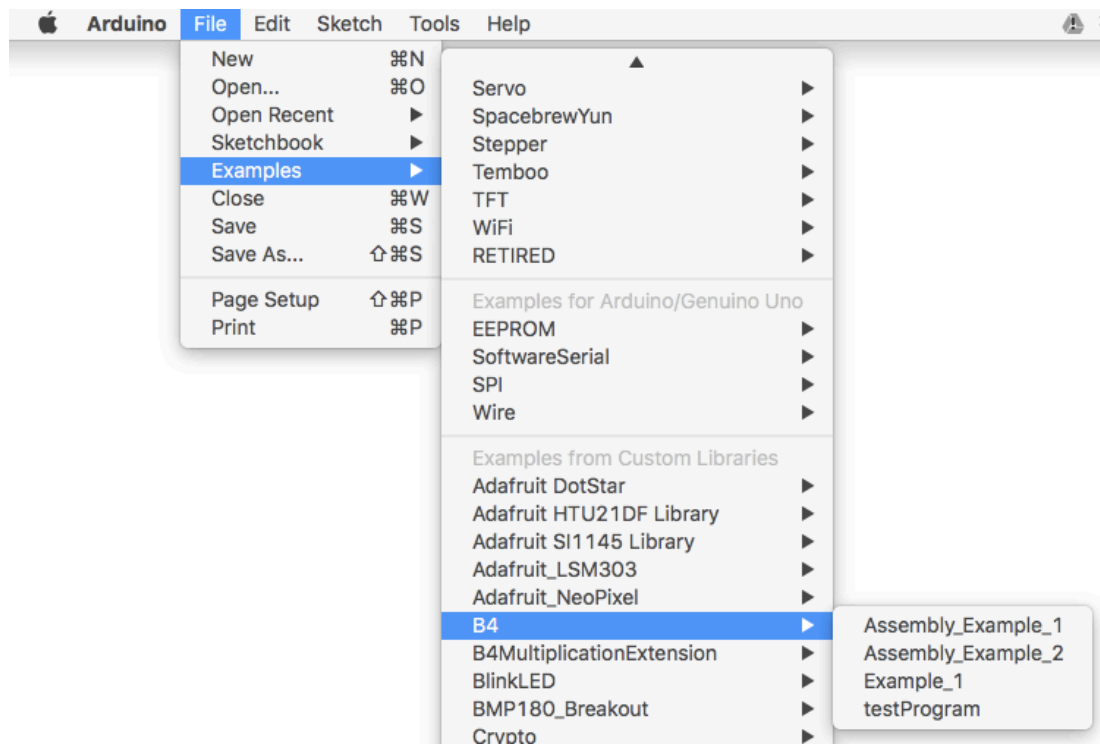


Arduino IDE: Selection of the Board

Then, go to the Ports submenu and select the USB port of your computer, which is connected to your Arduino.

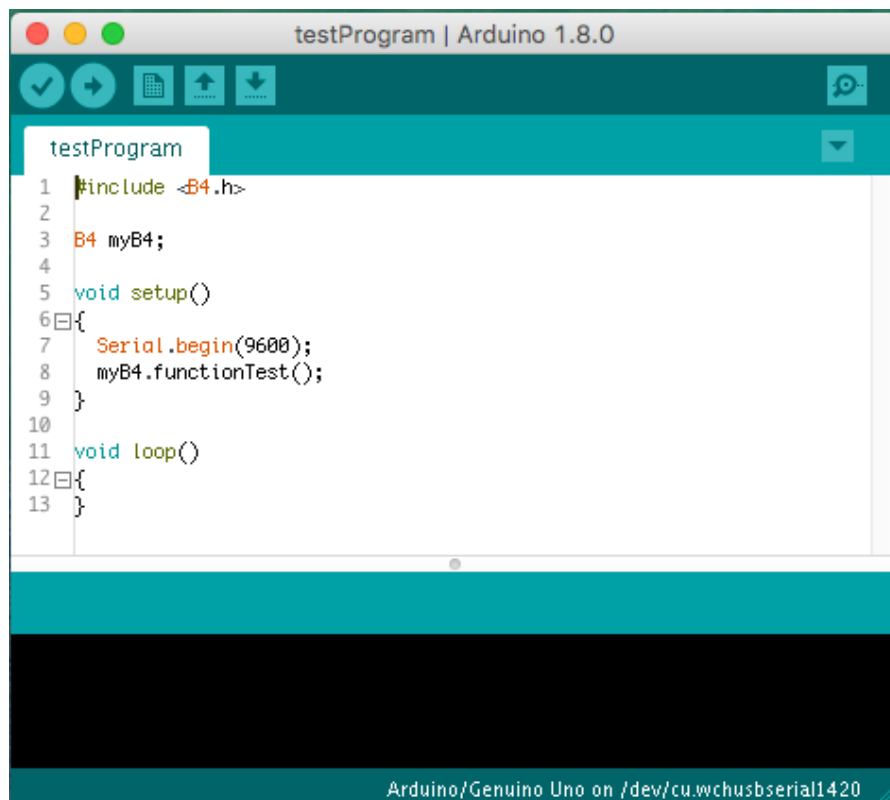
Step 4: Installing the B4 Arduino Library

Download the B4 library from <http://www.digital-technologies.institute/downloads>. Locate the folder, called B4-master and rename it to B4. Then, copy it into the Libraries folder in which your Arduino Sketches reside. On Windows and Macintosh machines, the default name of the folder is "Arduino/libraries" and is located in your Documents folder. Then, restart the Arduino IDE and go into the File menu. There, select Examples, and click on B4. This will look something like in the following figure:




Example Programs in the B4 Library

The Library already contains a number of programs. Let's run the testProgram first. Select it from the menu. This will open a new window which will look like in the following figure.



The B4 Test Program

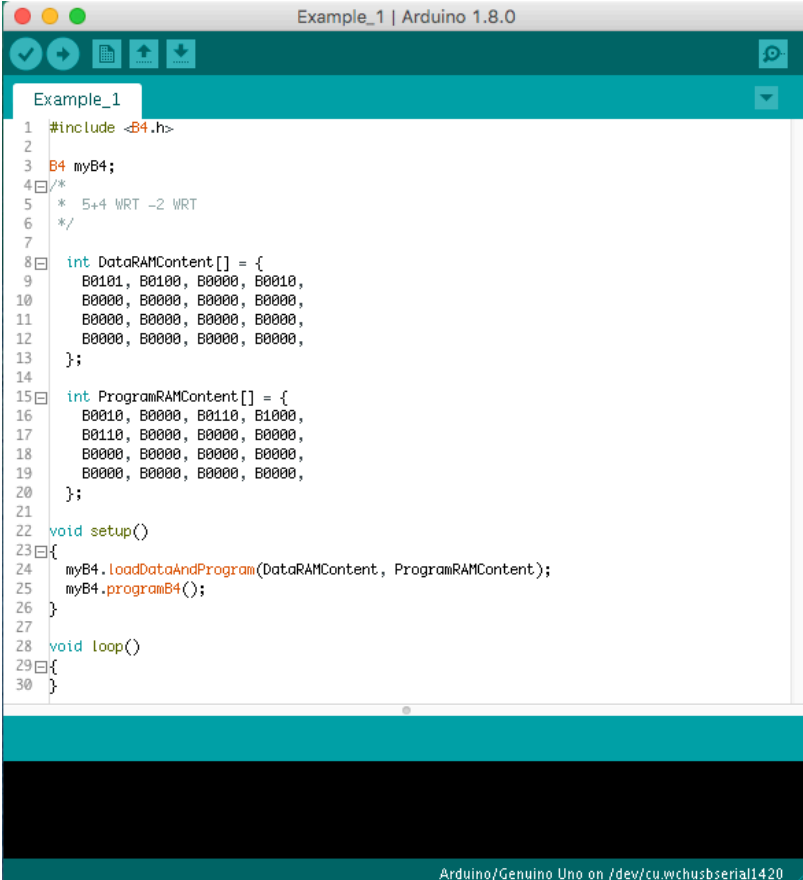
You just need to click on the second button from the top  to compile the code and upload it to the Arduino. Sit back and watch the LEDs of the B4 flashing, as the program gets uploaded.

The testProgram performs the following calculation $1+2-2+4-1+11-1+2-1$, which is hiding inside the *functionTest()* routine. This might appear a bit odd, but this calculation, including a number of write commands, requires all program steps and produces a result of 15, or binary 1111.

The program is designed so that we can check if the wiring is all ok. Click on the *Enter* button of the Program Counter repeatedly until it displays 15. If you see the Latch also showing 1111, then you can be sure that you have wired up the B4 correctly. If not, go back to the previous pages and double-check.


With the Automatic Programmer installed, we can load different programs really quickly. To run the programs already included in the Arduino library, all you need to do is to go into the library as explained above and select the program you want. We will see a little later how to design our own programs.

Now select the Example_1 program.



```
Example_1 | Arduino 1.8.0
1 #include <B4.h>
2
3 B4 myB4;
4 /*
5  * 5+4 WRT -2 WRT
6  */
7
8 int DataRAMContent[] = {
9     B0101, B0100, B0000, B0010,
10    B0000, B0000, B0000, B0000,
11    B0000, B0000, B0000, B0000,
12    B0000, B0000, B0000, B0000,
13 };
14
15 int ProgramRAMContent[] = {
16     B0010, B0000, B0110, B1000,
17     B0110, B0000, B0000, B0000,
18     B0000, B0000, B0000, B0000,
19     B0000, B0000, B0000, B0000,
20 };
21
22 void setup()
23 {
24     myB4.loadDataAndProgram(DataRAMContent, ProgramRAMContent);
25     myB4.programB4();
26 }
27
28 void loop()
29 {
30 }
```

Example_1 Program

To run this Program on the B4, click on . When the upload is complete, you can press the Enter button on the Program Counter.

Let's have a look at this program. Like the testProgram, it consists of a declaration of myB4, which is an instance of the B4 class. Then, we have a Data block, a Program block, a loadDataAndProgram() routine and finally a programB4() function. In the data and program blocks, the first 4 bit data is for program step 0 and the last one for program step 15. You have probably noticed that the 4 bit binary numbers all start with a B. This is the C-programming language way of knowing that it should deal with binary numbers. If we didn't declare that, then the compiler would assume that the binary number 0101 is actually one hundred and one. This would lead to the wrong results, as we want a 5, not a 101.

Let's explore the Data RAM Content first. There you can see the binary of the numbers 5, 4, 0, 2, and then 0s. We know that these are the numbers that we will perform some arithmetic operations on. Let's explore the Program RAM to find out what these are.

Take a look at ProgramRAMContent[]. The first element of the array is B0010, which means to load data into the Latch for further programming. The second element is B0000, which loads data into the Adder. The third element, B0110, stores data from the Latch into the Data RAM. Next, B1000 performs a subtraction and B0110 saves the data into program RAM and keeps it latched for further use. So, we now know that the program performs the following operations: 5+4, store the result (9), subtract 2, store the result (7).

```
int DataRAMContent[] = {
    B0101, B0100, B0000, B0010,
    B0000, B0000, B0000, B0000,
    B0000, B0000, B0000, B0000,
    B0000, B0000, B0000, B0000,
};

int ProgramRAMContent[] = {
    B0010, B0000, B0110, B1000,
    B0110, B0000, B0000, B0000,
    B0000, B0000, B0000, B0000,
    B0000, B0000, B0000, B0000,
};
```

Note that the positions of the elements in the arrays is important and that the indexes must match for the program to work on the correct data. For example, the DataRAMContent[0] is B0101, which is processed through ProgramRAMContent[0]=B0010 (LOAD).

Correspondingly, DataRAMContent[1] and ProgramRAMContent [1] form a data-processing pair and so on and so forth. Also note that we will fill those places of the program and data RAM that we don't need with zeros. This is important to get the machine in a defined state. Computer circuits can contain all sorts of random data when they get powered up. They need to be initialised.

In our familiar table format, the same program looks as follows:

| | Data RAM | | | | Program RAM | | | | Description |
|------------|----------|---|---|---|-------------|-----|-----|-----|--|
| bit # | 3 | 2 | 1 | 0 | SUB | WRT | SEL | USR | |
| Steps 5-15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | do nothing |
| Step 4 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | Store the result into the Data RAM |
| Step 3 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | Subtract 0010 from the contents of the Latch by activating the Inverter. |
| Step 2 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | Store the result back into the Data RAM |
| Step 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Add 0100 to the contents of the Latch. |
| Step 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | Load 0101 from the Data RAM into the Latch. This is the first number for the Adder |

Example_1 program in table form

After the declaration and initialisation of the Data and Program arrays, we call the *myB4.loadDataAndProgram()* function and pass along the *DataRAMContent* and *ProgramRAMContent* arrays which we want to be stored in the Data and Program RAM modules. As a last step, we call up the *myB4.programB4()* function. This is a collection of other functions that will then perform the necessary steps to program the B4. This includes the following functions:

```
void clearDataRAM();
void clearProgramRAM();
void setData();
void setProgram();
void reSetProgramCounter();
void clockCycle();
void writeRAM(int port);
void resetLatch();
```

The library shields these functions from the user to keep things simple. But you can explore the C++ code behind the entire B4 library by going into the *Arduino/libraries/B4* folder and have a look at the file *B4.cpp* with a simple text editor.

