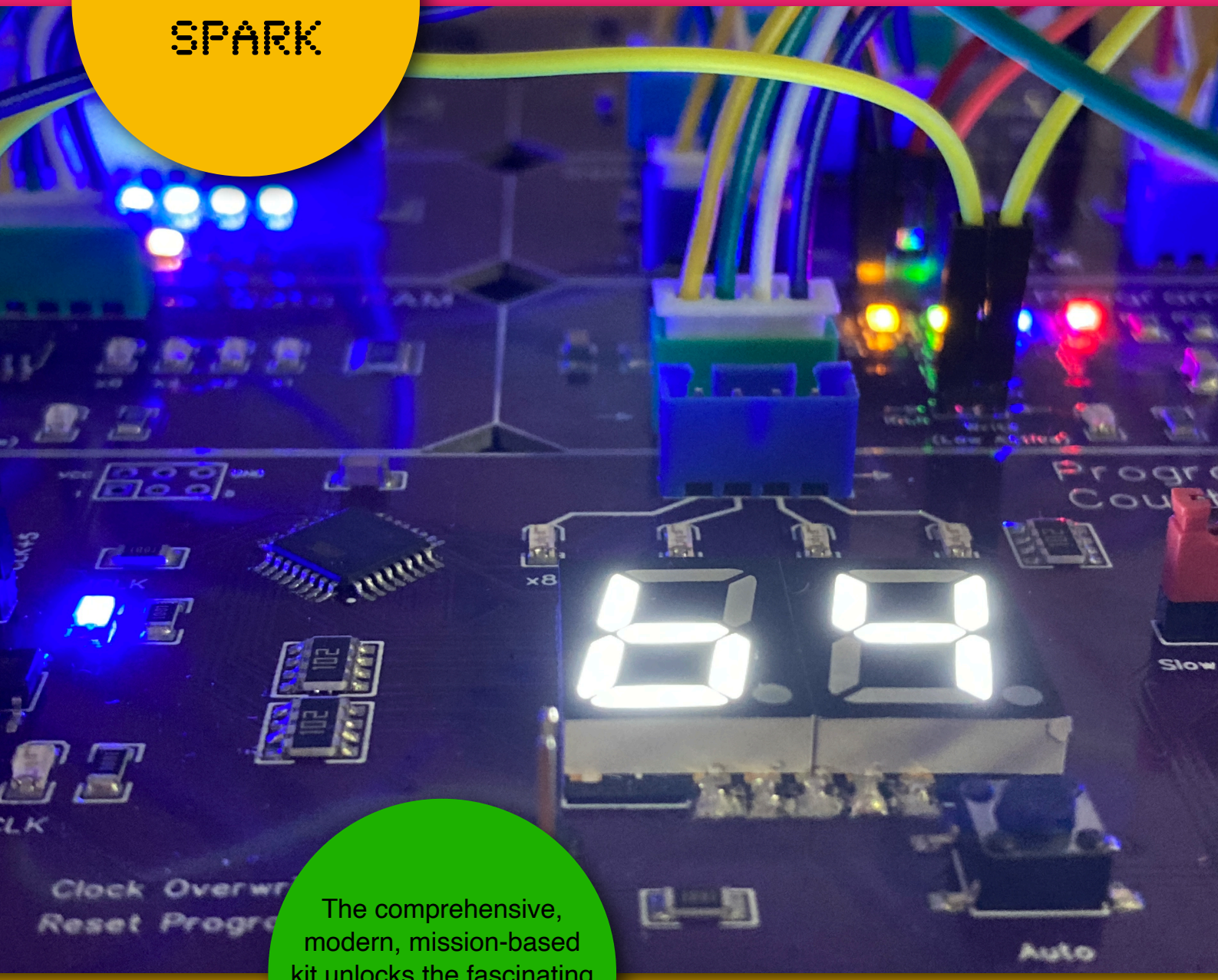
B4 SPARK



The comprehensive, modern, mission-based kit unlocks the fascinating world of Digital Technologies.

TO TRULY UNDERSTAND A COMPUTER, YOU MUST BUILD ONE.

0010
0000
0010
0101

DIGITAL TECHNOLOGIES INSTITUTE

“To truly understand a computer, you must build one.”

(anonymous)

Table of Contents

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 treating it with care and keeping 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 want 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 to 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 that runs 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 led to many improvements. Thank you!

The logic diagrams in this handbook have been designed using the Logicly program. We believe it is a great tool for quickly drawing and testing Boolean logic problems.

Dr. Karsten Schulz, CEO
The Digital Technologies Institute.

Included Parts

B4 Spark Core
Single board B4
2xVariable

4-Pin Wires
2-Pin Wires
1-Pin Wires
USB Cable
Student Handbook

B4 Spark Master Programmer
Automatic Programmer Arduino Shield
Arduino Uno compatible (required for the full function of the Automatic Programmer Arduino Shield)
B4 Arduino Library (available for download at http://www.digital-technologies.institute/downloads)

Power Consumption:
5V, 220mA (peak), 1.1W DC.

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

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

This handbook has been made with great care. If you find errors or have suggestions for improvement, please email us at enquiries@digital-technologies.institute.

Designed and manufactured in Australia

Hello Parents and Teachers

The B4 is an educational computer that supports students in their learning about digital systems. It has been designed from the ground up to support the new Australian Curriculum: Digital Technologies, as shown in the following table. A complete mapping of the B4 against the Australian Curriculum: Digital Technologies is available for download at: https://www.digital-technologies.institute/downloads

Digital systems
Data Representation
Acquiring, managing and analysing data
Investigating and defining
Generating and designing
Producing and implementing
Evaluating
Collaborating and managing
Privacy and security

The B4's design dates back to the 1970s, when early digital computers began to emerge. It follows similar design principles to some of the famous classic computers, such as the Apple I, the Altair 8800, or the Z-80. These principles remain valid today in modern computers, smartphones, and tablets. The B4 illustrates these principles and combines them with modern 21st-century Arduino technology, allowing students, parents, and teachers to explore the magic of creating a computer without needing a university degree.

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. However, this means that their parts have become so small that we can hardly see them with the naked eye. Therefore, the B4 has bigger components that you can easily see. Whereas the speed of modern computers is measured in billions of instructions per second, 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 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 the fundamentals of Digital Technologies. Still, you will be surprised at what can be done with a 4-bit computer.

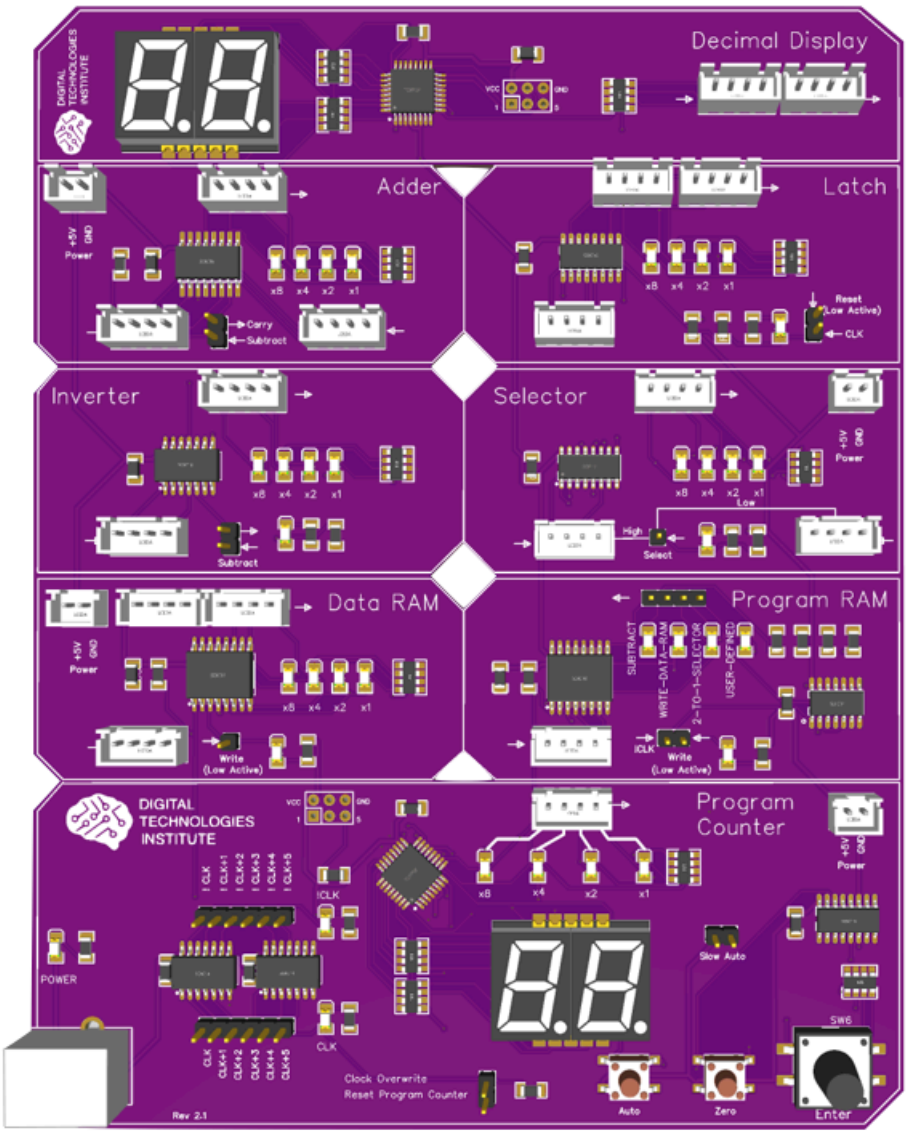
B4's Parts

Now let's see what's in the box: The B4 consists of modules that 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 for learning coding in detail. For convenience and to aid in the repetition and expansion of experiments, the B4 Master Programmer kit also features an Automatic Programmer, which requires an Arduino Uno or a compatible device.

Generally, each module receives its input through connectors at the lower end and provides an output through connectors at the top. All connectors are labelled with an **input arrow pointing to a connector or an output arrow pointing away from a connector**.

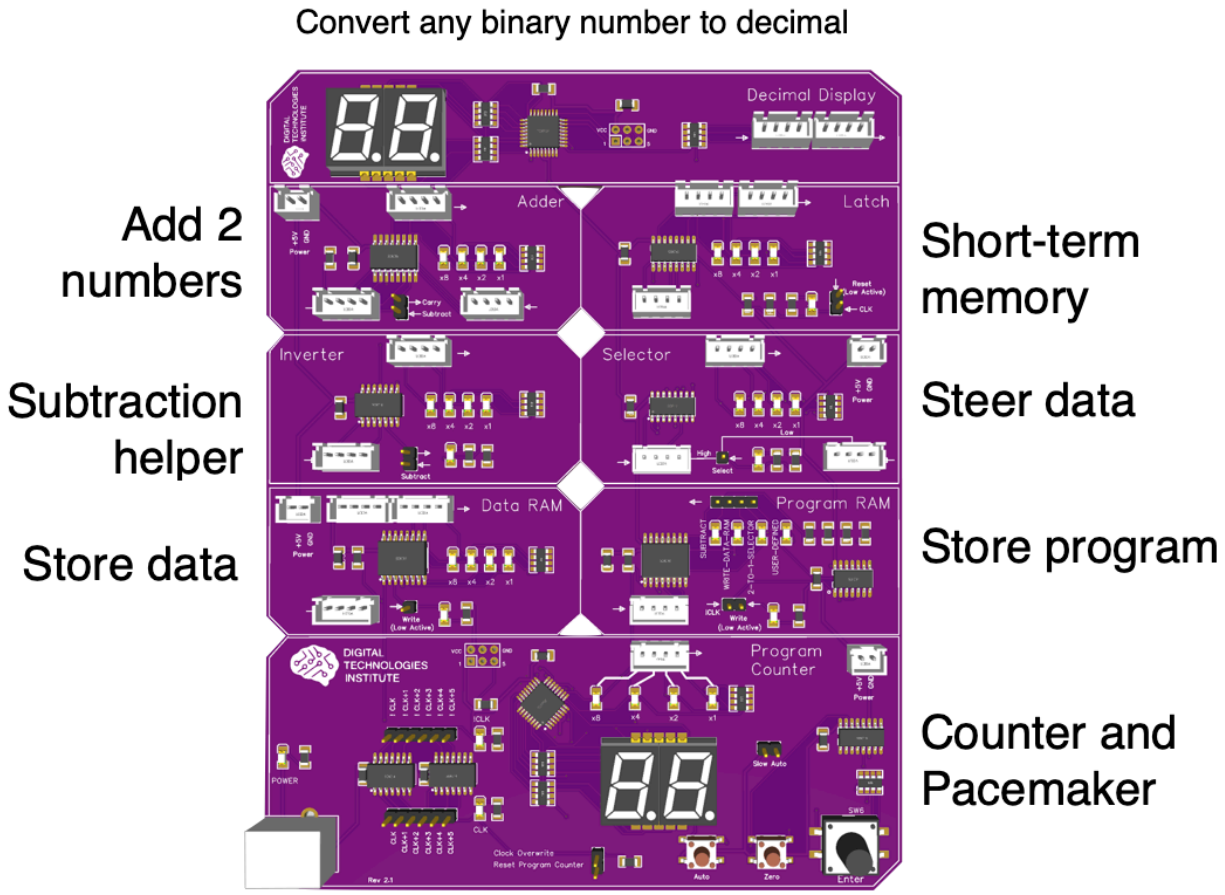
Output

Input



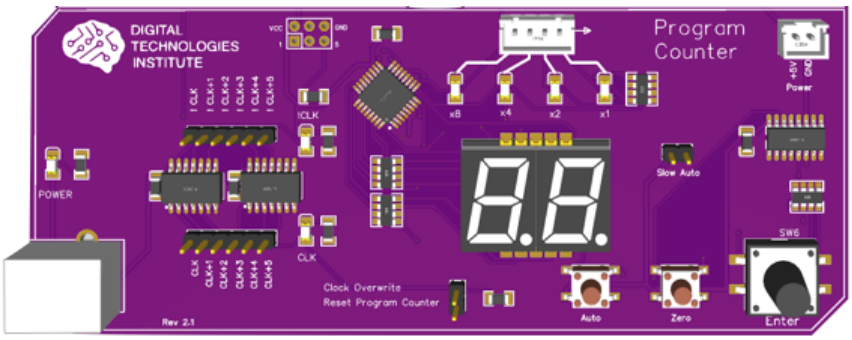
When we wire up the B4, we always connect the **output** of one module to the **input** of another module. We **never** connect two inputs.

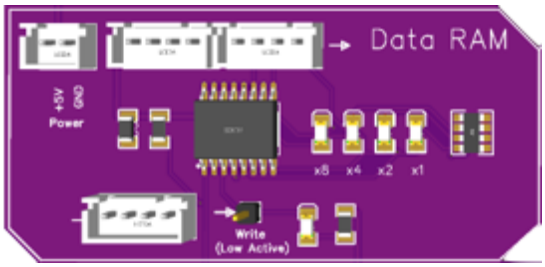
If you look closer at the board, you can see that white lines separate the board into eight departments. Each department has a special function. Let's explore them:



Core Modules

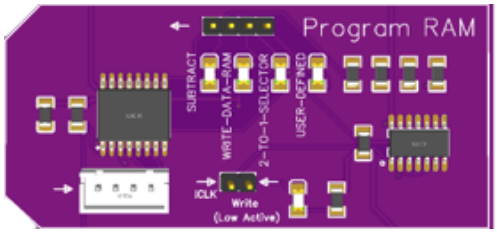
The **Program Counter**'s primary function is to count from 0 to 15. Every time you press the Enter button, the number on the display will grow by one. When it shows 15 and you press the button, the counter will tick over and start at 0 again. This number is important for the Data and Program RAM modules 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.

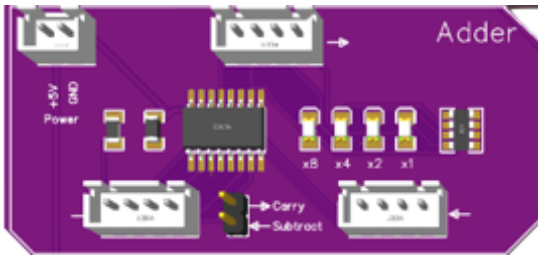




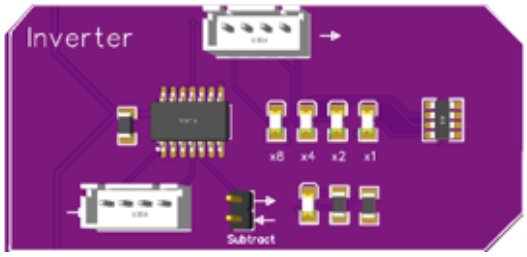
The **Data Random Access Memory (RAM)** holds the data that the program is working on. For example, it would hold two numbers that the program will be adding. The Data RAM has room for 16 numbers, which are 4 bit wide, thus representing the decimal numbers of 0 to 15. The Data RAM is a 16 by 4, or 16x4,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



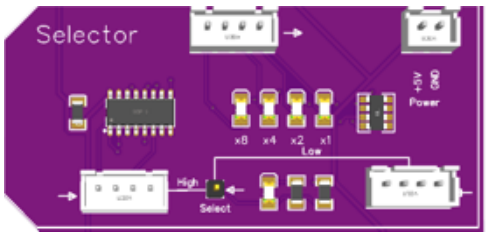


The **Adder** can add exactly two numbers.

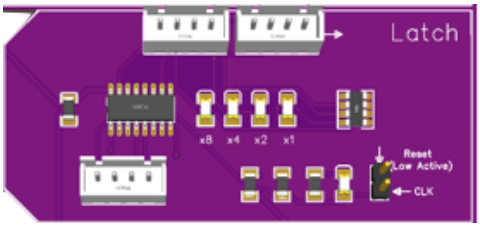


The **Inverter's** function is to help the Adder to subtract numbers. In the binary world, subtracting is the same thing as adding the binary complement, and then adding 1 to the result.

The **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.

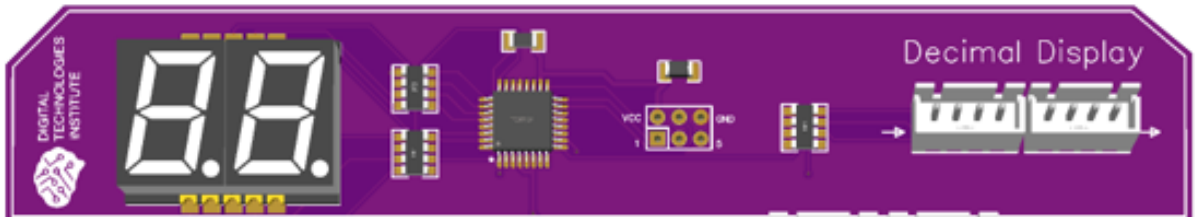


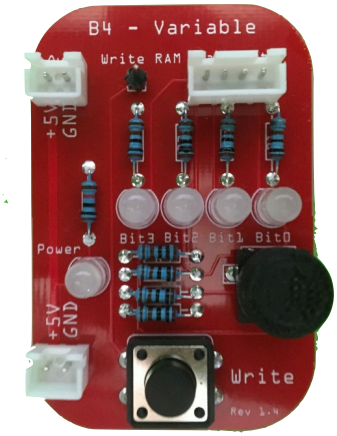
A **Latch** has the function of short-term memory in a computer. It stores (or 'buffers') some data before that data can be further processed. For example, in the B4, the Latch stores the first number so that a second number can be added to it.



Helper Modules

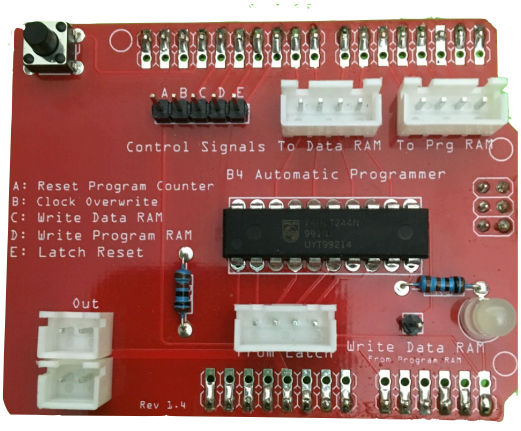
All computers internally work with binary numbers only. However, we humans are more familiar with decimal numbers. As you work with the B4, you will get used to 1's and 0's, and you will find it increasingly easy to remember that a 0111 is a decimal 7. The **Decimal Display** is a handy module that performs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



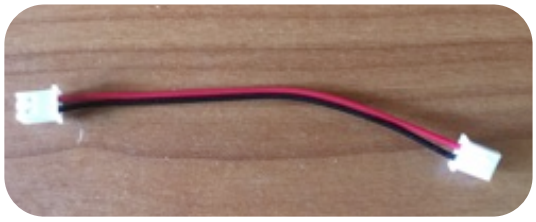
We now have a basic understanding of the modules of our B4. Don't worry if you haven't understood everything yet. We will revisit each module in more depth during the following experiments.

Wires and Connectors

To connect the B4 modules with the computer and with each other, the B4 comes with four types of wires. They are:

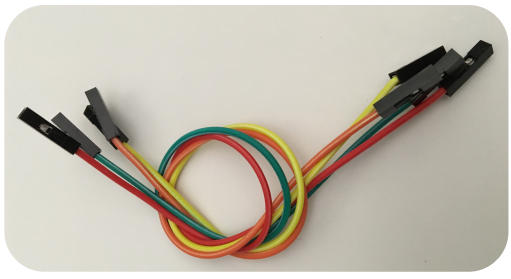
A **USB cable** to provide electricity from a power source to the B4's Program Counter module and from there, to all other modules connected to the Program Counter. You can connect the USB cable to a PC, Laptop, USB Hub, USB battery, or any other suitable 5V power source with a USB port.

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 from the output of one module to the input of another module.

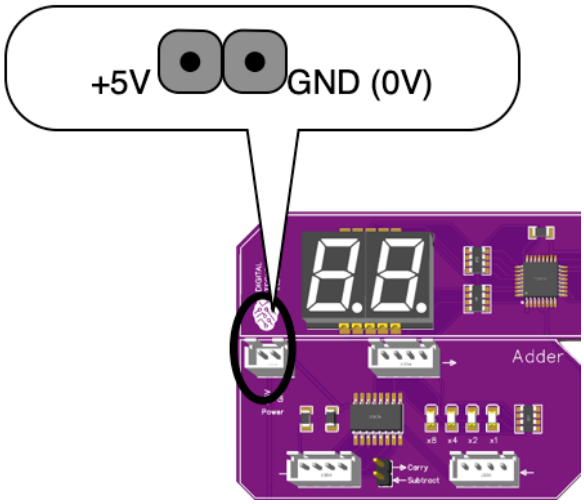
1 pin **control wires**. They transport operation codes and instruct some of the modules of the B4 to do special things, such as storing data. The 1 pin wires come in many different colours. However, they all work the same and their colour has no influence on their function



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.

A Word about Power

The B4 has an electric power distribution system. There are four white two-pin ports on the B4, and the Variables and Automatic Programmer have two.