Experiment 12: Program Language Design

Now that we understand how a computer works internally with its data and opcodes we can begin to think of a higher-level language to program the B4. Computer scientists often speak about a higher-level language when it resembles less the computer-internal representation, and more the way humans like to think and talk about programs and data. Ideally we want our computer to understand something like $5+4-2$, and this is our goal. We start with a first step by trying to make our program more compact and easier to read and write. Admittedly, dealing with arrays of binary data and having to remember opcodes is a bit tedious, so let's think of a language in which we write *what* we want the computer to do on which data we want the operation to be performed on. Of course, we want to express our data in the decimal format that we are familiar with. We could, for example, express $5+4-2$ as a list of the following five steps:

```
LOAD(5);
ADD(4);
WRT();
SUB(2);
WRT();
```

This is a more compact representation of the binary code representation that you are already familiar with:

```
int DataRAMContent[] = {
    B0101, B0100, B0000, B0010,
    B0000, B0000, B0000, B0000,
    B0000, B0000, B0000, B0000,
    B0000, B0000, B0000, B0000,
};

int ProgramRAMContent[] = {
    B0010, B0000, B0110, B1000,
    B0110, B0000, B0000, B0000,
    B0000, B0000, B0000, B0000,
    B0000, B0000, B0000, B0000,
};
```

Don't you agree that our new programming language is much easier to read?

However, our B4 doesn't yet understand what a LOAD, ADD, WRT and SUB command means and has definitely no idea what it should do with these commands. LOAD, ADD, WRT and SUB are called an *assembly language*, whilst the 0s and 1s we have been working with so far form a *machine code*. Internally, the B4 can only understand machine code.

So we need to write a program that can translate assembly to machine code. This is called an *assembler*.

To write an assembler, we first match the assembly commands to the corresponding machine code instructions. Let's do this in the following table.

| Assembly Language | Machine Code |
|-------------------|--------------|
| LOAD | B0010 |
| ADD | B0000 |
| WRT (write) | B0110 |
| SUB (subtract) | B1000 |

Matching Assembly Language with Machine Code

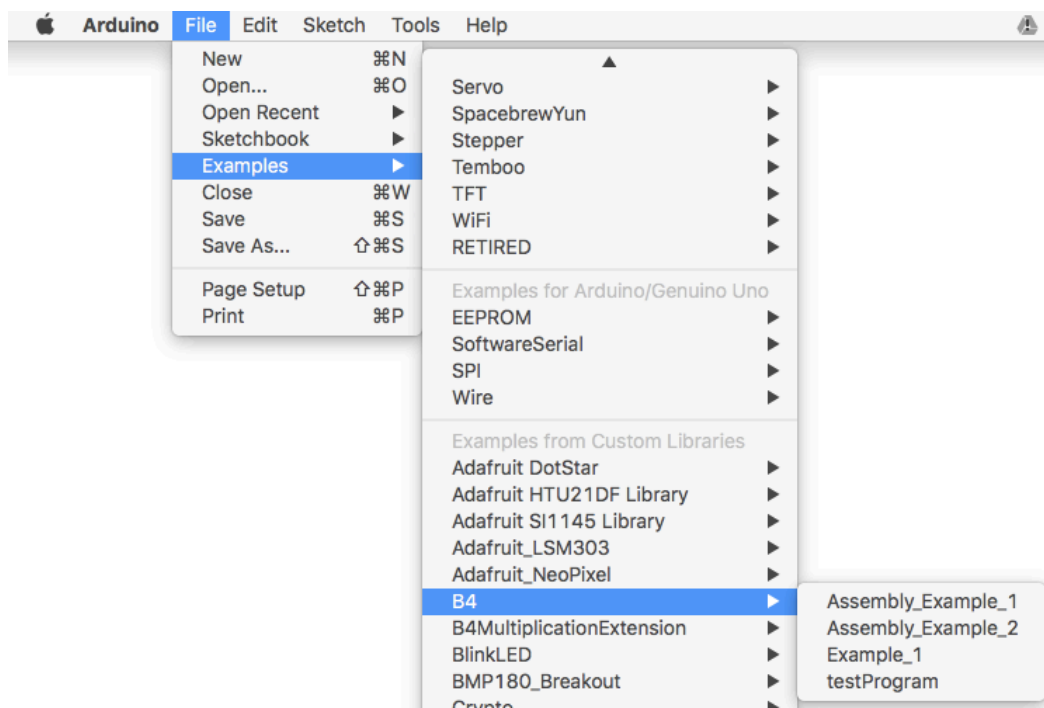
You notice, that LOAD, ADD and SUB have just one active bit, whilst WRT has two (B0110). Technically, we could decide that WRT is B0100 as this would suffice to write data into the Data RAM module. By activating the bit for the 2-to-1 Selector, we apply a clever little trick which allows us to use the result of a WRT command as input for the next arithmetic operation.

Our assembler will perform the following steps:

- 1) Break the program into the individual commands
- 2) Map the assembly commands to machine code
- 3) Bring the machine code into the proper sequence into the ProgramRAMContent[].
- 4) Identify the data that belongs to each command (5,4,0,2,0) and copy it into the DataRAMContent[] array.

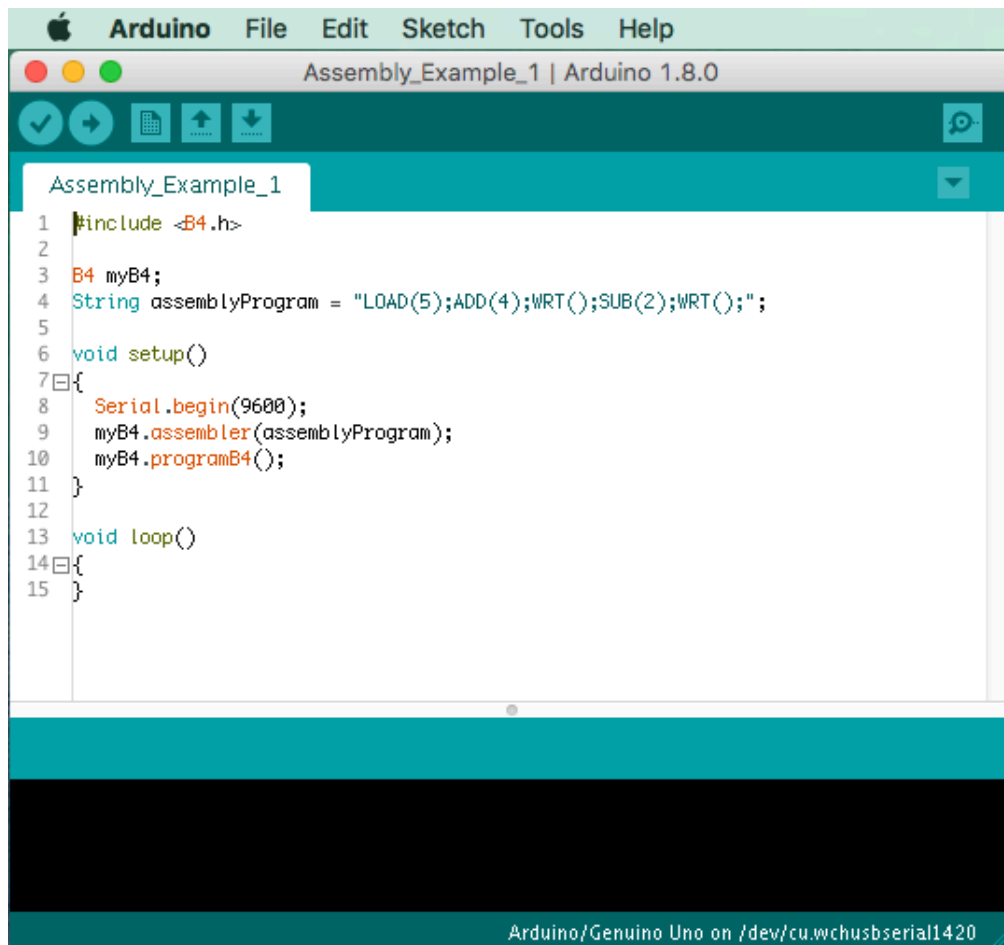
The function that performs the above-mentioned tasks (and more) is called *assembler* and is part of the B4 library. If you would like to get to know its details, you can open the file B4.cpp in the Arduino/libraries/B4 folder.

With this new function in place, the programming of our B4 is now significantly simplified. As shown in the following figure, go to Examples/B4/Assembly_Example_1 in the Arduino IDE and open it.



Loading the Assembler Example 1 Sketch

This will load the following Arduino sketch:

A screenshot of the Arduino IDE interface. The title bar shows 'Assembly_Example_1 | Arduino 1.8.0'. The menu bar includes 'Arduino', 'File', 'Edit', 'Sketch', 'Tools', and 'Help'. The toolbar has icons for check, run, upload, and download. The main text area shows the following code:

```
1 #include <B4.h>
2
3 B4 myB4;
4 String assemblyProgram = "LOAD(5);ADD(4);WRT();SUB(2);WRT()";
5
6 void setup()
7 {
8   Serial.begin(9600);
9   myB4.assembler(assemblyProgram);
10  myB4.programB4();
11 }
12
13 void loop()
14 {
15 }
```


The status bar at the bottom indicates 'Arduino/Genuino Uno on /dev/cu.wchusbserial1420'.

Assembly Example Sketch

In line 4, we declare a string, which we call *assemblyProgram* and fill it with our assembly code. Each command is completed by a semicolon.

"LOAD(5);ADD(4);WRT();SUB(2);WRT()";

You may have seen this before, for example in programming languages such as C, C++ or Java. The semicolon at the end of the line indicates the end of a command. This makes it much easier for the assembler to distinguish individual commands from each other and therefore translate assembly code correctly into machine code.

In line 9, we call the *assembler* function and pass the *assemblyProgram* along. Our B4 library will then perform the translation steps described above and produce the *DataRAMContent[]* and *ProgramRAMContent[]* arrays that you are already familiar with from the previous pages. You don't get to see them in this code, as they are being generated internally in the B4 library, but you can see them and some of the internal operation of the B4 library when you open the Arduino IDE's Serial port monitor . Make sure to set the baud rate to 9,600.

The final step, as shown in line 10, is to call the *programB44()* function. This will perform the necessary steps to load the contents of the *DataRAMContent[]* and *ProgramRAMContent[]* arrays into the B4's Data and Program RAM modules.


Simplifying our Program

Our program contains two WRT() functions. The first one stores the result of the 5+4 operation, whilst the second one stores the final result of 5+4-2. As the B4's Latch already holds on the result of 5+4 it is not really necessary to store 9 in the Data RAM. We can therefore simplify our program to:

```
"LOAD(5);ADD(4);SUB(2);WRT();"

```

Since the final result is only stored in RAM, but not being used for further arithmetic operations, the setting of the 2-to-1 Selector bit as part of the WRT() assembly code is irrelevant. It can therefore be simplified to B0100. You see how simple design choices, such as WRT() being either B0110, or B0100 are often made by the function we expect a computer to perform.

| Exercise 12.1 | |
|--|--|
|  | If you were to design a calculator, would you design WRT() to be B0110, or B0100? |
| | If WRT() were B0100 and you wanted the B4 to run the following program "LOAD(5);ADD(4);WRT();SUB(2);". What would the output of the Latch be after program step 3 has been executed? Why is the result not 7? How can this be explained? |

Summary

In this experiment, we have made a great step forward in simplifying the programming of the B4 and the readability of the B4 programs. We have designed our own higher-level programming assembly language and have translated the assembly code into machine code with an assembler program that is part of the B4 Arduino library.

When writing programs for the B4, we can now deal less with the internal workings of the computer. For example, the ADD instruction ensures that the 2-to-1 Selector's output is from the Data RAM.

This experiment has set the foundation for the design of compilers for other programming languages. For example, it is conceivable to translate some simple Scratch™ code into B4™ assembly and from there into B4 machine code. Or you could think of your very own commands instead of LOAD, ADD, SUB and WRT, possibly in a foreign language. You could even design your own programming language.

We would like to encourage you to explore this further.

Experiment 13: On the Role of Timing

In the previous chapters we have discussed that it is essential that the timing of the different components is done just right, so that the B4's modules operate in concert.

When we press the Enter button on the Program Counter, the following sequence of events takes place. We created a small video that you can find at <http://digital-technologies.institute/videos>. to watch every single step. Let's begin:

- 1) The Program Counter updates its value. It adds 1 to whatever it is showing presently.
- 2) The CLK signal is being generated and sent to the Latch. The Latch then stores the data from the 2-to-1 Selector.
- 3) Both, Data RAM and Program RAM switch to the data referenced by the Program Counter
- 4) The !CLK signal is being generated and sent to the Program RAM.
- 5) Where the bits are set to 1, the new output of the Program RAM activates the Inverter, 2-to-1 Selector, and storage into the Data RAM
- 6) If the WRT bit is 1, it is combined with the !CLK signal and sent to the Data RAM, which will then store whatever data is in the Latch. On the B4, the !CLK signal is as long as you press the Enter button. In comparison, Apple's A9 processor has a maximum clock rate of 1.85 GHz. There, a CLK or !CLK signal would only be 0.9 nano seconds long, or 0.000 000 000 9 seconds.
- 7) The output of the Data RAM is fed to the Inverter and the 2-to-1 Selector
- 8) The adder adds the data from the Latch and the data from the output of the Inverter

We see that, at step 2 of a new cycle, the Latch stores the result from the arithmetic operation of the previous cycle.