+5V is on the left and GND (Ground, or 0V) is on the right. The power wires will automatically connect in the correct 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 that are not designed to fit, and as long as you don't drop the modules, step on them, or use them as a doorstop, 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 missions. Let's get started.

Exploration through Missions

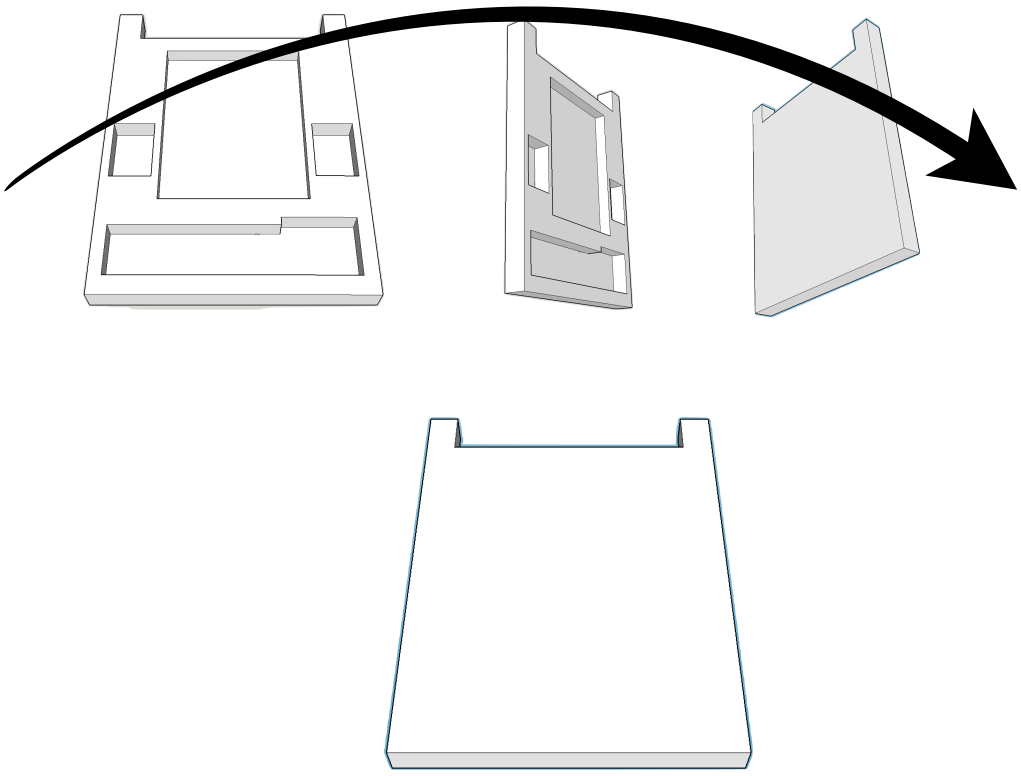
In this book, we conduct missions during which we will be plugging wires into the B4 modules, letting them work together and experimenting with data and hardware. On occasion, when the bell rings at the end of the lesson, we may not quite be finished with an investigation. So that we don't have to take all our good work apart (and start from scratch next lesson), the B4's packaging also serves as a storage tray. The foam insert features several cut-outs to protect the B4's modules 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 a mission 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.



Mission Overview

We have prepared several missions to help you become familiar with the B4 modules, their usage, and the functions they perform. You will learn how to combine modules so 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 communication.

We recommend that you take the missions 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.

Mission	Title	Objectives
1	The Heartbeat of the Machine	Bring your B4 to life by starting its heartbeat, learn to read its pulse in both human and machine language, and discover how it keeps every part in sync.
2	The B4 Learns to Add	Give your B4 its first real thinking skill — adding two numbers.
3	The B4 Learns to Subtract	Teach your B4 how to take one number away from another
4	The B4 Learns to Remember	Give your B4 a short-term memory so it can temporarily store information while working on a problem.
5	The B4 Builds Its Memory Tower	Upgrade your B4's memory from short-term storage to long-term storage using the Data RAM.
6	The B4 Learns to Route Traffic	Teach your B4 to choose which data stream gets through.
7a-c	Adding Three Numbers	3 missions in which we add numbers a) by hand, b) automatically and c) with the Data RAM
8	Automating the Selector with Program RAM	Learn how to use Program RAM to control the Selector automatically.
9	Program Tables	Learn how to write programs in a clear table format, making it easier to describe, run, and share your programs.
10	Subtraction with the Inverter	Extend your program table so the Program RAM can control the Inverter. This allows your computer to perform subtraction automatically.

Mission	Title	Objectives
11	Storing Calculation Results in the Data RAM	Learn how to let Program RAM control writing into Data RAM.
12	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.
13	Program Language Design	Investigating a compact and conceptional notation. Assembly language and the role of an assembler. Shortening of the software design and testing cycle.
14	On the Role of Timing	Fundamental role of precise timing of the communication of the B4 modules.
15	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.
16	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.

🟢 Part 1 – Building the Foundations

In the first part of your journey, you'll explore the basic building blocks of a computer. One by one, you'll discover how modules work and how they can be connected. Step by step, you'll move from adding two numbers to chaining calculations and finally using Data RAM as memory.

By the end of Part 1, you'll understand the "vocabulary of hardware and binary numbers": how data is represented and how it flows through the B4, how temporary storage works, and how simple arithmetic can be performed. You've essentially built a very basic calculator — but one where you can see and control every single bit.

Mission 1 – The Heartbeat of the Machine

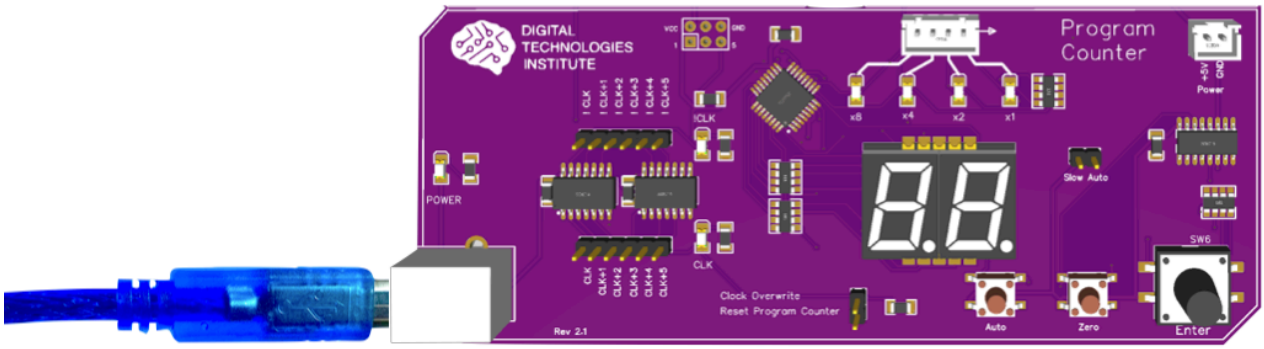
Objective
Bring your B4 to life by starting its heartbeat, learn to read its pulse in both human and machine language, and discover how it keeps every part in sync.

The Story
Every living thing has a heartbeat — and so does your computer. On the B4, the Program Counter is the heart. Each beat sends a signal pulsing through the system, telling every part when to work. Without it, the machine is silent and lifeless. In this mission, you'll plug in the heart, make it beat, and learn to read its rhythm in two languages: machine speak (binary) and human speak (decimal). You'll also find out how this simple beat secretly controls every other module in the B4.

Step 1 – Waking the Heart

Connect the Program Counter module to your computer using the USB cable. A white display will flash b4 — the machine's way of saying "Hello."

On the Program Counter, find the Enter button. Press it once. The display changes to 0. Press it again and it changes to 1. Every press increases the number by one. It's counting — but not just for you. Every press also sends an invisible electrical "beat" through the B4.

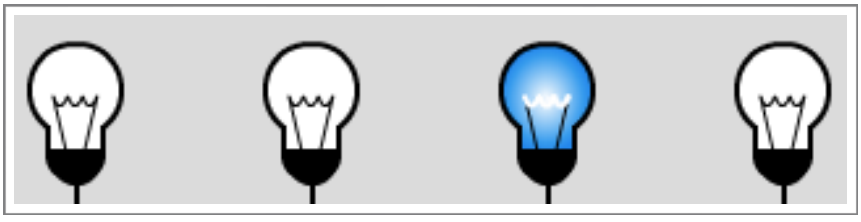


Setup of Mission 1

Step 2 – The Language of Light

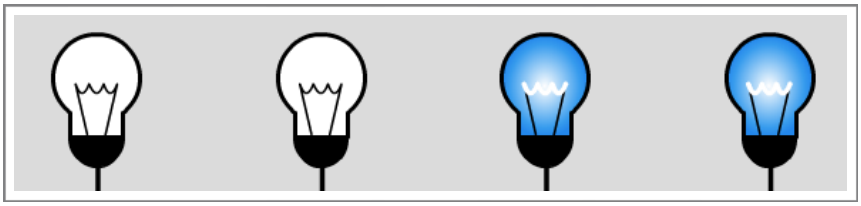
At the top of the Program Counter are four tiny LEDs. They light up in different patterns each time you press Enter. This is binary — the language your machine speaks.

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:



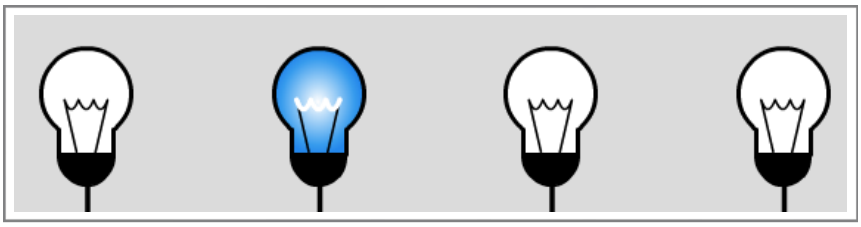
0010 in binary is a decimal 2

When 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 continue to press 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?

Each LED has a value: the one on the far right is worth 1, the next is worth 2, then 4, and the one on the far left is worth 8. If the LEDs show 1101, you add 8 + 4 + 0 + 1 = 13. Binary may look mysterious, but it's just another way to write numbers — and your B4 can only count up to 15 (binary 1111) before looping back to 0.

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.

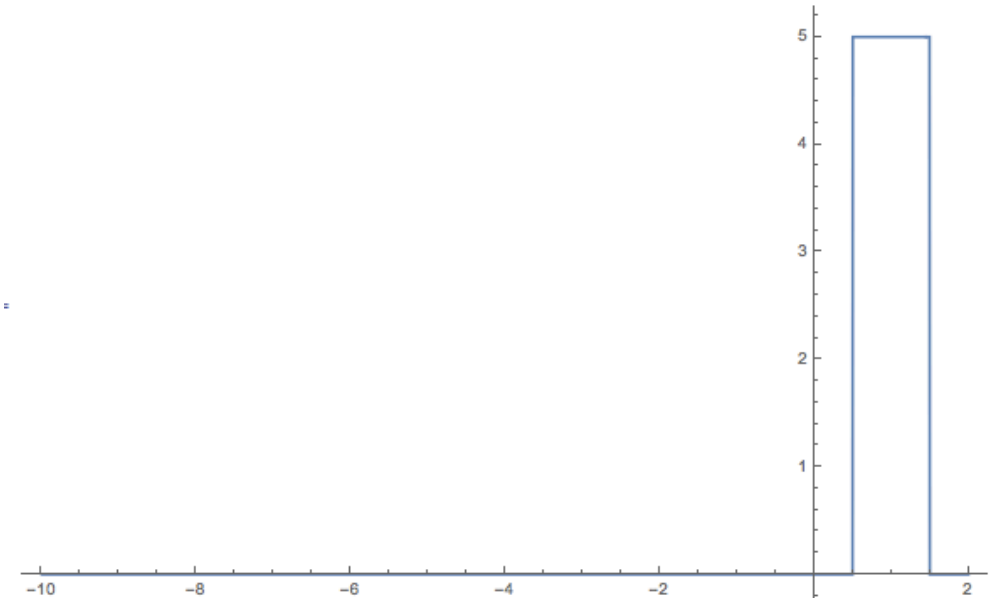
Checkpoint Challenges 1	
?	What is the decimal value of 1111?
	What is the decimal value of 0110?
	What is the decimal value of 1010?
	What is the binary value of decimal 15?
	What is the binary value of decimal 12?
	What is the binary value of decimal 9?
	How can you easily spot an odd binary number?

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.

Step 3 – The Beat of the Clock

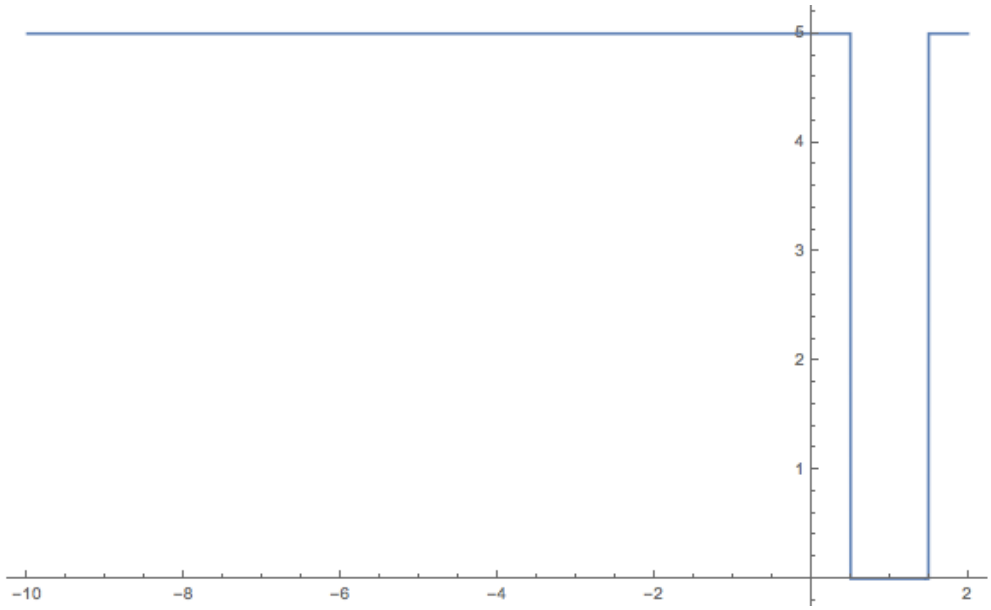
Every time you press Enter, the Program Counter sends out a clock signal (CLK). Think of it as the drumbeat in a song — every other module listens for that beat so they know exactly when to do their part.

The signal jumps from 0V to 5V, resides there for a short period, and then drops back to 0V. We refer to 0V as **LOW** and 5V as **HIGH**. In the B4, the CLK signal remains HIGH as long as the Enter button is held down. When you release the Enter button, the CLK signal returns to LOW.



CLK signal

Some modules prefer the opposite rhythm, so the Program Counter also produces an inverse clock, called NOT-CLOCK (!CLK), which is “on” when CLK is “off.” Later, you’ll use both signals to control different parts of the B4. 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.



Inverse CLK Signal: !CLK

You might ask why we need both, CLK and !CLK? The answer is that some of the computer circuits require a positive 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.

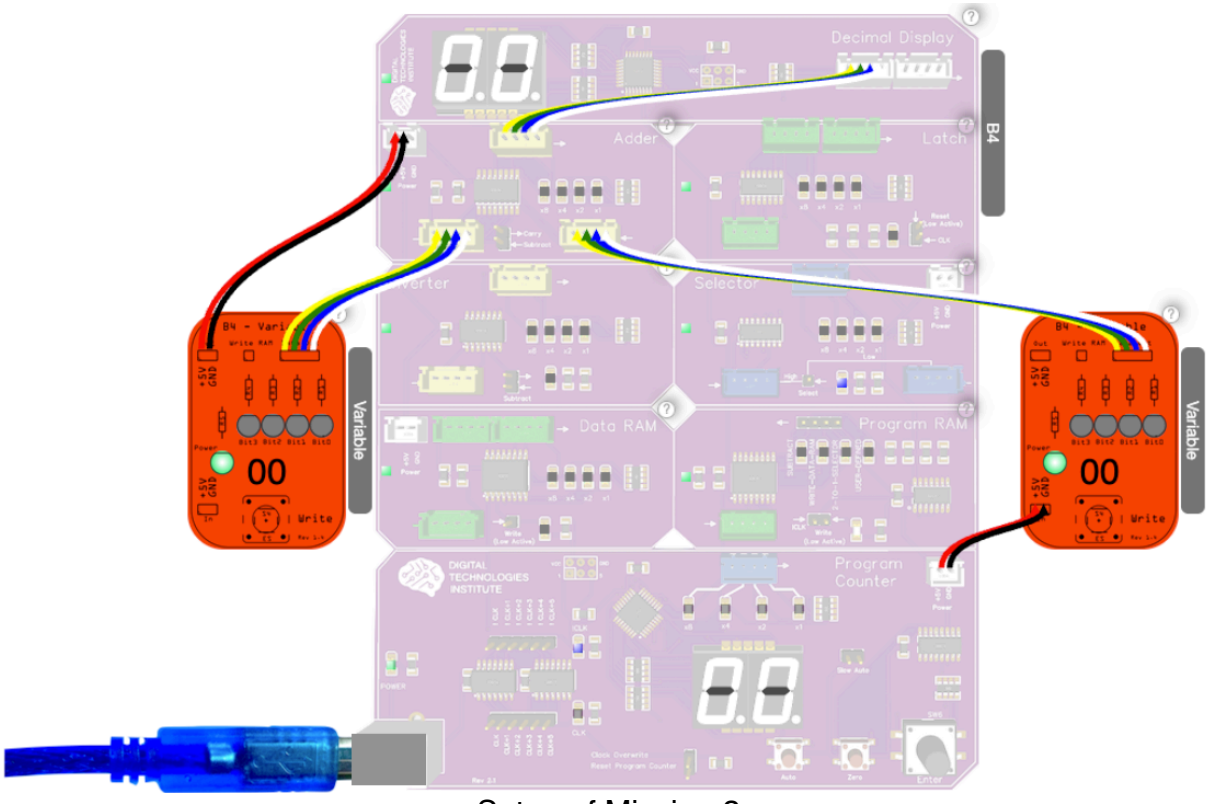
We’ll talk more about CLK and !CLK when we cover the Latch and RAM modules in the following experiments, and also at the end of the book in mission 14, which is about timing.

Mission 2 – The B4 Learns to Add

Objective
Give your B4 its first real thinking skill — adding two numbers. You'll meet the Adder, learn how it works, and see why addition is the foundation of almost everything a computer can do.

The Story
Your B4's heart is now beating, but right now it's just counting time — it doesn't know how to do anything useful with numbers. A heartbeat without a brain doesn't get you far. It's time to give the B4 its math brain — the Adder module. This is the part of your computer that can take two numbers and combine them into one. It might sound simple, but here's the twist: once a computer can add, it can also subtract, multiply, divide, and do just about any calculation you can imagine. Addition is the bedrock of computing.

Build
Use: • Program Counter (for power only this time) • Adder module • Two Variable modules (these are like your "number knobs") • Decimal Display (so you can see the result in human numbers) Connect them as shown in the diagram. The Variables will feed numbers into the Adder. The Program Counter will simply power the system.



Setup of Mission 2

Experiment

1) Turn the knobs on both Variables until their LEDs are all off.
2) Slowly turn the knob on one Variable so that just the rightmost LED is lit — that's binary 0001, or decimal 1.

What is the output of the Adder? The right LED on the Adder will light up.

Understand

That's because 0+1 is 1.

Turn the knob of the other Variable to show the 0001 LED pattern.

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.

```
 0001     0101
+0001    +0110
--------  --------
 0010     1011
```

Checkpoint Challenges 2	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?

Mission 3 – The B4 Learns to Subtract

Objective

Teach your B4 how to take one number away from another by introducing it to the Inverter module and the clever trick of using complements.

The Story

Your B4 can now add, but there's a problem — life isn't all about getting more. Sometimes, you need to take things away. Computers don't have a built-in "minus" button the way your calculator does. Instead, they use a clever hack: they turn subtraction into addition.

How? By flipping all the bits in the second number — making what's called the binary complement — and then adding 1. It's like if you wanted to find out how far someone is from the finish line, you could measure from the end backwards instead of walking all the way there.

This trick is so common that it's built right into computer hardware. On the B4, the Inverter module does the job of flipping the bits for us.

Step 1 – Meet the Inverter

The Inverter is like a tiny mirror for binary numbers. If a bit is 0, it flips it to 1; if it's 1, it flips it to 0.

Example: 0011 becomes 1100.

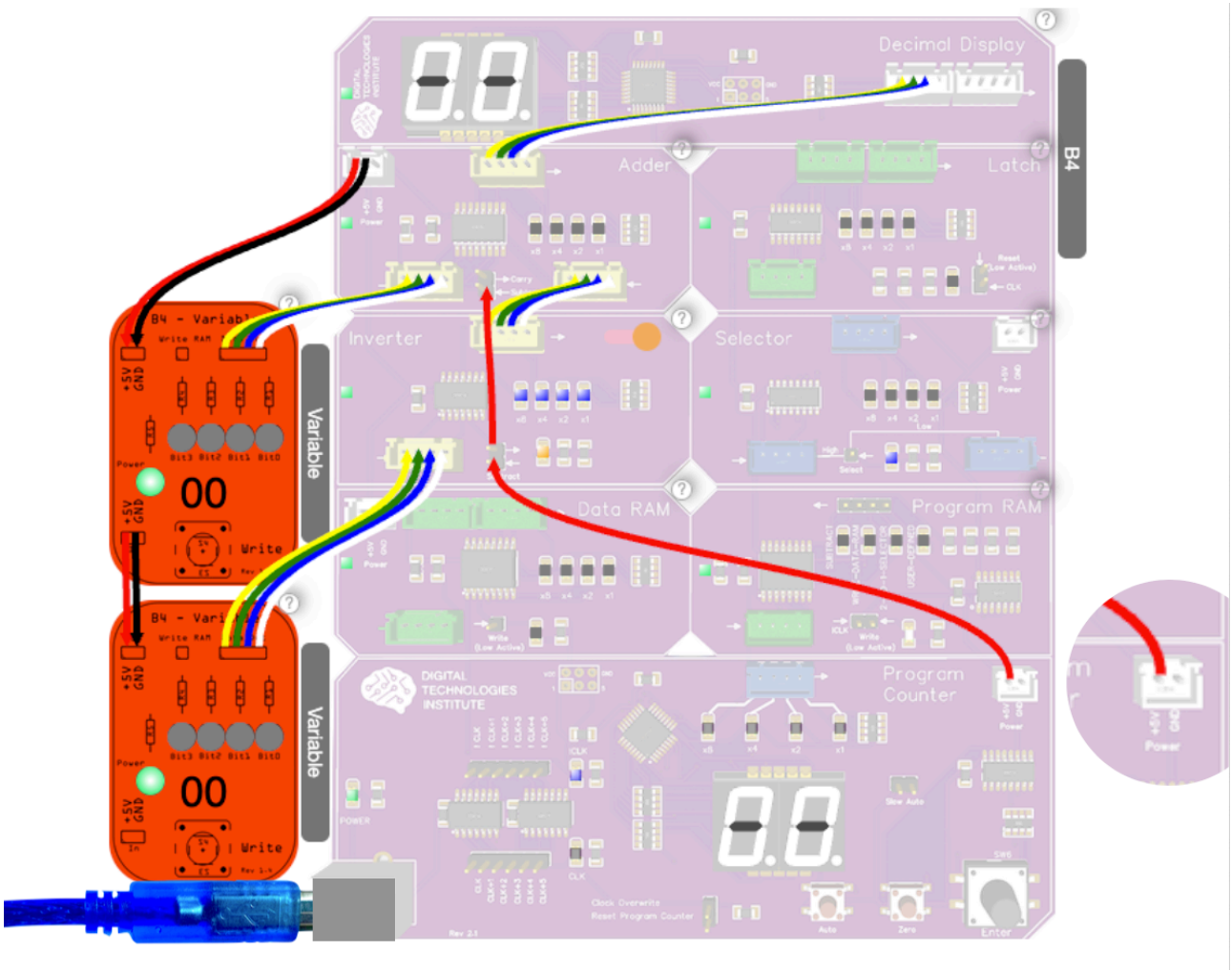
Step 2 – Building the Subtraction Circuit