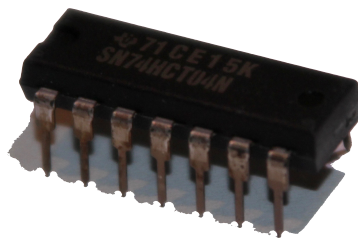
As we have seen, even a very simple computer like the B4 requires quite a bit of coordination. You can imagine that the timing of modern processors is much more sophisticated. Let's assume we have a modern smartphone with a 1GHz processor. 1GHz means that the processor operates at 1 billion instructions per second and that one instruction is therefore 1 billionth of a second long. The speed of light, and therefore the speed at which electricity can travel through a wire, is approximately 300,000 km per second, or 300,000,000 meters per second. Distance is defined as speed x time, so if we multiply 300,000,000 m/s with 0.000 000 000 1 s we get 30cm.

The faster our processors tick, the shorter the maximum allowable length of the wires. That's one of the reasons, why, for example, the USB wires to external devices are never very long. As the transfer speed increases, so has the length of wires to decrease.

Experiment 14: So, how does a Computer work ... actually?

Now that you have progressed to this chapter you have learned about the different parts that a basic computer is made of, such as an adder, inverter, latch, etc. You have also learned that opcodes control the flow of data and activate and deactivate modules and that they instruct the RAM to store data.

You might wonder, however, how all this is happening physically. In experiment 5, where we discussed random data, we mentioned that the RAM is made of hundreds of little switches. The switch nature is true for all the logic chips that you find in the B4. These are the little black boxes with legs. They look like this:



A Logic Gate Integrated Circuit

The question is: What do they do? Let's explore this on the following pages.

Computers exist because of three major achievements:

- 1) Our philosophers, scientists and mathematicians have developed the concept of logic which is the systematic study of the form of arguments.
- 2) Some more philosophers, scientists and mathematicians have been able to translate really complex logic to simple yes/no decisions.
- 3) Our physicists and engineers have learned to design and build machines where switches are so tiny so that millions and billions of them can be packed in tiny spaces where they reliably and rapidly solve logic problems near light speed.

Logic and Boolean Logic

Let us consider the above points 1) and 2) a bit closer. The systematic study of logic dates back to ancient times in China, India and Greece. One of the founding fathers of Greece logic, which became widely used in the Western and Arabian world, was Aristoteles. He lived in the 4th century BC. His work set the foundation of more work on logic since then, including in the Middle Ages. In the 1850's Mr. George Boole made a remarkable breakthrough when he developed a branch of algebra in which the values of the variables are the truth values TRUE and FALSE. In his honour, we speak of Boolean Logic.

The history of logic alone would fill many books and is outside of the scope of this handbook, but suffice to say that today's computing has a foundation that started some 2,500 years ago.

Let's explore Boolean Algebra: You would be surprised to hear that just a few words in the English language (and in most if not all other languages) are the key to modern computer science. These are TRUE, FALSE, AND, OR, and NOT.

Let's take a look: If you want both, apples and bananas you would say: "I would like apples AND bananas". This indicates to anyone hearing you that you want both. However, you might be content with receiving apple or bananas, or both, then you would say "I would like apples OR bananas". Your mum would then give you apples, or bananas, or apples and bananas. Let's assume you don't like bananas and you want to make sure your mum doesn't give you bananas. Then you could say. "I would like apples but NOT bananas". If you wanted apples or bananas, but never both, you would say: "I would like either apples or bananas"

These logical operations are called AND, OR, Negation, and Exclusive OR (XOR)

"I would like ..." is a bit verbose in day to day use in mathematics and computer science, so we can safely reduce these expressions to:

apples AND bananas
 apples OR bananas
 apples AND NOT bananas
 apples XOR bananas

Let's assume that you want to build a little machine that looks at the inputs to tell us whether your request has been met, with a simple TRUE/FALSE output statement.

We can use a truth table to determine if these conditions are met. Below, we have written the truth tables for AND, OR, AND NOT (NAND), and Exclusive OR (XOR)

| apples | bananas | output |
|--------|---------|--------|
| TRUE | TRUE | TRUE |
| TRUE | FALSE | FALSE |
| FALSE | TRUE | FALSE |
| FALSE | FALSE | FALSE |

apples AND bananas truth table

| apples | bananas | output |
|--------|---------|--------|
| TRUE | TRUE | TRUE |
| TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE |
| FALSE | FALSE | FALSE |

apples OR bananas truth table

| apples | bananas | output |
|--------|---------|--------|
| TRUE | TRUE | FALSE |
| TRUE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| FALSE | FALSE | FALSE |

Apples AND NOT bananas truth table

| apples | bananas | output |
|--------|---------|--------|
| TRUE | TRUE | FALSE |
| TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE |
| FALSE | FALSE | FALSE |

Either Apples or bananas (XOR) truth table

A Logical Adding Machine

Let's put our newly-acquired knowledge about logic to some good use and think about a machine that adds two one bit binary numbers, A and B. This will result in a 2 bit number. From now on, let's set 1 for TRUE and 0 for FALSE so we have a little bit less to write. How would the truth table of such a machine look like? Let's have a look at the following table:

| A | B | sum |
|---|---|-----|
| 1 | 1 | 10 |
| 1 | 0 | 01 |
| 0 | 1 | 01 |
| 0 | 0 | 00 |

Adding two binary numbers

$1+0=1$ and so is $0+1$. $0+0=0$ and $1+1=2$, which is in the binary system 10 (one zero).

We can write this a bit differently in the following form:

| A | B | sum (higher bit) carry over | sum (lower bit) |
|---|---|--------------------------------|-----------------|
| 1 | 1 | 1 | 0 |

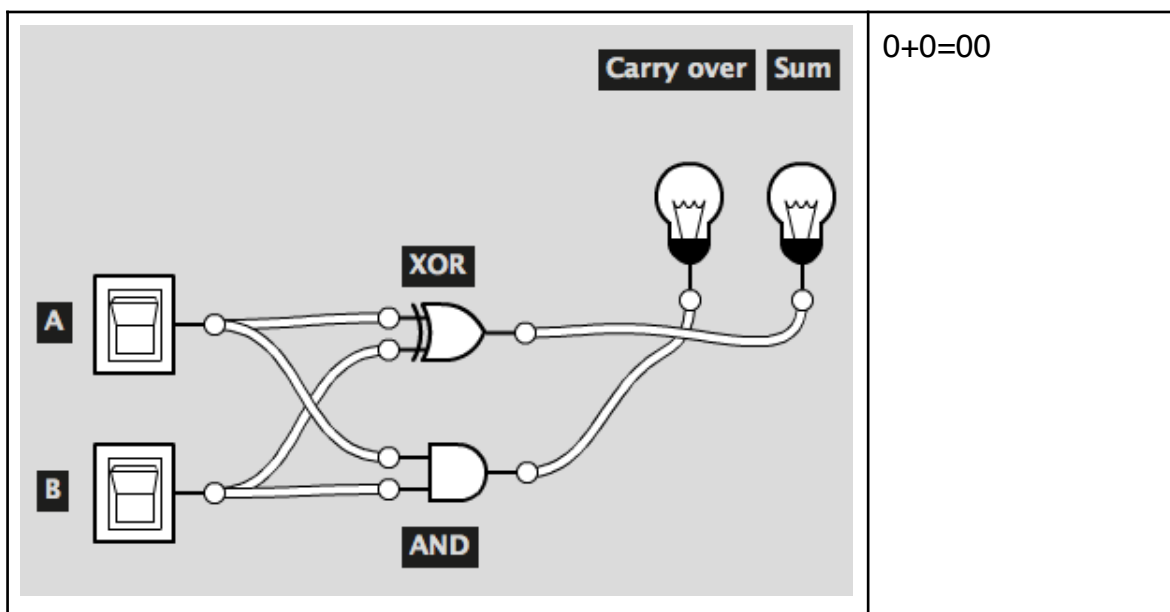
| A | B | sum (higher bit) carry over | sum (lower bit) |
|---|---|--------------------------------|-----------------|
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

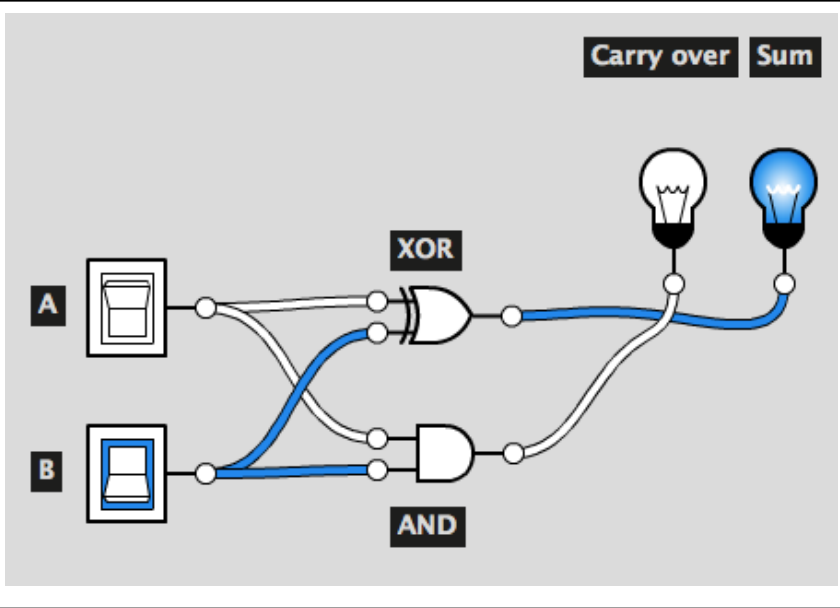
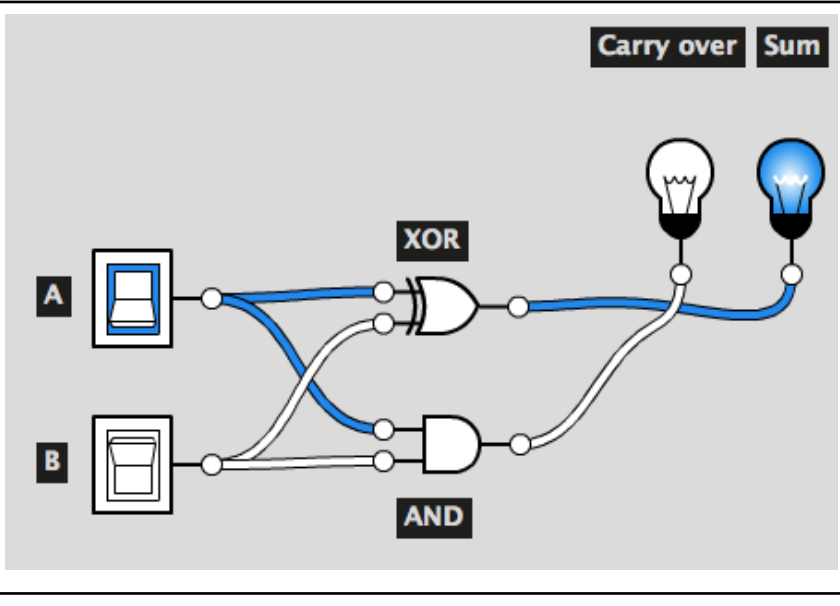
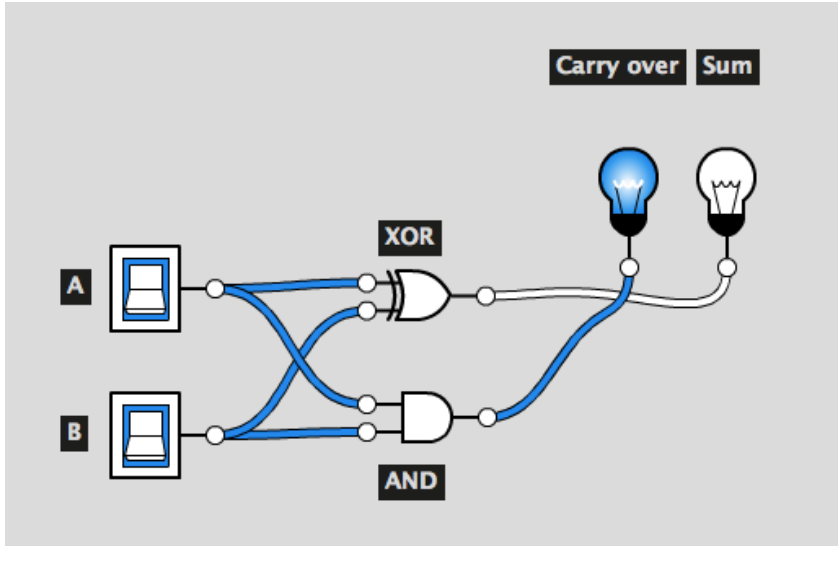
Adding two binary numbers

So, the sum's lower bit is 1 when either A OR B are 1, but not when both or none of them is 1: So we write: A XOR B

The carry-over is only 1 when A AND B are both 1: We write A AND B

So, to add two 1 bit numbers , we need two machines: One XOR machine and one AND machine. Let's call these machines gates. Let's then arrange these two machines so that our two binary numbers A and B are connected to the inputs of the AND and XOR Gates. Let's then change the values of A and B and observe the sum and carry over outputs. The following table shows the 4 possible combinations of A and B and the outputs that our gates produce. Let's have a look:



| | |
|--|----------|
|  | $0+1=01$ |
|  | $1+0=01$ |
|  | $1+1=10$ |

A Simple Adding Machine with Logic Gates

By applying Boolean logic to the problem of arithmetic, we can design a small machine that can add two binary values. We have not yet found out how we would actually engineer

such a machine. Let's park the engineering issue for a moment until we have applied Boolean logic to the issue of memory in the following section.