You'll need:

- Program Counter (for power)
- Adder
- Two Variable modules
- Inverter
- Decimal Display

1) Use the setup from mission 2. In the image below, we have rearranged one of the variables so that you can see the wiring between the modules better. But you can keep the variable where it was and just use longer wires, if you like.
2) Use the Inverter module between the bottom Variable and the Adder
3) Connect a 1-pin wire from the Subtract Out Pin of the Inverter to the Subtract In Pin of the Adder module.
4) Connect another 1-pin a wire from the Inverter's *Subtract In* pin to the +5V Pin.
5) Set both Variables to 0000 (all LEDs are off).

The following diagram shows the setup of this mission.



Setup of Mission 3

Step 3 – First Subtraction
Let's calculate 4 – 3. • On the top Variable, set 0100 (decimal 4). • On the bottom Variable, set 0011 (decimal 3). What is the output of the Adder? What is the output of the Inverter? The Adder shows 0001. We conclude that 4 - 3 = 1 YEA! The Inverter Displays the complement of 0011, which is 1100.

Step 4 – Why This Works

When the Inverter is active, it flips 0011 into 1100 — the binary complement of 3. The Adder then does:

```
  0100  (4)
+ 1100  (complement of 3)
  ———
 10000
```

That's a 5-bit number, but our B4 can only handle 4 bits, so it chops off the extra "1" at the left, leaving 0000.

But wait — that's not the answer yet. We need to add 1 to complete the subtraction trick:

```
  0000
+ 0001
  ------
  0001  (decimal 1)
```

And there's your answer: 4 – 3 = 1.

The complement-and-add method is used in nearly every modern computer, from laptops to smartphones. Instead of building separate circuits for addition and subtraction, engineers use the same Adder for both — saving space and parts.

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

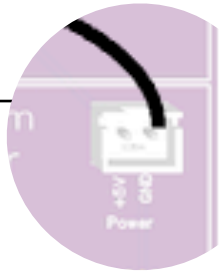
Step 5 – Finding +1 in hardware

You may now wonder: "How does the Adder know when to add 1? Do you see the 1-pin wire that connects the Adder and the Inverter? If you pull it out, then the +1 goes away and the Decimal Display will show the output of the Adder as 0. Try it. Reinsert the wire when you are done to restore correct subtraction.

Step 6 – Deactivate the inverter

When you move the endpoint of the 1-pin wire from +5V to GND, then this deactivates the inverter. Try it! The Inverter will then not flip the bits and the Adder will then just add the numbers, rather than subtracting them.

So: 4+3=7



A simple electrical signal makes the B4 do subtraction. Imagine if a program could send this signal automatically. We will explore this in mission 8.

Step 7 – Going Further
If you keep subtracting the same number from a starting value until you get below that number, you're actually doing division. Example: 15 – 5 = 10 10 – 5 = 5 5 – 5 = 0 → Done! (3 subtractions, so 15 ÷ 5 = 3)

Why This Matters
You've just taught your B4 how to think backwards. With addition and subtraction in its toolkit, it's ready for more advanced calculations — and you now understand a trick that's been hiding inside every computer for decades.

Checkpoint Challenges 3	Compute with your B4 and also on paper:
?	5 minus 2
	10 minus 0
	15 minus 15
	2 minus 3. What do you see?

Mission 4 – The B4 Learns to Remember

Objective

Give your B4 a short-term memory so it can temporarily store information while working on a problem. You'll meet the Latch and discover how it works alongside the clock signal.

The Story

Your B4 can now add and subtract, but it's got a big limitation — it can only work with the two numbers you give it right now. The moment you change one of them, the old number is gone forever.

Humans have the same problem when we try to calculate something without remembering the intermediate result. Imagine working out 3 + 8 + 1 without remembering that 3 + 8 = 11 — you'd have to start over every time.

Computers solve this problem with short-term memory. On the B4, this role is played by the Latch. The Latch can grab a number from its input and hold onto it until told to let go.

Step 1 – Meet the Latch

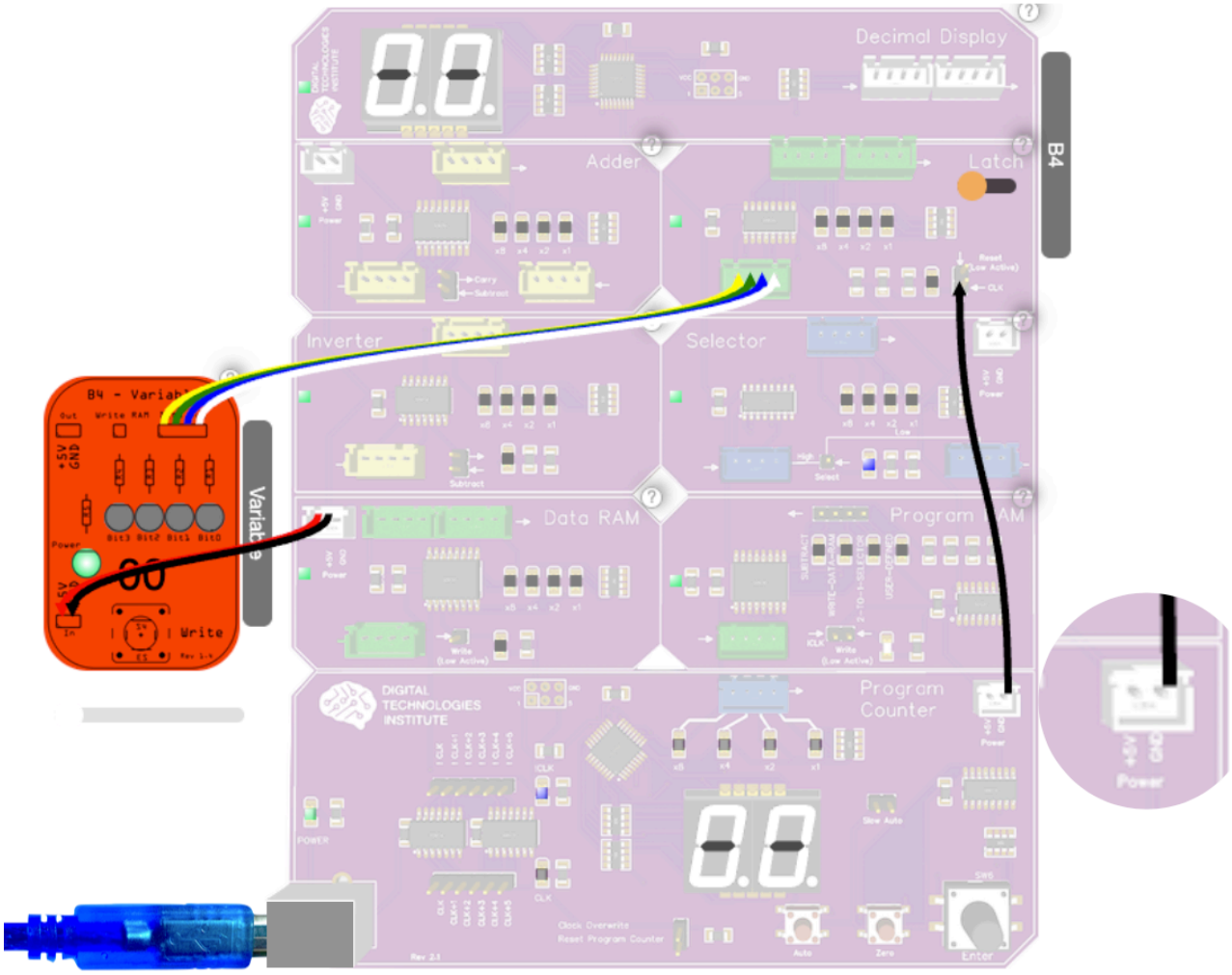
The Latch is like a tiny 4-room storage locker — one bit per room. When it gets an activation signal (from the clock), it takes whatever's at its input and locks it in. Until it gets another activation signal, it won't change, even if the input changes.

Step 2 - Building the Memory Circuit

You'll need:

- Program Counter (for power)
- Variable module
- Latch

Set up the modules as in the diagram for Mission 4. Connect a control wire from the Latch's CLK input pin to GND so it isn't activated yet.

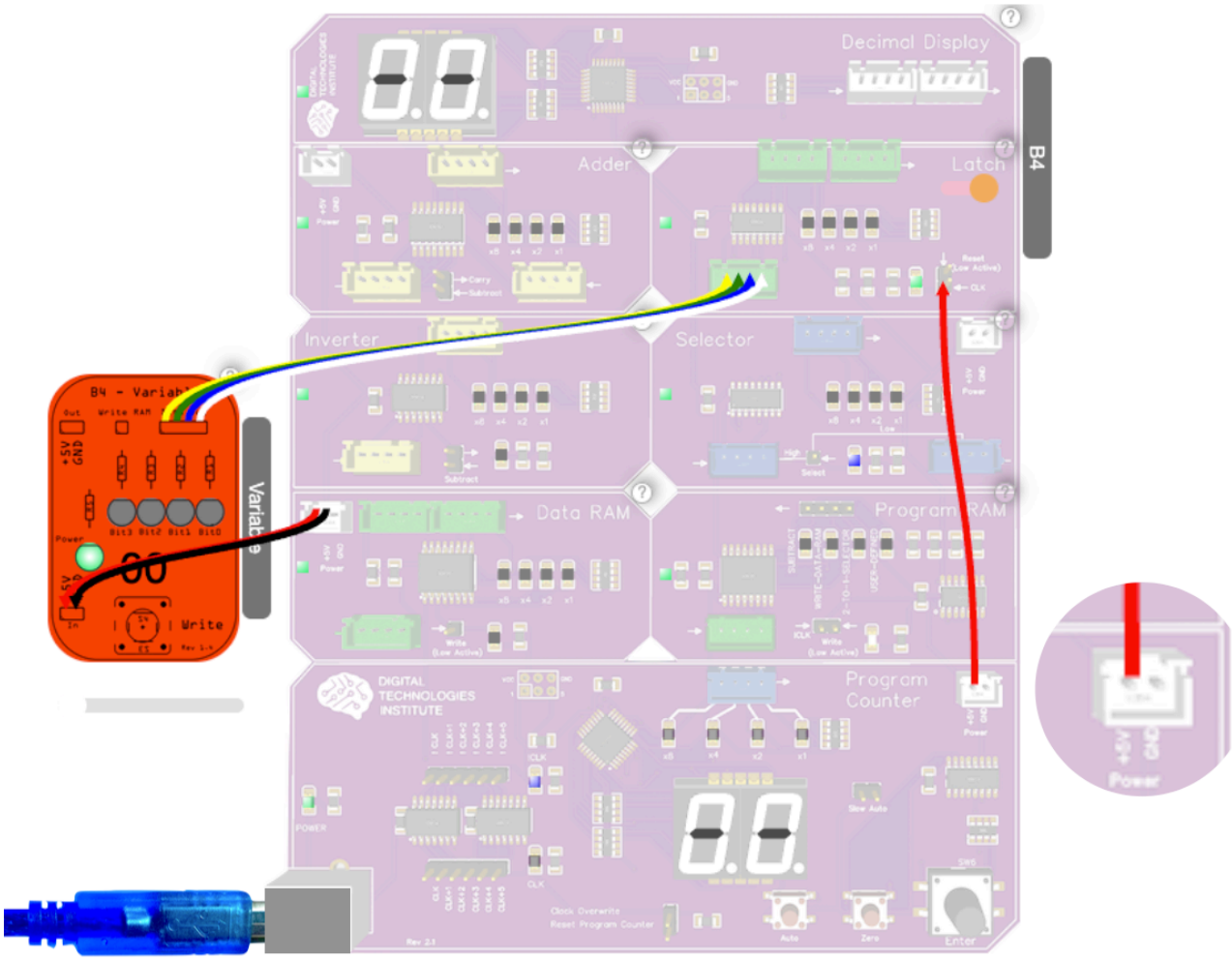


Setup of Mission 4, First Memory Test

Step 3 – First Memory Test
1. Set the Variable to 0011 (decimal 3). 2. Look at the Latch's LEDs — nothing changes. Why? Because the Latch is waiting for its "remember now" signal.

Why this doesn't work
The Latch is waiting for an activation signal. This is really important, as we need to tell the Latch *when* it should remember something.

Step 4 – Triggering Memory
Move the control wire from GND to +5V as shown in the next figure. The Latch now receives an activation signal. The LEDs on the Latch will instantly match the Variable — it has stored 0011.



Setup of Mission 4, Triggering Memory

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

Step 5 – Does It Hold?
While keeping the control wire on +5V, change the Variable to 0100 (decimal 4). The Latch stays on 0011.

How this works

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.

Step 6 – Changing Memory

Disconnect the control wire from +5V, then reconnect it. This creates a manual clock pulse.

Now the Latch updates to match the Variable's new value. This works — but in a real computer, constantly unplugging and reconnecting wires is not practical.

Planning for the next step

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 mission 1, that one of the functions of the Program Counter is the production of the CLK Signal?

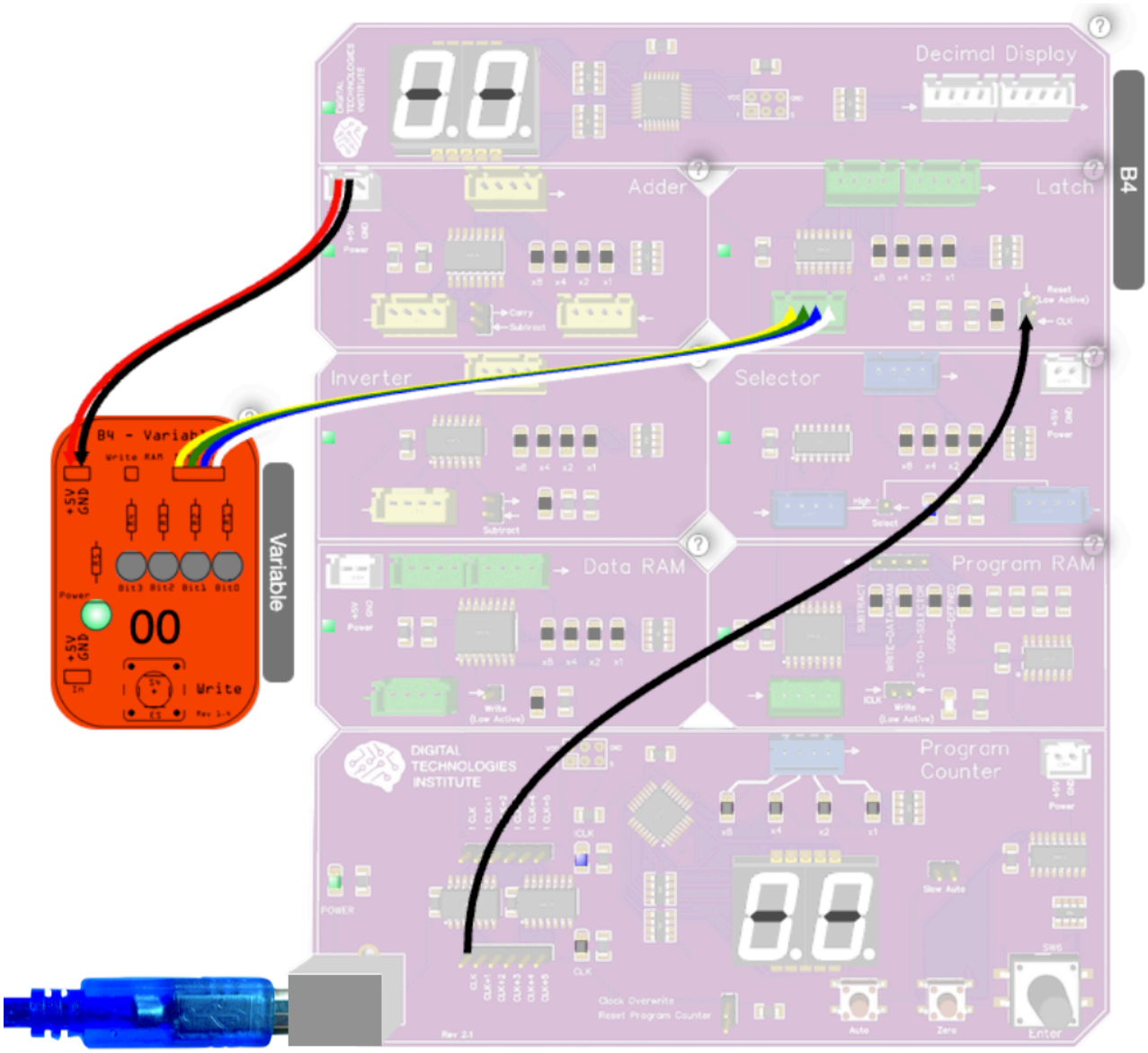
Step 7 – Automatic Memory

Connect the Latch's CLK In pin to the CLK output on the Program Counter, as shown in the next figure.

Now, every time you press Enter on the Program Counter, it *automatically* sends a clock pulse that tells the Latch to store the current value from the Variable.

Try it:

1. Set the Variable to 1000 (decimal 8).
2. Press Enter on the Program Counter.
3. The Latch updates to 1000 automatically.



Setup of Mission 4, Automatic Memory

Why This Matters
With a Latch, your B4 can remember results from one calculation and use them in the next. This is essential for adding more than two numbers, avoiding infinite loops, and building more complex programs.

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

Mission 5 – The B4 Builds Its Memory Tower

Objective

Upgrade your B4's memory from short-term storage to long-term storage using the **Data RAM**. You'll learn how to store data permanently (until the power is turned off) and how the Program Counter acts like an elevator to place data on the correct "floor."

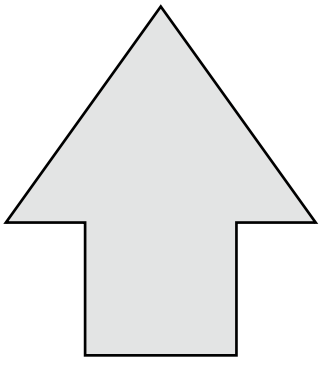
The Story

In the last mission, you gave your B4 short-term memory with the Latch — handy for holding onto information for a moment, but easily overwritten. If you want the B4 to remember something for the whole program (or until it's powered down), you need long-term memory. We call this Random Access Memory (RAM).

Step 1 – Meet the Data RAM

Think of the Data RAM as a tall apartment building with 16 floors. Each floor has 4 rooms (bits), and each room can hold either a 0 or a 1. You can store any 4-bit number (0–15 in decimal) on any floor you like. The Program Counter is like the building's elevator, moving up floor by floor to deliver or collect data. **We also call the floor an address**. To store data into the Data RAM Module, we first let the Program Counter 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				



Data RAM: 16x 4 bit.

Step 2 – Building the Long-Term Memory Circuit

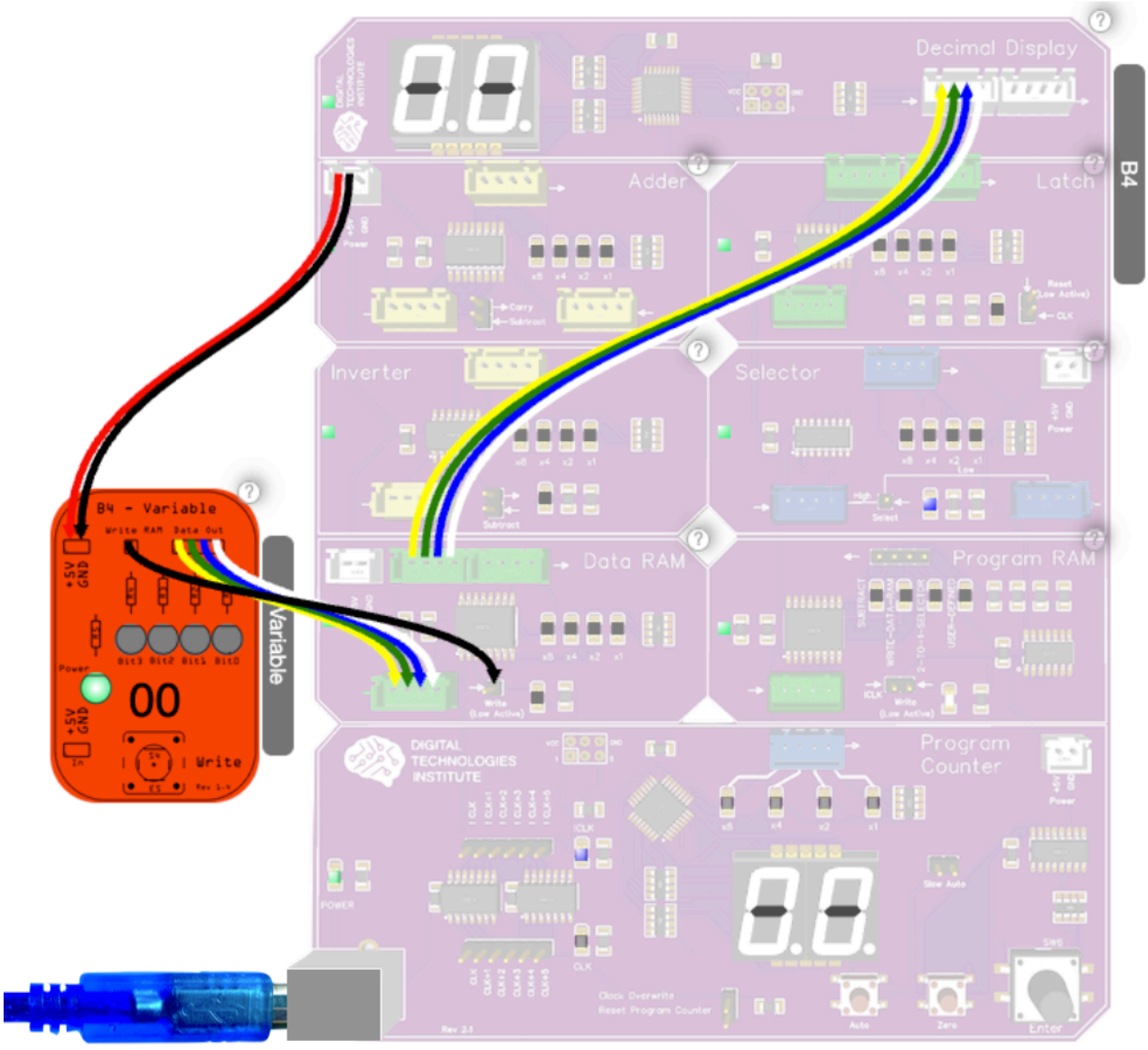
You'll need:

- Program Counter
- Data RAM
- Variable module
-

Connect them as in the diagram:

1) Variable Out → Data In on the Data RAM (this delivers the data to store).
2) Write button on the Variable → Write Data RAM In on the Data RAM (this is the "store now" command).