A Logical Memory Machine

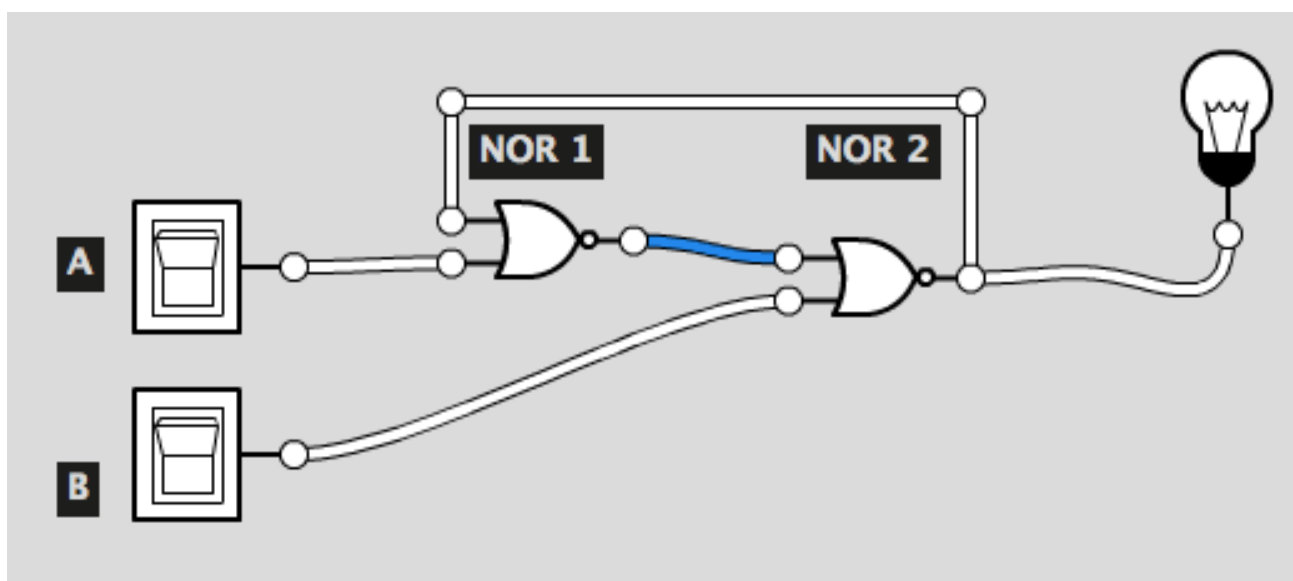
Logic gates can not only add, but also remember. Memory, as you have seen throughout this handbook, when we explored the Latch and RAM modules, is a fundamental function of a computer.

To explore this further, let's quickly expand our knowledge of the logic gates from the previous section, where we learned about AND, OR, NOT, and XOR. If we combine OR and NOT, we get a gate that is called NOT-OR, or, in brief, NOR. The truth table for NOR is similar to the familiar OR truth table, with the main difference being that the output is always negated. This means that when the OR gate produced a TRUE output, the NOR gate produces a FALSE and when OR resulted in FALSE, then NOR will be TRUE. The NOR truth table is shown below.

| A | B | output |
|-------|-------|--------|
| TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| FALSE | TRUE | FALSE |
| FALSE | FALSE | TRUE |

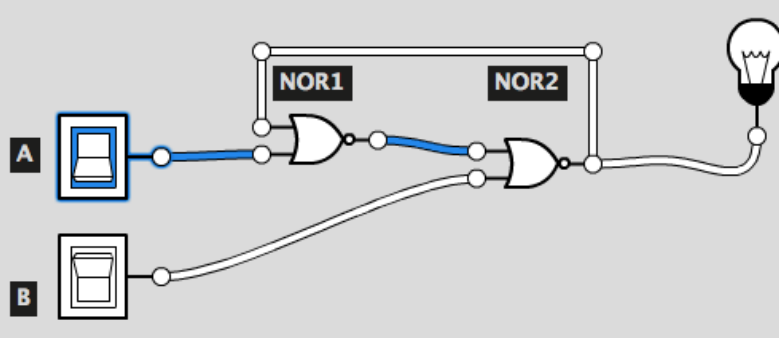
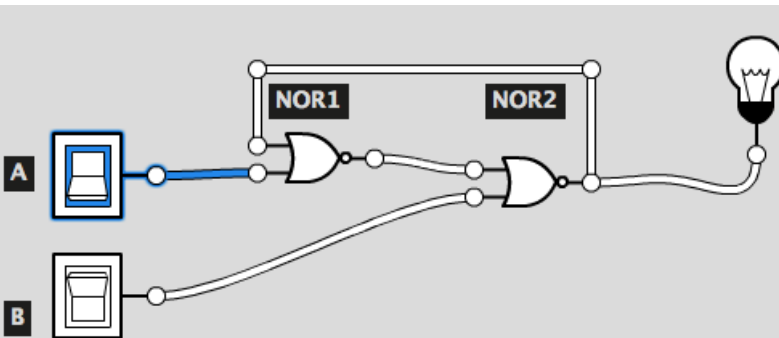
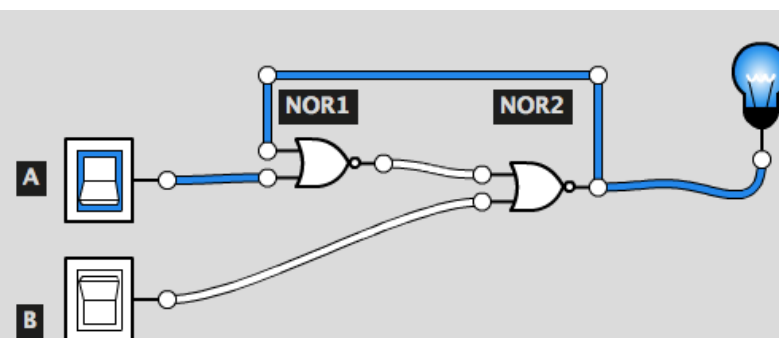
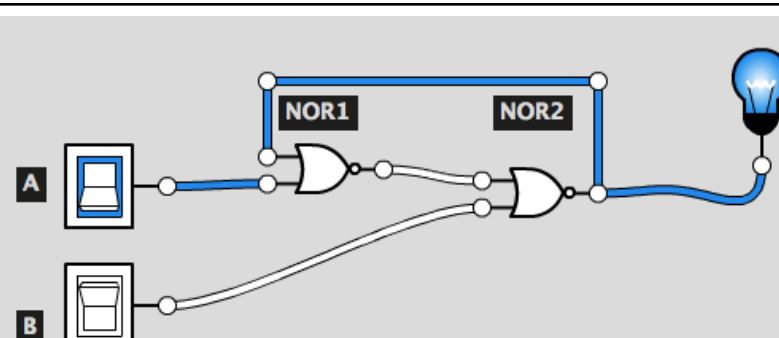
NOR truth table

We now take two NOR gates and wire them up in a way that the output of each gate is connected to the input of the other. This, as you will see, is a common characteristic in computers: The output of one part is the input of another, and vice versa. This is called a feedback loop.

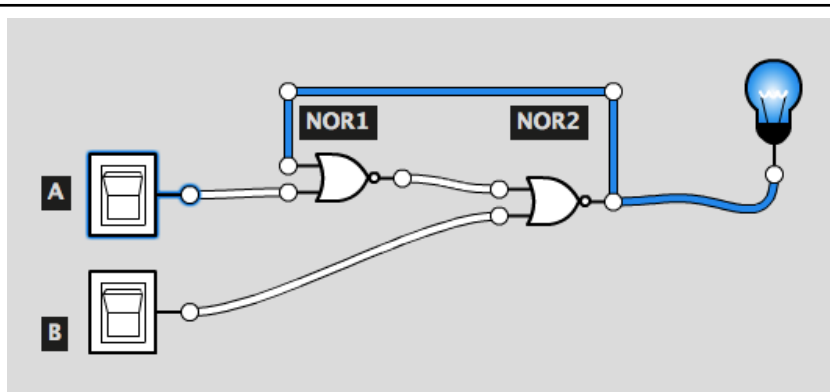


A Feedback Circuit with two NOR gates

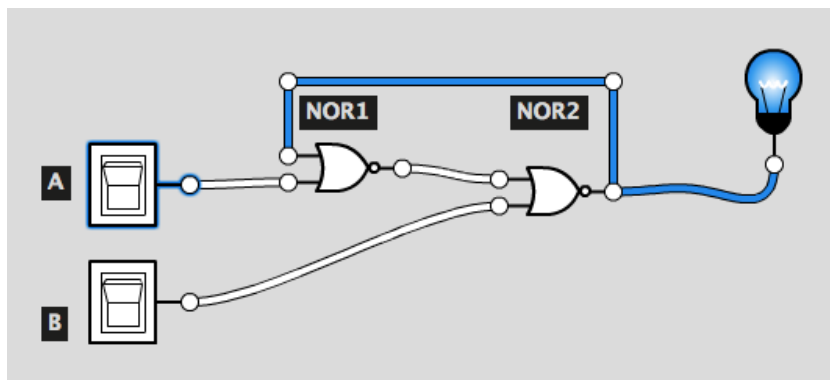
Let's explore this circuit by playing with our input switches A and B. We start by pressing button A.

| | |
|---|---|
|  | <p>Button A has been pressed. NOR1 received a 1 from button A and a 0 from NOR 2 ...</p> |
|  | <p>... 0 NOR 1 is 0, which the NOR1 gate now gives to the NOR2 gate ...</p> |
|  | <p>... 0 NOR 0 is 1, which the NOR2 gate now gives to the light bulb and to the NOR1 gate ...</p> |
|  | <p>... and because 1 NOR 1 is 0, the NOR1 gate does not change its output. The circuit is now stable.</p> |

But what happens when we release button A? Let's find out.



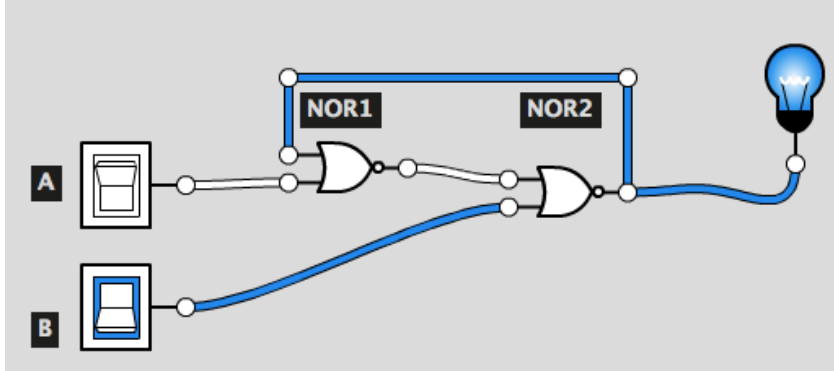
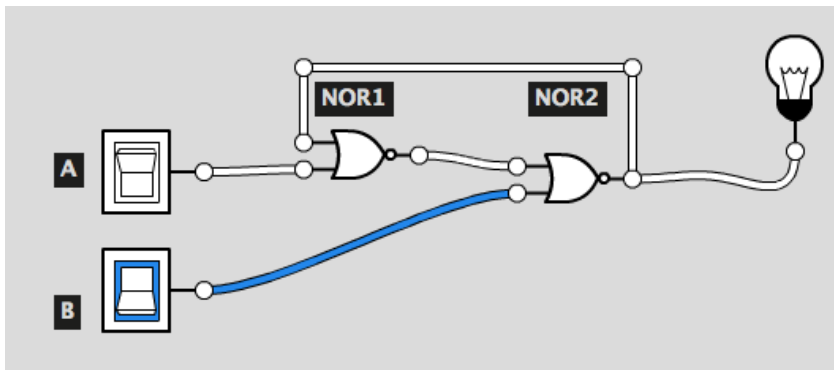
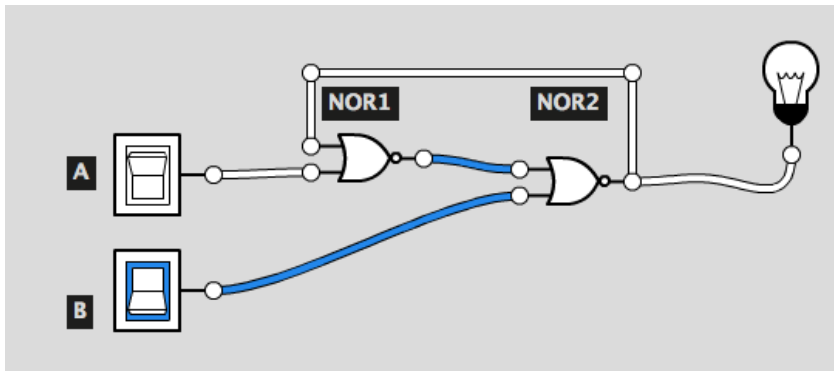
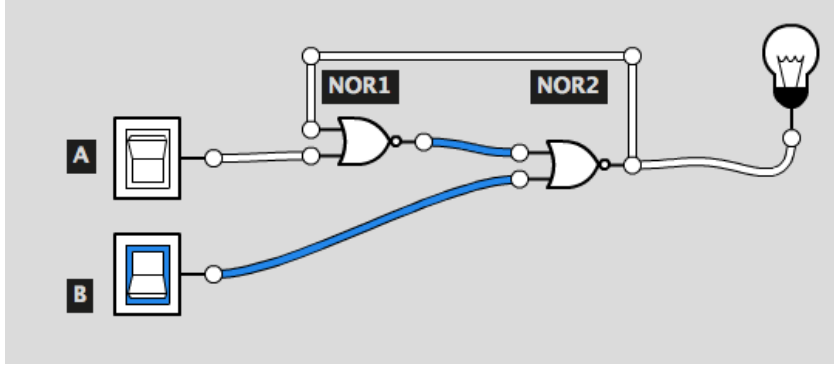
Button A has been released. NOR1 receives a 0 from button A and a 1 from gate NOR2 ...



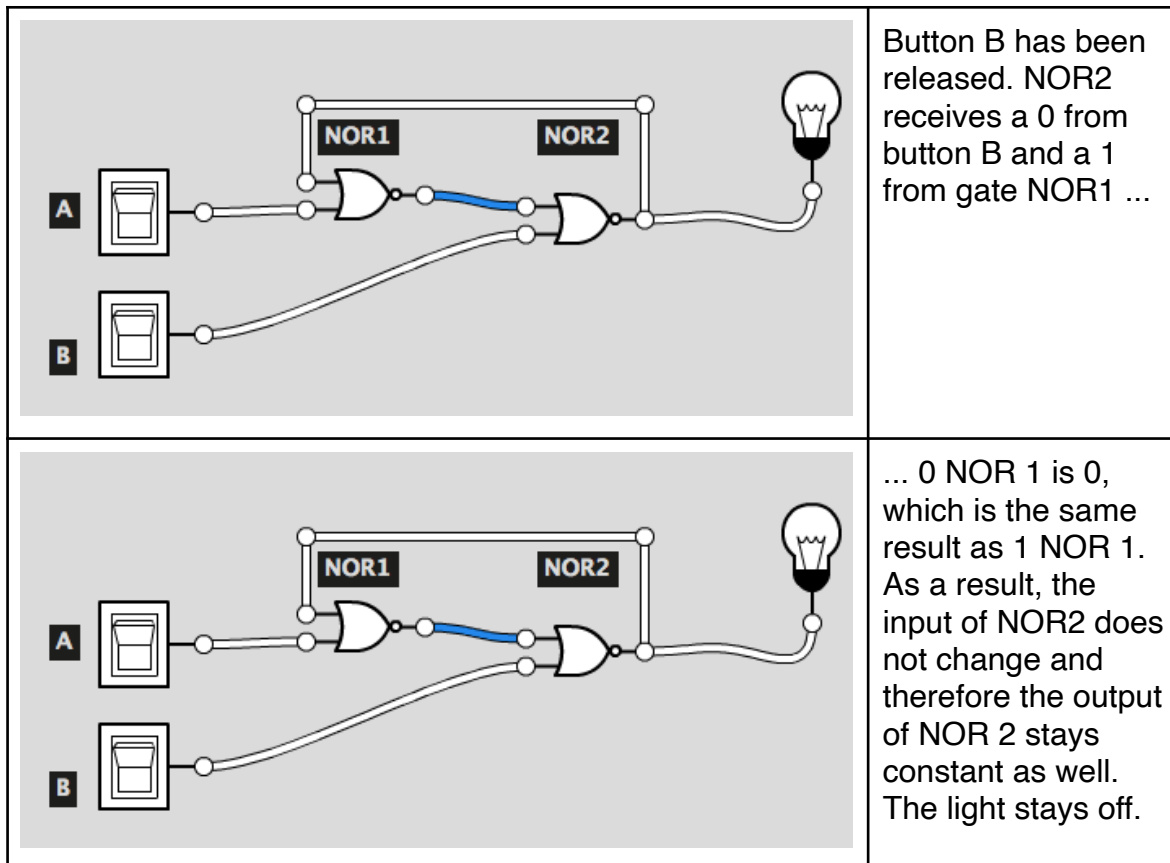
... 0 NOR 1 is 0, which is the same result as 1 NOR 1 when the button A was still pressed. As a result, the output of NOR1 does not change and therefore the output of NOR 2 stays constant as well. The light stays on.

Releasing button A has no impact on the output of our circuit. It has remembered that A has been pressed. We have just constructed a 1 bit memory cell - congratulations !

Our memory cell not only needs to remember when it was activated (1), but also when it should reset to 0. An this is the function of button B. Let's now explore when button B is pressed. We continue from the previous picture.

| | |
|---|---|
|  | <p>Button B has been pressed. NOR2 receives a 1 from button B and a 0 from NOR1 ...</p> |
|  | <p>... 0 NOR 1 is 0, which the NOR2 gate now gives to the output and to the NOR1 gate ...</p> |
|  | <p>... 0 NOR 0 is 1, which the NOR1 gate now gives to the NOR2 gate ...</p> |
|  | <p>... and because 1 NOR 1 is 0, the NOR2 gate does not change its output. The circuit is now stable.</p> |

Finally, we release button B. This produces the following steps



You have probably seen the similarities between switching the buttons A and B on, and between switching them off. We can say that one gate plays the helper for the other gate to keep it either on and off. In this relationship neither of the two gates plays any greater or lesser role than the other gate. It is interesting to note that neither of the two NOR gates is able to store information by itself. However, two NOR gates, properly connected with each other has the ability to memorise information. This circuit is called a flip-flop. The first electronic flip flop was invented by two British physicists in 1918. Since then, many different types of flip-flops have been invented. Some of them use other gate types than NOR, such as NAND (NOT AND) gates. However, common to all flip-flops is the feedback characteristic between at least two gates and that flip-flops can hold a state. Some flip-flops only require one input switch, as opposed to the two input switches that our flip-flop uses. Our flip-flop is a SR NOR flip-flop. SR means 'set-reset' and denotes two inputs: one to Set the flip-flop to an output of 1 and another to Reset the flip-flop's output to 0. In our SR NOR flip-flop, button A is the set button and B is the reset button.

Engineering

To this point, we have learned that we need different types of gates (AND, XOR) to make an adding machine, and other gates (NOR) to build memory. Each of these gates can be constructed of a cleverly-arranged set of little switches, called transistors. They have been around since the 1920's, but developed in earnest since the 1940's. Transistors are electronic switches that can be closed by applying an electric current. They can be fabricated in semiconductor materials and can be made so tiny so that billions of them fit on a chip the size of your fingernail. A typical AND or OR gate would require 2 transistors, a XOR gate 6 and a NOR gate 2.

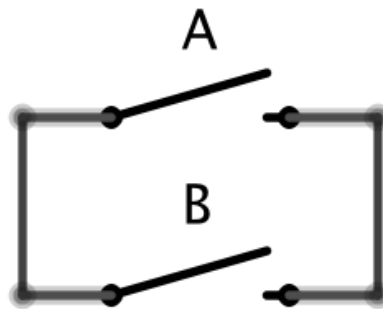
For example, to build an AND gate, one would arrange two switches in sequence as follows:



Realising an AND gate with two switches

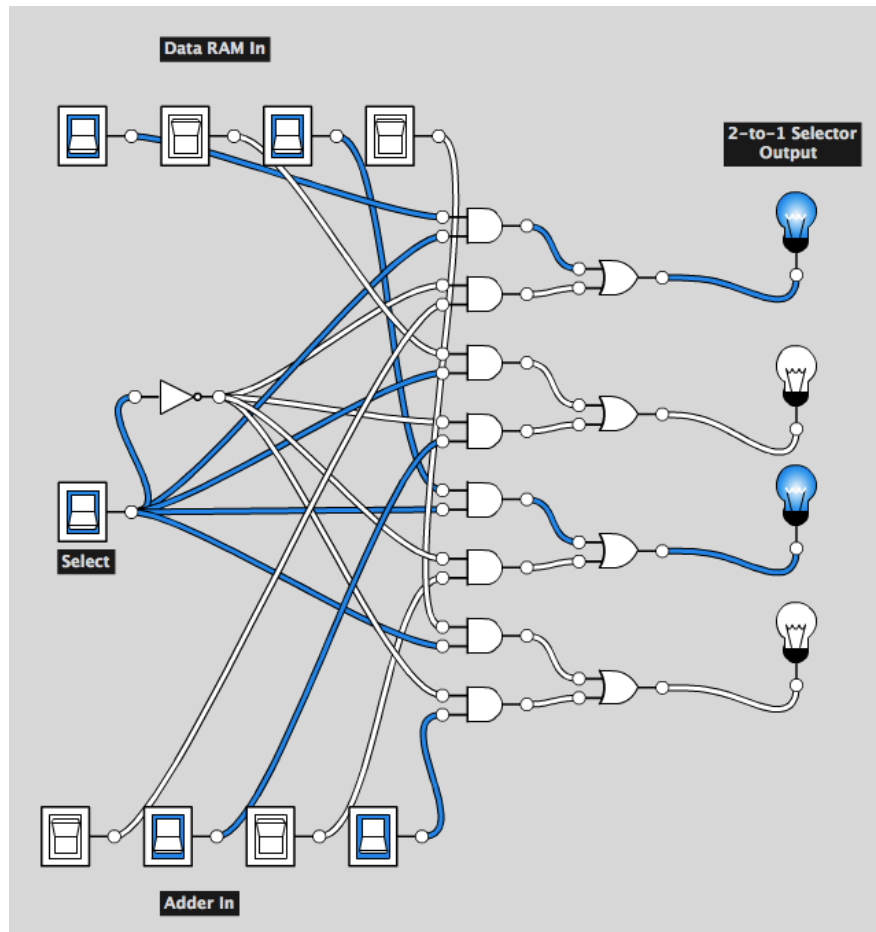
The circuit can only be closed by closing the switches A and B simultaneously.

In order to make an OR Gate, we would arrange the switches in parallel, so that when either is pressed, current can flow. This would look like this:



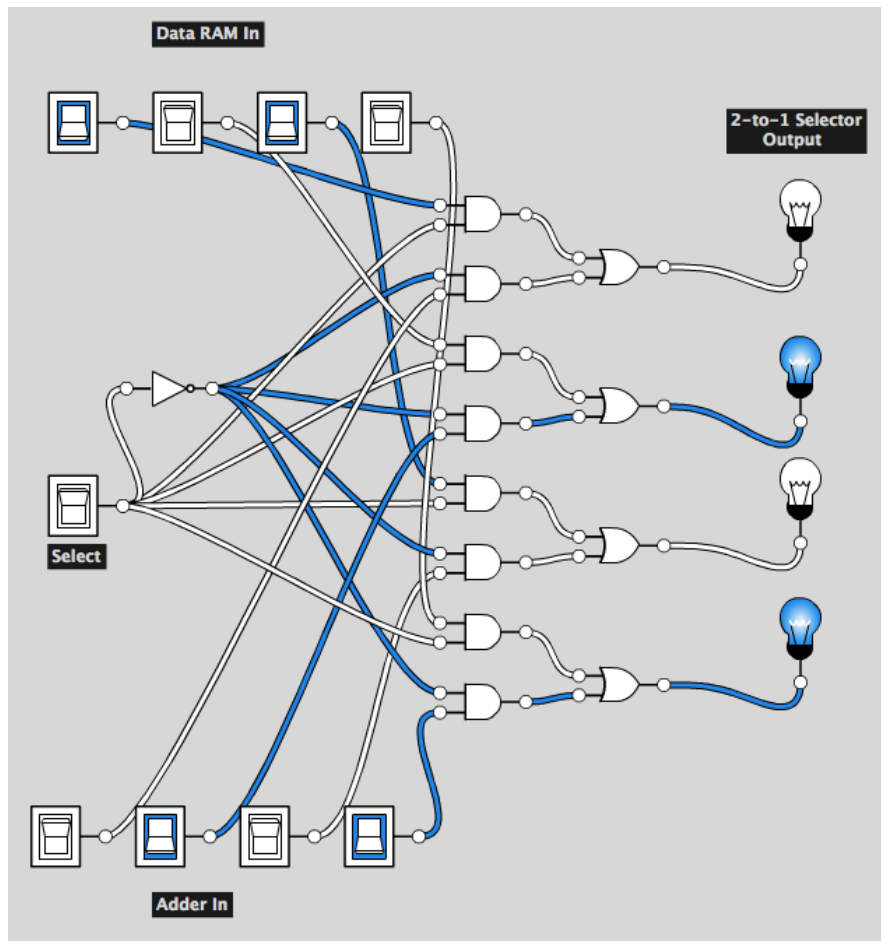
Realising an OR gate with two switches

Let's look at a concrete example of AND and OR gates: The B4's 2-to-1 Selector is a set of transistors arranged in such a way that they switch data from an input to an output. In their *on* state they switch data from the Data RAM. In their *off* state, the data is directed from the Adder to the output. Below, we have the logic diagram of the inside of the 2-to-1 Selector. Let's take a look:



Inside the 2-to-1 Selector

At the top you see four switches that represent the input from the Data RAM. At the bottom, there are four switches representing the Adder Input. On the right, there are four Light bulbs representing the output of the 2-to-1 Selector. These are the same lights you see on the 2-to-1 Selector module. In the middle, on the left hand side of the above figure, you see a switch, called *Select*. When activated, it selects the data from the Data RAM to be channelled to the output. When in the off state, data from the Adder will reach the Output. In between the switches and light bulbs, you see 8 AND gates and 4 OR gates = 12 gates in total. Each gate consists of two transistors leading to 24 transistors. There is also one inverter consisting of 1 transistor. The entire 2-to-1 Selector circuit therefore consists of 25 transistors. Try to analyse the function of this circuit. To help you, we provide one additional screenshot with the Select switch in off position:



2-to-1 Selector (Select switch off)

According to Wikipedia, the largest transistor count in a commercially available single-chip processor in the year 2016 is over 7.2 billion. This is the Intel Broadwell-EP Xeon processor.¹

You can imagine that a chip consists of transistors that have been arranged in such a way that they form all sorts of different gates which are interconnected in clever ways so that they form arithmetic units that can perform calculations, such as adding. Other gates interact to work as memory and other gates engage in the control flow of data. This is quite extraordinary, as the underlying transistors can only switch on and off. By connecting them intelligently, we can let them perform very complex functions, which you see every day when you use a computer. Brilliant research was required to produce special materials, such as semiconductors, which have defined capabilities to conduct electronic current only when an electric charge is applied to them. In the diagram below, you see how each design step in the design process from semiconductors to transistors and from there to gates and higher-level functions has led to increased functionality and sophistication. Semiconductors were first discovered around the year 1821. It took 150 years of research and development until the first integrated micro-controller, the Intel 4004, was released.