The Program Counter is already connected to the Data RAM and automatically tells it the address. This is hardwired on the B4's printed circuit board.



Setup of Mission 5

Step 3 – Storing Your First Data

1) Press Reset on the Program Counter to go to address 0 (0000).
2) Set the Variable to 1010 (decimal 10).
3) Press the button on the Variable — the Data RAM's LEDs will now show 1010 at address 0.
4) Press Enter on the Program Counter to move to address 1 (0001).
5) Change the Variable to 0101 (decimal 5).
6) Press the Variable's button again to store it on address 1.

Step 4 – Testing the Memory

Change the Variable to something random — the Data RAM doesn't change unless you press the write button.
Why? Because the Write signal is like the "save" button — without it, the RAM ignores whatever is on its input.

Step 5 – How the Elevator Works

Remember from Mission 1 that the Program Counter counts from 0 to 15. Conveniently, the Data RAM has exactly 16 floors — so the Program Counter can address every single one, floor by floor. When you get to floor 15 (1111) and press Enter, it loops 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.

Step 6 – Clearing Memory

To erase the Data RAM:

1. Set the Variable to 0000.
2. Go to floor 0 and press the Variable's button to store.
3. Press Enter to go to the next floor.
4. Repeat steps 2-4 you're back at floor 0.

Congratulations — you've just run your first loop algorithm: "repeat until done."

Random Data Mystery

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 powered up, some of them are randomly open and some of them are randomly closed. That's not a problem. We simply clear the memory cells, as we have shown.

Checkpoint Challenges 5.1	
?	Store the number 0111 (decimal 7) on floor 5. Move away and come back to check it. Does it remain?
	If you wanted to store a high score in a game, would you use a Latch or the Data RAM? Why?
	If you wanted to store the value 0 on floor 0, value 1 on floor 1 … until the value 15 on flor 15, how would you do this automatically, just using the Ptrogram Counter and the Data Ram Module?

Mission 6 – The B4 Learns to Route Traffic

Objective

Teach your B4 to choose which data stream gets through. You'll use the Selector (a 2-to-1 multiplexer, or MUX) to steer data from one of two sources to a single destination.

The Story

Your B4 now has a heartbeat, can add and subtract, and can remember. But there's a new problem: sometimes two parts want to speak at once. If both signals rush down the same lane, you get a pile-up. Real computers solve this with a traffic controller that opens one lane and closes the other at exactly the right time.

On the B4, that controller is the **Selector**. Flip its control line one way, and it forwards **Input A**; flip it the other, and it forwards **Input B**. This seems small, but it's how computers make decisions about *where* data goes next.

Step 1 – Meet the Selector

Think of the Selector as a guarded gate with two entrances (left and right inputs) and one exit (the output). A single control wire tells the guard which entrance to open:

- Control HIGH → pass Input A (left side)
- Control LOW → pass Input B (right side)

(Your board's silk and LED label show which side is active; follow the icon/LED on your module.)

Tip: Engineers call this device a multiplexer or MUX. You've just met one of the most common building blocks in digital design.

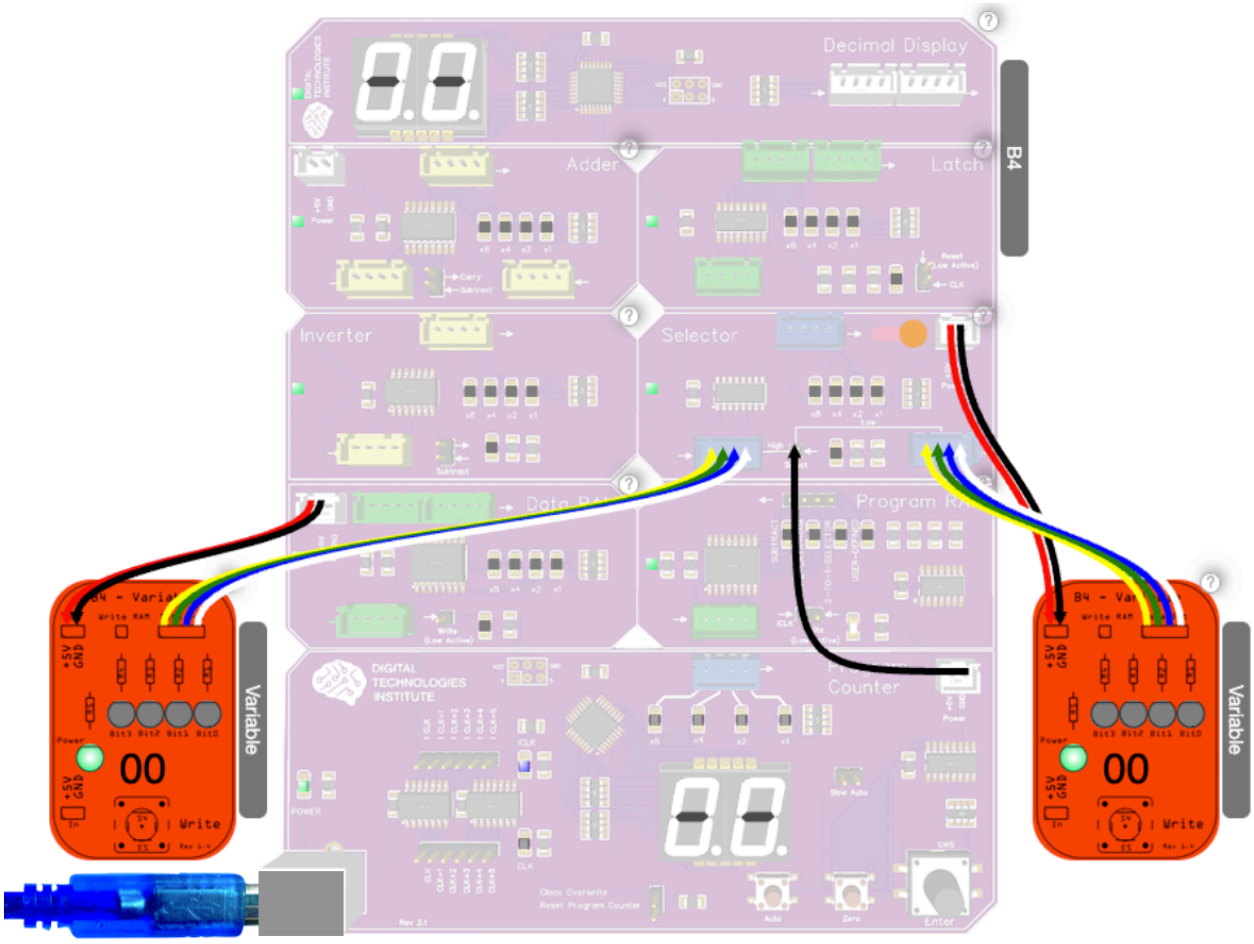
Step 2 – Build the Router

You'll need:

- Selector module
- Two Variable modules (to act as two different data sources)

Wire it up:

1. Plug each Variable output into one of the Selector's two inputs.
2. Connect a 1-pin control wire from the Selector's Select pin to GND first (we'll flip it later).
3. Ensure power is connected to the Variables.



Setup of Mission 6

Step 3 – Try Routing Data

1. Set the left Variable to a pattern you'll recognise (e.g., 1010).
2. Set the right Variable to a different pattern (e.g., 0101).
3. With Select = GND, check the Selector's LEDs: you should see the right Variable's pattern at the Selector's output.
4. Move the Select wire from GND to +5V and watch the Selector's output switch to the left Variable's pattern.
5. Flip the Select line back and forth a few times. The Selector's output should cleanly toggle between A and B every time.

What you've learned: one tiny control bit decides which whole 4-bit value gets through. That's powerful.

Step 4 – Why This Matters

Right now your Selector is choosing between two Variables, but soon it will choose between the Adder's output and Data RAM. That choice is how a computer decides whether to:

- reuse a fresh result from the ALU (Adder), or
- fetch a stored value from memory.

That's the secret of "dataflow": choosing the right lane at the right moment.

Mission 7a – Adding Three Numbers by Hand (with a Memory Boost)

Objective

Use the Latch to store an intermediate sum so you can reuse one of the Variables to enter a third number.

The Story

Your B4 can add two numbers easily — but what if you want to add three? With only two Variables, you can't feed all three numbers in at once.

The trick is to do the first addition, store the result in short-term memory (the Latch), then reuse one of the Variables for the third number. This means you, the operator, will act as the "data transfer system" — reading the value from the Latch and manually setting it on a Variable.

It's not glamorous, but it works — and it's exactly how early computers were tested by engineers before everything was automated.

Analyse

The aim is to add three numbers. 1, 2 and 4

We can express this as a mathematical equation in the form: **1 + 2 + 4 = end result**

1 + 2 + 4 is the same as first calculating 1 + 2 and then adding 4 in a second step. We therefore **decompose** the addition of three numbers into two additions of two numbers each:

1 + 2 = 3 (first sum)
3 + 4 = 7 (end result)

And as we know, the Adder module can handle two inputs — sweet!