¹ Source: Wikipedia: https://en.wikipedia.org/wiki/Transistor_count

| |
|---|
| higher-level functions, such as Arithmetics, Memory, Switching, etc. |
| Gates |
| Transistors |
| Semiconductor Materials |

Summary

In this chapter we have learned how the 2,500 year long history of logic has led to a method of Boolean algebra in which we can define logical functions we call gates. Intelligently arranged, these gates can be put to good use to add or store information. Gates themselves are made of transistors, also intelligently arranged to perform the desired function of the gates, such as AND, XOR, NOR, etc. Computer chips can consist of billions of transistors. The design of a computer chip is therefore a high-tech task that requires many scientists and engineers. The B4's different modules demonstrate some of the most important parts of a computer's central processing unit. Each of the B4's modules has chips on it, which are internally made of gates and transistors. We haven't really counted them, but we estimate that the B4 is made of a few thousand gates. Most of them would be in the Data RAM and Program RAM chips.

| | |
|---------------|--|
| Exercise 14.1 | |
| ? | Compare your knowledge about transistors that form gates to what you know about biological systems. Can you identify similarities? |
| | If transistors were made of mechanical parts that moved, rather than semiconductor materials, what disadvantages would this bring? |
| | How much does it cost to manufacture a microprocessor? What would be the price per transistor for this microprocessor? |

Experiment 15: Cyber Security

Once we understand the hardware and software of a digital system in detail, we can start to think about hacking it.

The B4 Computer Processor kit comes with a unique software library that interfaces between the B4 and the included Arduino Uno. You have used it previously when you programmed the B4 from the convenience of a laptop computer. However, it can also be used to hack into the B4.

It is possible to hack the library so that it alters data and program code. We can also make the Automatic Programmer module interfere with the normal operation of the B4 at runtime. For example, you can hack the B4 to perform subtraction instead of adding, or flip bits in the memory modules. It is quite entertaining and instructional to realise that hardware can get hacked at such a low level that no virus scanner would be able to detect it.

By conducting cyber security attacks in the B4, the impacts of these hacking exercises are contained in the B4 environment and do not impact on the safe and reliable operation of the connected laptop computers.

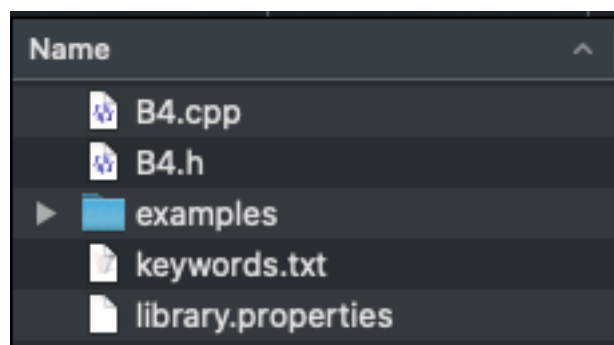
In this section we are investigating two strategies of hacking into the B4. We call such strategies attack vectors. The better you understand a system, the better you can hack it. We start with understanding the B4's software and then move to the function of its hardware.

Hacking into the computer processor is not necessarily intended to interfere with normal operations in a bad way. It can also increase its abilities.

Software: Understanding the B4's Arduino Library

In Experiment 11, you installed the B4 Arduino library on your laptop or PC. If you haven't done this yet, go back to Experiment 11 and follow the installation procedure. You will need the B4 library for this experiment. Previously you were a mere user of the B4's Arduino Library. You used it to send machine or assembly code from the Arduino IDE to the B4. We now take a deeper look into the library.

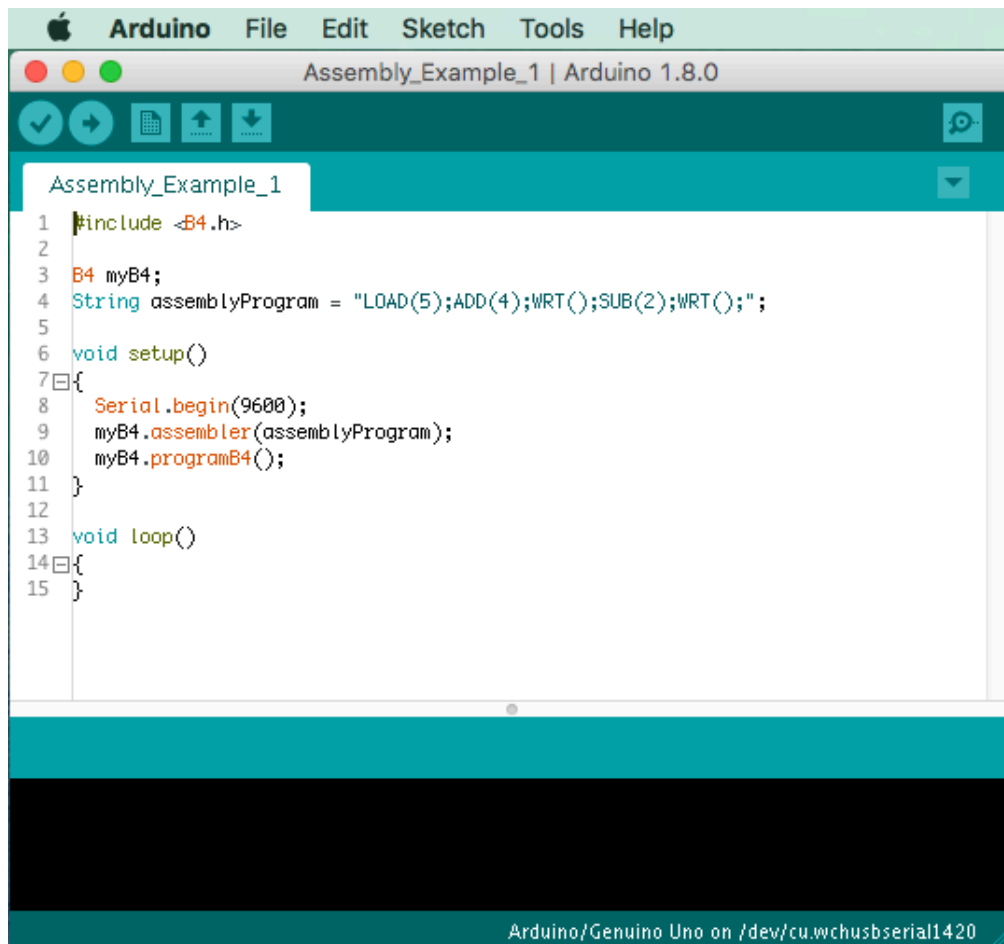
Find the Arduino libraries folder. On Windows and Macintosh machines, the default name of the folder is "Arduino/libraries" and is located in your Documents folder. In there you find a folder called B4. Open it.



Inside the B4's Arduino library

B4.cpp contains the implementation of the B4 library, whereas B4.h is a header file. It contains the definition of the B4 class. Yes, it is object oriented, but don't worry about this.

If you remember our little Assembly example from Experiment 12, you notice that we send the *assemblyProgram* string to the *assembler* function with `myB4.assembler(assemblyProgram)`



```
1 #include <B4.h>
2
3 B4 myB4;
4 String assemblyProgram = "LOAD(5);ADD(4);WRT();SUB(2);WRT()";
5
6 void setup()
7 {
8   Serial.begin(9600);
9   myB4.assembler(assemblyProgram);
10  myB4.programB4();
11 }
12
13 void loop()
14 {
15 }
```

Assembly example from Experiment 12

Open the file B4.cpp and find the *assembler* function.

It starts like this:

```
void B4::assembler(String assemblerProgram)
{
    String assemblerProgramLines[16];
    String assemblerCodes[] = {"LOAD", "ADD", "SUB", "WRT"};
    int machineCodes[] = {B0010, B0000, B1000, B0110};
    String assemblerCode;
    String dataCode;
    int semicolonIndex = 0;
    int openBracketIndex = 0;
    int closingBracketIndex = 0;
    int programCounter = 0;
    int programLength = 0;

    int DataRAMContent[] = {
        B0000, B0000, B0000, B0000,
        B0000, B0000, B0000, B0000,
        B0000, B0000, B0000, B0000,
        B0000, B0000, B0000, B0000,
    };

    int ProgramRAMContent[] = {
        B0000, B0000, B0000, B0000,
        B0000, B0000, B0000, B0000,
        B0000, B0000, B0000, B0000,
        B0000, B0000, B0000, B0000,
    };

    ...
}
```

The beginning of the assembler function in the B4 Arduino library

This function converts the assembly code (like LOAD(5), ADD(4) and so on) into the corresponding binary representation, which it stores in the *DataRAMContent[]* and *ProgramRAMContent[]* arrays.

Notice the two arrays at the top: *assemblerCodes[]* contains the four instructions that the B4 knows: Loading, Addition, Subtraction, and Writing (Storing) or data. The *machineCodes[]* array contains the matching machine codes.

```
assemblerCodes[] = {"LOAD", "ADD", "SUB", "WRT"};
machineCodes[] = {B0010, B0000, B1000, B0110};
```


In Experiment 12, we learned that LOAD maps to a B0010, ADD to B0000 and so forth, as per the following mapping table.

| Assembly Language | Machine Code |
|-------------------|--------------|
| LOAD | B0010 |
| ADD | B0000 |
| WRT (write) | B0110 |
| SUB (subtract) | B1000 |

Matching Assembly Language with Machine Code

For the correct operation of the B4 it is very important that these mappings are absolutely precise. But what if we changed the order of the elements of assemblerCodes array? We could, for example, swap ADD and SUB.

```
assemblerCodes[] = {"LOAD", "SUB", "ADD", "WRT"};  
machineCodes[] = {B0010, B0000, B1000, B0110};
```

Now every time the B4 is supposed to add two numbers, it will instead subtract them and when it is supposed to perform a subtraction, it will instead do an addition. That sounds like fun. To try this out, make these changes in the code of your B4 library and then save it. Then, load the Examples/B4/Assembly_Example_1 in your Arduino IDE. Again, flip back to Experiment 12 for the instructions. This will load the following program into your Arduino IDE:

```
LOAD(5);ADD(4);WRT();SUB(2);WRT();
```

Upload it to your B4's Automatic programmer and run the program. What do you observe?

Instead of performing $5+4-2=7$, the B4 will instead do $5-4+2=3$.

Observe that the user still sees the same program

```
LOAD(5);ADD(4);WRT();SUB(2);WRT();
```

But we have hacked one level deeper where the program is translated into machine instructions.

What else can we do? Well, we can design our own language. Instead of calling our instructions LOAD, ADD, SUB and WRT, we can name them differently. How about LIZARD, APPLE, SAUSAGE and WOMBAT? All you need to do to make this change is to write

```
assemblerCodes[] = {"LIZARD", "APPLE", "SAUSAGE", "WOMBAT"};  
machineCodes[] = {B0010, B0000, B1000, B0110};
```

A corresponding assembly program would then look like this

```
LIZARD(5);APPLE(4);WOMBAT();SAUSAGE(2);WOMBAT();
```

You can choose any words you like. You can design your very own secret programming language that no-one else can understand.

But perhaps you find the semicolons (;) that separate the instructions a bit dull and prefer exclamation marks instead.

```
LIZARD(5)!APPLE(4)!WOMBAT()!SAUSAGE(2)!WOMBAT()!
```

Looks cool. To make this change in your library, find the line

```
semicolonIndex = assemblerProgram.indexOf(';');
```

and change it to

```
semicolonIndex = assemblerProgram.indexOf('!');
```

Save the change and you can run

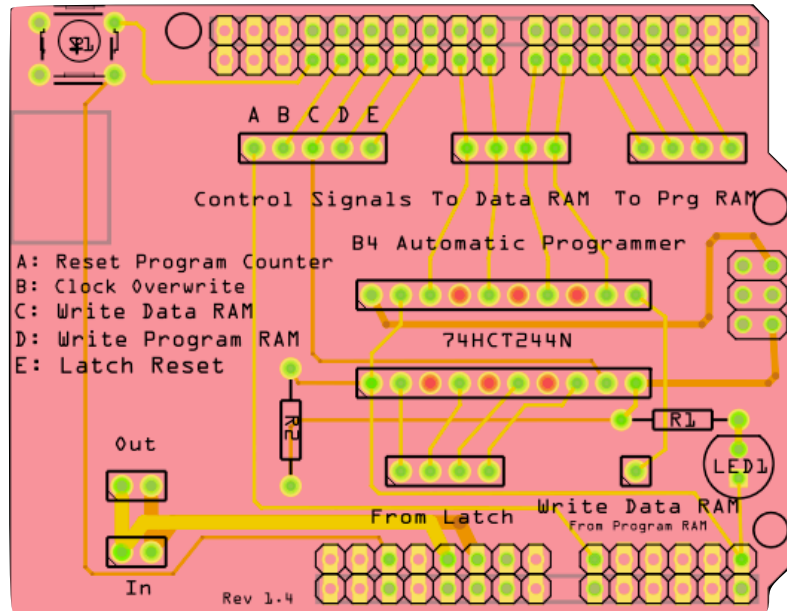
```
LIZARD(5)!APPLE(4)!WOMBAT()!SAUSAGE(2)!WOMBAT()!
```

from the Arduino IDE.

Have a play and imagine other words for your assembly instructions and the characters that could separate them. Be careful to only use separators that are not already part of your assembly instructions or the parentheses ().

Hardware: Hacking deeper yet by understanding the Automatic Programmer

Let's take a closer look at the hardware of the Automatic programmer. Here you see its circuit board with the various wires (yellow and orange) between the pins. For the moment, we are mainly interested in the light yellow wires.



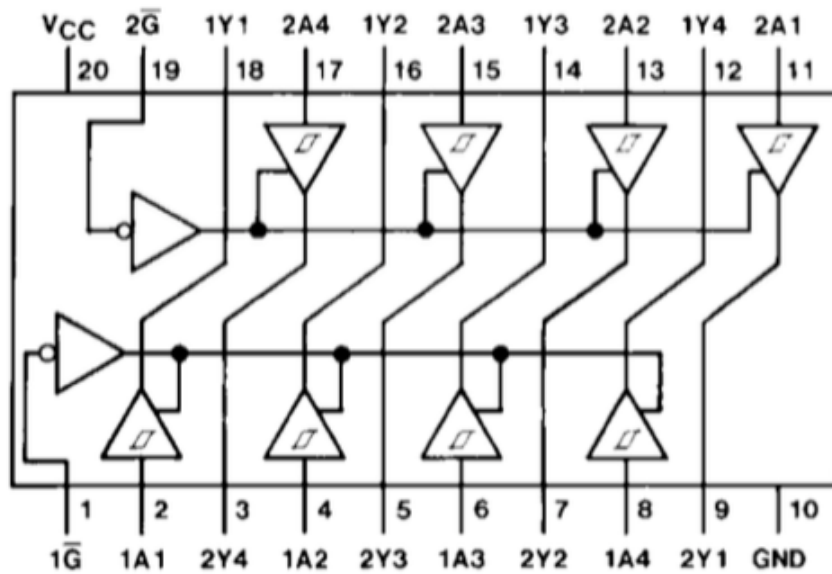
Automatic Programmer circuit board

The Automatic Programmer can do a number of interesting things:

- 1) It can send control signals to
 - A. Reset the Program Counter to zero, which is equivalent to the user pressing the Zero button on the Program Counter
 - B. Issue a Clock signal, which is equivalent to the user pressing the Enter button on the Program Counter
 - C. Issue a write command to the Data RAM
 - D. Issue a write command to the Program RAM
 - E. Reset the Latch

It can also provide binary data to the Data and Program RAM modules.

The Chip on the Automatic Programmer is a 74HCT244. We use it to let data from the Latch pass through to the Data RAM when the Automatic Programmer is inactive, or separate the Latch from the Data RAM and feed data from the Arduino Uno to the Data RAM. We use this when we program the Data RAM. Below is a schematic of it.



74HCT244 Schematic

The pins 1 \bar{G} and 2 \bar{G} are in charge of opening and closing the connections from the input pins denoted with an A, which are 1A1, 1A2, 1A3, 1A4, and 2A1, 2A2, 2A3, 2A4 to their corresponding output pins, which are 1Y1, 1Y2, 1Y3, 1Y4, and 2Y1, 2Y2, 2Y3, 2Y4.

The *74HCT244* allows for the separate operation of the four switches 1A1 to 1A4 by 1 \bar{G} and the four switches 2A1 to 2A4 by 2 \bar{G} . However, on the Automatic Programmer Shield, there is a connection between 1 \bar{G} and 2 \bar{G} . This means they are linked together and we can operate all eight switches at the same time.

So when we activate the Automatic Programmer by pulling the pin A5 of the Arduino Uno high to 5V, with:

`digitalWrite(A5, HIGH)`, the 74HCT244 will separate the Latch from the Data RAM. The Automatic Programmer will then act as a Man in the Middle and provide data of our own choosing to the Data RAM.

The Arduino-provided data for the Data RAM comes from the Arduino pins 6 .. 9
 The Arduino-provided data for the Program RAM comes from the Arduino pins 2 ... 5
 Further, the control signals are assigned to the Arduino pins as follows

```
reset program Counter = A0
latchReset = 10;
writeProgramRAM = 11;
writeDataRAM = 12;
clockCycle = 13;
```

The following table summarises the mapping of the Arduino pins to corresponding function of the Automatic programmer.

| Arduino pin number | Function | | Note |
|--------------------|---|--------|-------------|
| A5 | activates the Automatic programmer and separate the Latch from the Data RAM | | HIGH active |
| A0 | Reset Program Counter (program counter gets re-set to 0) | | HIGH active |
| 2 | Program RAM | x1 bit | HIGH active |
| 3 | Program RAM | x2 bit | HIGH active |
| 4 | Program RAM | x4 bit | HIGH active |
| 5 | Program RAM | x8 bit | HIGH active |
| 6 | Data RAM | x1 bit | HIGH active |
| 7 | Data RAM | x2 bit | HIGH active |
| 8 | Data RAM | x4 bit | HIGH active |
| 9 | Data RAM | x8 bit | HIGH active |
| 10 | reset Latch | | LOW active |
| 11 | write Program RAM | | LOW active |
| 12 | write Data RAM | | LOW active |
| 13 | clockCycle (Program Counter gets incremented by 1) | | HIGH active |

All the HIGH active pins require a

```
digitalWrite(pin number, HIGH)
```

to do something, whilst the LOW-active functions need a LOW to do their function with

```
digitalWrite(pin number, LOW)
```

In order to carry out hacking, all we need to do is to hijack the Arduino and activate or deactivate pins A0, A5, 2..13 in clever ways.

Let's explore a few ideas:

- 1) We could write a piece of software on the Arduino that would, at regular or random time intervals, pull pin 13 to HIGH and therefore issue a clockCycle to the Program

Counter. The user would then believe that perhaps the Enter button is broken, complain to the manufacture and get it replaced.

- 2) That same piece of software could briefly activate the Automatic Programmer, write some random data into the Data RAM and then deactivate. This will happen so quickly that the user will not see it happen.
- 3) The software could also quickly reset the latch, therefore erasing the intermediate results of a computation, or the result of a LOAD() instruction.
- 4) A little program could randomly re-set the Program Counter to 0.

There are further options that we could pursue. Let's try option 1) and 2), because they are a bit different. We start with 1)

Hack 1: Randomly incrementing the Program Counter

We start by specifying a piece of code that, at random intervals, performs a call to the clockCycle function in the B4 Library.

For this code, we start with the program Assembly_Example_1, which comes with the B4 library.

We add a new variable randNumber in which we store the value for the delay. The Arduino automatically calls the loop() function repeatedly, so we don't need to write another loop ourselves. All we need to do is add three lines of code.

- Firstly, a line that generates a random number. We have chosen a value between 5,000 and 19,999 mili-seconds (ms).
- Then a call to the clockCycle() function from the B4 library
- Finally the delay of the value we previously generated. This will stop our code for a short period.

```
#include <B4.h>

B4 myB4;
String assemblyProgram = "LOAD(5);ADD(4);WRT();SUB(2);WRT()";
long randNumber;

void setup()
{
    Serial.begin(9600);
    myB4.assembler(assemblyProgram);
    myB4.programB4();
}

void loop()
{
    randNumber = random(5000, 20000); // generates a random number
between 5000ms and 19999ms
    myB4.clockCycle(); // perform a ClockCycle
    delay(randNumber); // wait for the number of ms.
}
```

Upload the code to your B4 and observe what happens.

The B4 will 'randomly' perform a clock cycle. You can experiment with the random values and make them larger or smaller. The larger, they are, the more irregularly the events appear, making things look confusing from the perspective of the unsuspecting user.

Hack 2: Randomly changing the Data RAM

For this hack, we need to take control of the pins 6,7,8,9 and 12. Pins 6-9 provide data to the Data RAM and pin 12 activates the Automatic programmer.

```
#include <B4.h>

B4 myB4;
String assemblyProgram = "LOAD(5);ADD(4);WRT();SUB(2);WRT()";
long randNumber;

void setup()
{
  Serial.begin(9600);
  myB4.assembler(assemblyProgram);
  myB4.programB4();
  pinMode(6, OUTPUT); //bit 0 of the Data RAM
  pinMode(7, OUTPUT); //bit 1 of the Data RAM
  pinMode(8, OUTPUT); //bit 2 of the Data RAM
  pinMode(9, OUTPUT); //bit 3 of the Data RAM
  pinMode(12, OUTPUT);
}

void loop()
{
  randNumber = random(5000, 10000); // generates a random number
  between 5000ms and 19999ms
  digitalWrite(A5, HIGH); // activate automatic programmer

  digitalWrite(6, HIGH); // write bit 0 of the Data RAM
  digitalWrite(7, HIGH); // write bit 1 of the Data RAM
  digitalWrite(8, HIGH); // write bit 2 of the Data RAM
  digitalWrite(9, HIGH); // write bit 3 of the Data RAM

  digitalWrite(12, LOW); // write cycle, part 1
  digitalWrite(12, HIGH); // write cycle, part 2
  delay(1000); // keep the LED of the Automatic Programmer on.
  Deactivate this line after testing.
  digitalWrite(A5, LOW); // deactivate automatic programmer
  delay(randNumber); // wait for the number of ms.
}
```

How does it work?

- Firstly, we set the pins to Output, so that we can write to them later. We do this in the setup () function, because we only need to do this once.
- Then, we do the recurring tasks inside the loop() function

1. generate a random number as value for the delay
2. activate the automatic programmer
3. set the values of the data bits. In this example, we set them all to HIGH
4. we then keep the automatic programmer active for 1000ms
5. we deactivate the Automatic Programmer
6. finally, we wait until we do this again.

The 1000ms delay in step 6 is only there to show you that the code is working. When you are happy that it works correctly, you can deactivate this line and upload the code again. The code runs so quickly that our human eyes are not fast enough to see the Automatic Programmer's LED switching on. The unsuspecting user will think that the Automatic programmer is inactive the whole time. Tricked ya !

You see that our hack is completely bypassing the B4 library. And there is not much the library can do to prevent our hack.

As an extension, you can change the code to choose the data RAM values randomly, rather than 1111 as we've done.

As you know understand these two hacks you can tackle the hacks 3 and 4 yourself.

Further Reading

Below we have listed some really good resources that we used during the design of the B4. We very much recommend reading them.

Charles Petzold, CODE The Hidden Language of Computer Hardware and Software, 1999

<http://www.charlespetzold.com/code/>

Logic Gate: https://en.wikipedia.org/wiki/Logic_gate

Digital Logic Gates: http://www.electronics-tutorials.ws/logic/logic_1.html

Flip Flops: [https://en.wikipedia.org/wiki/Flip-flop_\(electronics\)](https://en.wikipedia.org/wiki/Flip-flop_(electronics))

History of Logic: https://en.wikipedia.org/wiki/History_of_logic

Transistor: <https://en.wikipedia.org/wiki/Transistor>

Troubleshooting

Every good experiment has the potential for failure. This is usually the moment when we learn something new. Below is a list of the typical errors and their solutions.

| Symptom | Solution |
|---|---|
| Green light of a module is off | Check if power cable is connected |
| | Check if the wires at the power cable plugs are fully inserted. Change cable. |
| Unexpected behaviour. Odd output of the modules. Looks erratic. | Check if all wires are properly connected. Tick them off one by one on the schematic of the corresponding experiment. |
| | Check if the wires at the data cable sockets are fully inserted. Change cable. |
| | Have you inserted the correct module? Check! |
| All lights are off | Connect USB cable to a computer, USB power outlet or USB battery. |
| | There may be a short circuit, usually cause by a power cable. Disconnect all power cables from the Program Counter and check if the Program Counter's green LED comes on. If yes, carefully connect one module after the other. |

Still got problems? Email us at: enquiries@digital-technologies.institute.

Appendix A: Programming Table Template

You can photocopy this table and use it to design and document your own programs for the B4.

Name of the Program _____

Author(s): _____

| | Data RAM | | | | Program RAM | | | | Description |
|---------|----------|---|---|---|-------------|-----|-----|-----|-------------|
| Step # | 3 | 2 | 1 | 0 | SUB | WRT | SEL | USR | |
| Step 15 | | | | | | | | | |
| Step 14 | | | | | | | | | |
| Step 13 | | | | | | | | | |
| Step 12 | | | | | | | | | |
| Step 11 | | | | | | | | | |
| Step 10 | | | | | | | | | |
| Step 9 | | | | | | | | | |
| Step 8 | | | | | | | | | |
| Step 7 | | | | | | | | | |
| Step 6 | | | | | | | | | |
| Step 5 | | | | | | | | | |
| Step 4 | | | | | | | | | |
| Step 3 | | | | | | | | | |
| Step 2 | | | | | | | | | |
| Step 1 | | | | | | | | | |
| Step 0 | | | | | | | | | |

Appendix B: Fun Algorithms

In this section we collect some of the interesting problems that people have solved with the B4 computer. We start with a fun algorithm that students have suggested.

B.1 Fairly Sharing Chocolate

As many of us agree, there is no such thing as ‘too much chocolate’. Recently, six students were given a package of Merci chocolates. As you might know, it contains 16 small bars of chocolate, two from each type. So there are eight different types of chocolate. But how do we distribute the chocolate most fairly amongst the students?

Appendix C: Solutions

Here are the solutions to the tasks from the different chapters in this book.

| Exercise 1.1 | | Solution |
|--------------|---|---|
| ? | What is the decimal value of 1111? | $8+4+2+1=15$ |
| | What is the decimal value of 0110? | $4+2=6$ |
| | What is the decimal value of 1010? | $8+2=10$ |
| | What is the binary value of decimal 15? | 1111 |
| | What is the binary value of decimal 12? | 1100 |
| | What is the binary value of decimal 9? | 1001 |
| | How can you easily spot an odd binary number? | Odd numbers always end with a 1. (and even numbers with a 0). |

| Exercise 2.1 | | Solution |
|--------------|---|--|
| ? | What is $0101 + 1010$? | 1111 ($5+10=15$) |
| | What is $0010+0010$? | 0100 ($2+2=4$) |
| | What is $0111+0001$? | 1000 ($7+1=8$) |
| | What is $1111 + 0001$? Why are all the Adder's LEDs off? | 10000 (16). This is a 5 bit number. All LEDs are off because the Adder can only work with 4 bits. It is simply 'blind' to the 5th bit. |

**Exercise
3.1**

Solution

Calculate in binary:

5 minus 2

0101-0010 is equivalent to 5 plus the binary complement of 2 plus 1.

```

0101
+1101
-----
10010
+   1
-----
10011

```

We ignore the 5th bit (because we only have a 4 bit computer) and the result is 0011 (3)

10 minus 0

1010 (obviously), but let's walk through the calculation. The binary complement of 0 is 1111

```

1010
+1111
-----
11001
+   1
-----
11010

```

We ignore the 5th bit (because we only have a 4 bit computer) and the result is 1010 (decimal 10)

15 minus 15

The binary complement of 15 (1111) is 0000

```

1111
+0000
-----
1111
+   1
-----
10000


```

We ignore the 5th bit (because we only have a 4 bit computer) and the result is 0000 (decimal 0)

?

| Exercise 3.1 | | Solution |
|-----------------|-----------------------------|--|
| | 2 minus 3. What do you see? | <p>The binary complement of 3 (0011) is 1100</p> <pre> 0010 +1100 ----- 1110 + 1 ----- 1111 </pre> <p>That's 15, not -1. Why? Our computer can only deal with positive numbers, so for it -1 is the same as 15. Again; that's not a bug. We simply haven't told our computer about negative numbers yet.</p> |

| Exercise 12.1 | | Solution |
|------------------|--|---|
| | <p>If you were to design a calculator, would you design WRT() to be B0110, or B0100?</p> | <p>In a calculator, we often want to use the output of one calculation as input of another, such as $5+4=9$ minus $2=7$. So WRT() should be B0110. This way, the 2-to-1 Data Selector would keep feeding intermediate results from the Data RAM to the Latch, which provides it to the Adder.</p> |

| Exercise 12.1 | | Solution |
|--|---|---|
|  | <p>If WRT() were B0100 and you wanted the B4 to run the following program "LOAD(5);ADD(4);WRT();SUB(2);". What would the output of the Latch be after program step 3 has been executed? Why is the result not 7? How can this be explained?</p> | <p>To answer this question, we can conduct an experiment by running the following program:</p> <pre>#include <B4.h> B4 myB4; int DataRAMContent[] = { B0101, B0100, B0000, B0010, B0000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, }; int ProgramRAMContent[] = { B0010, B0000, B0100, B1000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, B0000, }; void setup() { myB4.loadDataAndProgram(DataRAMContent, ProgramRAMContent); myB4.programB4(); } void loop() { }</pre> <p>After program step (3), the output of the Latch is B0010. This is not B0111 because the 2-to-1 Data selector is inactive and has channeled the output of the Adder (2) to the Latch. 2 is the result of 9 (from the Data RAM)+9 (from the Latch) after program step 2. The sum is 18=B10010. But since we have a 4 bit computer, the leading 1 is omitted. The result is B0010.</p> |

| Exercise 14.1 | | Solution |
|------------------|---|---|
| ? | <p>Compare your knowledge about transistors that form gates to what you know about biological systems. Can you identify similarities?</p> | <p>Transistors form gates, which form higher-level functions, such as arithmetics, memory, switching, etc. Similarly, cells form organs, which in turn form organisms.</p> |
| | <p>If transistors were made of mechanical parts that moved, rather than semiconductor materials, what disadvantages would this bring?</p> | <p>Mechanical parts are larger, consume more electricity and wear more quickly than semiconductors. Modern processors consist of billions of transistors. Let's assume we had a 1 billion transistor chip and we wanted to build it with relays, which are electromechanical switches. If each transistor were to be replaced with one relay, then we would require 1 billion relays. Let's further assume that 1 relay would require 1cm³ (the size of a sugar cube) of space and that we need another 1cm³ of space around each relay for wiring, etc.. So 2cm³ of space per relay. That would be 2 billion cm³. for all our 1 billion relays. That's 2,000,000,000 cm³ = 2,000 m³, or the equivalent of a cube with a side length of 12.6m, equivalent to a 4 storey building. If each relay required 50mA of current at 5V, then we'd need 50mA*1,000,000,000=50,000,000A. 50,000,000A*5V=250,000,000W, which is 250 Mega Watt. A smaller coal fired power plant produces 500 megawatt of electricity and burns 1.4 million tons of coal each year. We'd need half of this.</p> <p>In summary: If we could build such a relays computer, it would be the size of a 3 storey house, require half a coal-fired power plant and consume 700,000 tons of coal each year. This would be a tad too big for our pants. Not to mention the heat that the 700,000 tons of coal generate.</p> |

| Exercise 14.1 | | Solution |
|------------------|---|--|
| | <p>How much does it cost to manufacture a microprocessor? What would be the price per transistor for this microprocessor?</p> | <p>Let's pick the Xbox One processor which has 5 billion transistors. The Xbox console's retail price is about \$350. Let's assume that the cost of the processor is maybe \$50. So, the price per transistor is $\\$50/5\text{billion}=\\$0.000\ 000\ 01$ or 0.000001 cents, that's a thousands of a thousands of a cent per transistor. Let's put this into perspective: The print edition of the New York times newspaper has about 140,000 words. The average length of an English word is 5 letters. We conclude that the New York times contains $5*140,000=700,000$ letters. If it costs \$2 to make one copy of the New York times, then the cost per letter is $\\$2/700,000=\\$0.000\ 003$ or 0.000 3 cents.</p> <p>0.0003 divided by 0.000001 is 300.</p> <p>So, making a transistor in a chip is about 300 times cheaper than printing a letter in a newspaper.</p> <p>What if we estimated the price of the processor wrong? If it is less than \$50, then the ratio is greater than 300:1. If it is more than \$50, let's say \$100, then the ratio is 150:1.</p> |

Fairly Sharing Chocolate:

We assign each students a number from 1 to 6. With 16 chocolates available, each students gets 2 chocolates. The remaining 4 go to the teacher :-). $2 \times 6 = 12$, so there are 12 rounds in which students select one chocolate each. We number the rounds from 0 to 11 and assign rounds to students. There are many possible ways of assigning them. Below is one of them. Student1 draws first (Round1), followed by Student2, 3, 4, 5, and Student6. Student 6 hen draws twice, followed by Student5, 4, 3, 2, and finally Student1 draws her second piece of chocolate.

| Student1 | Student2 | Student3 | Student4 | Student5 | Student6 |
|----------|----------|----------|----------|----------|----------|
| Round0 | Round1 | Round2 | Round3 | Round4 | Round5 |
| Round11 | Round10 | Round9 | Round8 | Round7 | Round6 |

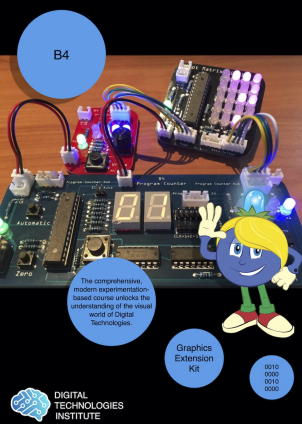
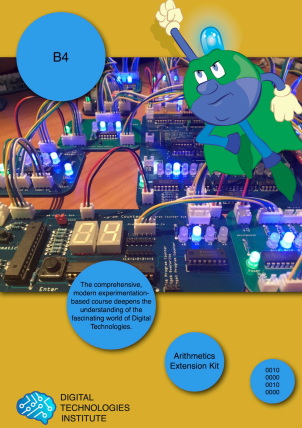
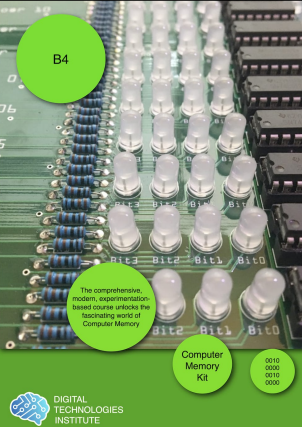
We want the Adder of the B4 to display the number of the student whose turn it is to select a chocolate.

| | Data RAM | | | | Program RAM | | | | Description |
|---------|----------|---|---|---|-------------|-----|-----|-----|-------------------------------------|
| Step # | 3 | 2 | 1 | 0 | SUB | WRT | SEL | USR | |
| Step 15 | | | | | | | | | |
| Step 14 | | | | | | | | | |
| Step 13 | | | | | | | | | |
| Step 12 | | | | | | | | | |
| Step 11 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Subtract 1. That's student1 |
| Step 10 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Subtract 1. That's student2 |
| Step 9 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Subtract 1. That's student3 |
| Step 8 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Subtract 1. That's student4 |
| Step 7 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Subtract 1. That's student5 |
| Step 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Add 0. Student 6 gets a second draw |
| Step 5 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Add 1. That's student6 |
| Step 4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Add 1. That's student5 |
| Step 3 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Add 1. That's student4 |
| Step 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Add 1. That's student3 |
| Step 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Add 1. That's student2 |
| Step 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | Load 1 |

Question: What other method can you think of to distribute the chocolates? How would a program look like that implements your method?

Appendix D: Extension Kits

The B4 Computer Processor Kit can be extended towards graphics, arithmetics and memory. The extension kits are available at <https://www.digital-technologies.institute/shop>

| | | |
|---|--|---|
| <p>Graphics Extension Kit</p> |  <p>The comprehensive, modern experimentation-based course unlocks the understanding of the visual world of Digital Technologies.</p> <p>Graphics Extension Kit</p> <p>0010 0001 0010 0000</p> | <p>This extension kit adds graphics output capabilities to the B4 Computer Processor. The Dot Matrix Display Module can output ASCII-style characters and symbols on a 4 by 5 LED matrix. In addition, students can program 16 of the LEDs separately and thus design their own graphics.</p> |
| <p>Arithmetics Extension Kit</p> |  <p>The comprehensive, modern experimentation-based course deepens the understanding of the fascinating world of Digital Technologies.</p> <p>Arithmetics Extension Kit</p> <p>0011 0001 0011 0000</p> | <p>This kit offers an exciting opportunity to dive deeper into the inner workings of a digital system by expanding the arithmetic capabilities of the B4 towards multiplication, division and beyond. In the process of implementing these capabilities from the ground up in a computer, students learn about loops, conditional jumps, data pointers and memory addresses.</p> |
| <p>Computer Memory Kit</p> |  <p>The comprehensive, modern experimentation-based course unlocks the fascinating world of Computer Memory.</p> <p>Computer Memory Kit</p> <p>0011 0001 0011 0000</p> | <p>The ability to remember is one of the fundamental functions of any computer. Memory is essential to perform algorithms. We have taken a deep look inside the black box and enlarged it. The result is an interactive memory kit that shows us the inner workings of a data RAM The RAM module. can be used as a replacement of the Data RAM module in the B4 Computer Processor kit.</p> |

Appendix E: Quick Reference Guide

| binary | decimal |
|--------|---------|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | 10 |
| 1011 | 11 |
| 1100 | 12 |
| 1101 | 13 |
| 1110 | 14 |
| 1111 | 15 |

| | | Output Program RAM Module | | | |
|-------------------|-----|---------------------------|---|---|---|
| Opcodes | | A | B | C | D |
| Subtract | SUB | 1 | 0 | 0 | 0 |
| Write to Data RAM | WRT | 0 | 1 | 1 | 0 |
| Select | SEL | 0 | 0 | 1 | 0 |