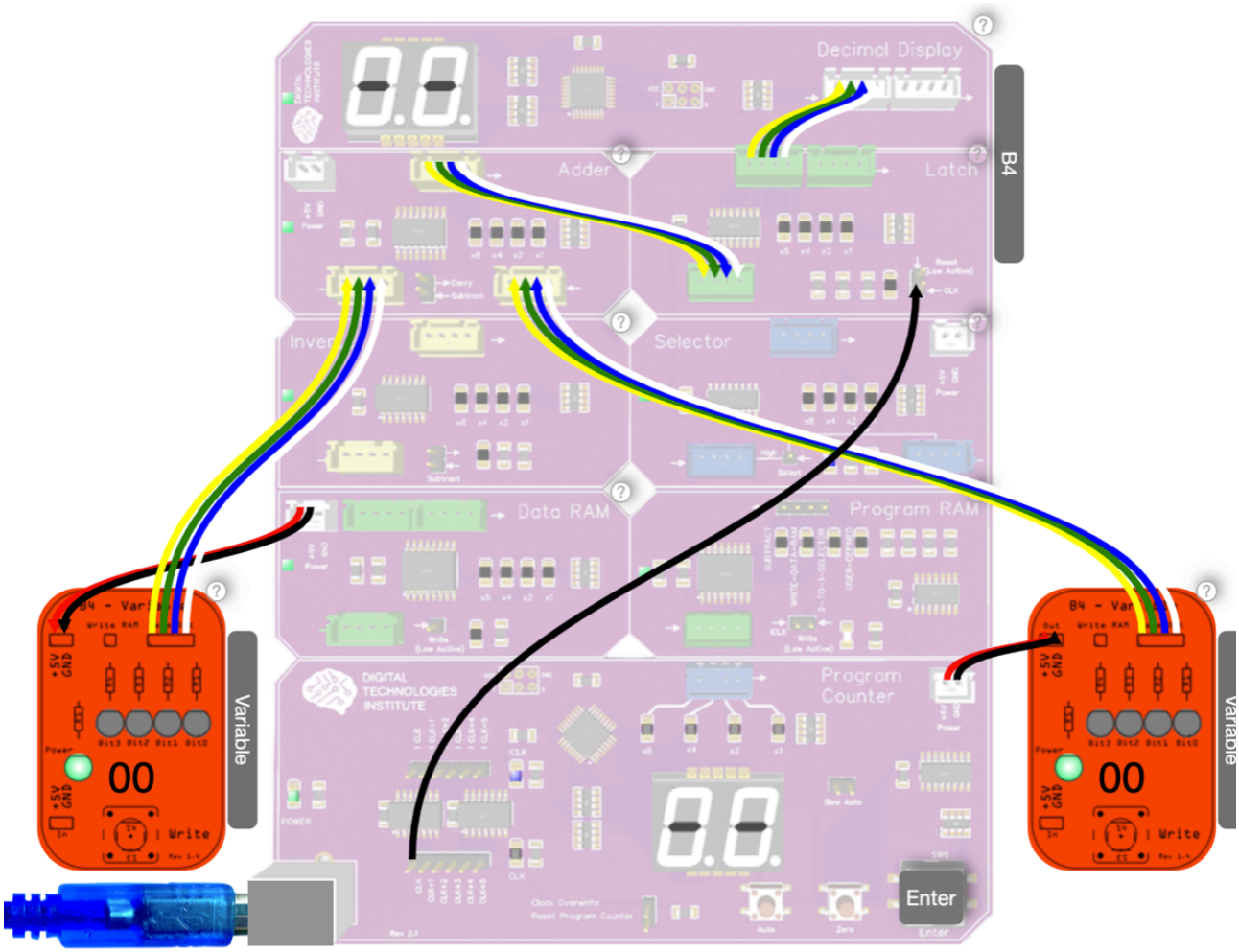
Step 1 – Build the Circuit

You'll need:

- Program Counter (for power and clock)
- Adder
- Latch
- Two Variable modules
- (Optional) Decimal Display

Connections:

1. Left Variable → left Adder input
2. Right Variable → right Adder input
3. Adder output → Latch input
4. Latch CLK In → Program Counter CLK output (so pressing Enter stores the Adder output)



Setup of Mission 7a

Step 2 – First Addition (1 + 2)
1. Set the left Variable to 1 (your first number). 2. Set the right Variable to 2 (your second number). 3. Press Enter on the Program Counter — the Latch stores the sum 3 (1 + 2). 4. Check the LEDs on the Latch — this is your first sum

Step 3 – Second Addition (3 + 4)
1. Decide which Variable you'll reuse (often the right one is easiest). 2. Look at the LEDs on the Latch and set the chosen Variable to exactly the same pattern — you've now "copied" 3 into that Variable. 3. Change the other Variable to 4 (your third number). 4. The Adder now displays 0111 (7) — the sum of all three numbers.

Step 4 – Why This Matters
This method teaches the key principle of storing and reusing intermediate results — something every CPU does constantly. You've done the "data transfer" step manually here, but in the next mission, we'll make the B4 do it automatically using the Selector.

Checkpoint Challenges 7a	
	Try 2+6+7. What's the final result?
	What happens if you forget to store the first sum in the Latch before changing a Variable?
	Could you extend this method to add four numbers? How?

Mission 7b – Automating the Third Number (No More Manual Copying)

Objective

In 7a, you copied the intermediate result from the Latch into a Variable. Since computers are all about automation, we now want to instead use the Selector to bring the intermediate result from the Latch to the Adder.

The Story

In Mission 7a, you acted as the “data courier” — reading the value from the Latch and copying it by hand into a Variable. That worked, but it was slow, clumsy, and error-prone.

Real computers don’t rely on humans to shuffle data around. They use circuits that make those decisions automatically, switching data paths at the right moment. On the B4, the Selector is that circuit.

By combining the Selector with the Latch, your B4 can now add three numbers smoothly: first add 1 and 2, store the result (3) in the Latch, then route 3 automatically back into the Adder to add 4, which comes from the left Variable.

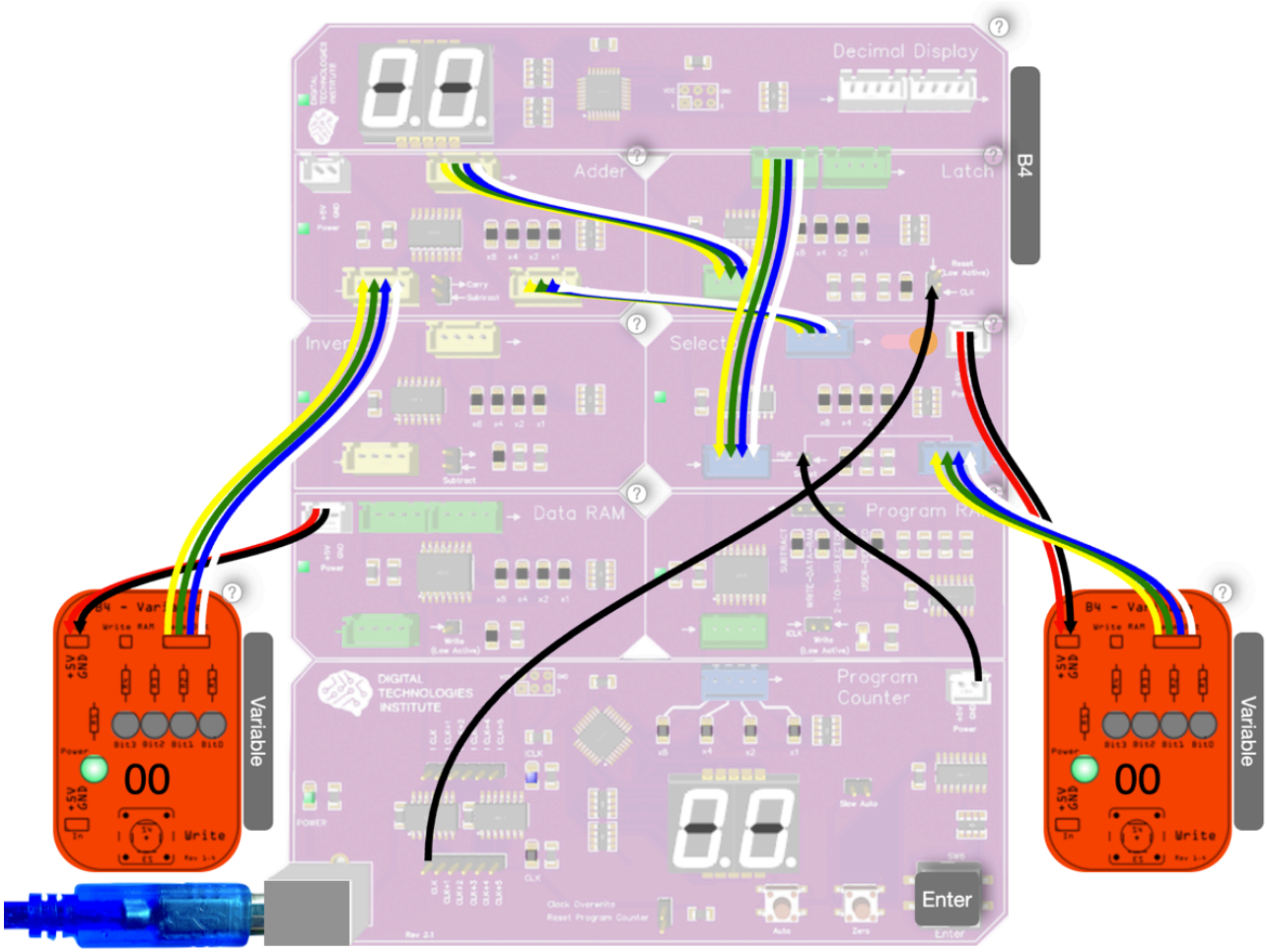
Step 1 – Build the Circuit

You’ll need:

- Program Counter (for power and clock)
- Adder
- Latch
- Selector
- Two Variable modules
- (Optional) Decimal Display

Connections:

1. Left Variable → left Adder input
2. Selector output → right Adder input
3. Right Variable → right Selector Input
4. Latch output → left Selector input
5. Adder output → Latch input
6. Latch CLK In → Program Counter CLK (so Enter stores the Adder output)
7. Selector Select control:
    - Select = HIGH → The output of the Latch is connected to the Adder.
    - Select = LOW → Adder takes the right Variable.



Setup of Mission 7b

Again, we try to compute 1+2+4:

Step 2 – First Addition (1 + 2)
1. Set the left Variable to 1 (your first number). 2. Set the right Variable to 2 (your second number). 3. The value of the left Variable goes directly to the Adder (left input port). The value of the right Variable travels via the Selector to the right input port input of the Adder 4. Press Enter on the Program Counter — the Latch stores the sum 3 (1 + 2). 5. Check the LEDs on the Latch — this is your first sum.

Step 3 – Second Addition (3 + 4)
1. Move the Select wire from GND to +5V and watch the Selector's output switch to the the same number that the Latch holds. (3). The Selector is now providing 3 to the right port of the Adder. 2. Change the left Variable to 0100 (4) (your third number). 3. The Adder now displays 0111 (7) — the sum of all three numbers.

Step 4 – Why This Matters

This is your first taste of automation. You've stopped doing the busywork yourself and let the B4 manage its own dataflow. Modern CPUs are full of circuits like the Selector, automatically choosing which path data should take at each clock cycle.

With this, your B4 can now chain calculations without you constantly re-entering values — a huge step towards a fully programmable computer. But we still needed to move that Selector wire and enter the data into the variables.

As we will see in the next mission, 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 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 mission, we have learned that we can add more than two numbers by using the Selector and the Latch. The Selector helps us to switch the output of the Latch into the next addition cycle, whilst the Latch remembers the intermediate result. 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 mission, 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 problem is, surprisingly, more memory.

Checkpoint Challenges 7b	Try adding four numbers. 1+2+4+5.
?	What's the final result?
	How often do you need to move the endpoint of the select wire?

Mission 7c – Loading and Adding with Data RAM

Objective
Understand why numbers from RAM must be handled one at a time. Learn how to first **LOAD** a number from RAM into the Latch (to park it safely), and then **ADD** another number from RAM to it. This models the fundamental cycle used in real computers: load a value into a register, then perform arithmetic on it.

The Story
In Mission 7b, you added numbers by chaining results through the Selector. That was easy while one of the numbers sat in a Variable. But real computers don't work that way. Instead, **all numbers live in RAM** — a big row of memory boxes, each with an address. The CPU can only open one box at a time. So here's the problem: • You can grab the first number from RAM, but as soon as you go to get the second one, the first number is gone. • You need somewhere to **park the first number** so it's ready when the second one arrives. That's where the Latch comes in. Real CPUs do this in two clear steps: 1. **LOAD** → Take a number from RAM and park it in the Latch (like storing it in short-term memory). 2. **ADD** → Go back to RAM, fetch the next number, and add it to the Latch. To make this work, we change the wiring slightly: • The **Selector now feeds the Latch**. • The **Latch then feeds the Adder**. This new cycle forces us to **LOAD first, then ADD** — the same rhythm a CPU follows millions of times per second. And it's the same way you do mental arithmetic: hold the first number in your head, then add the next one to it.

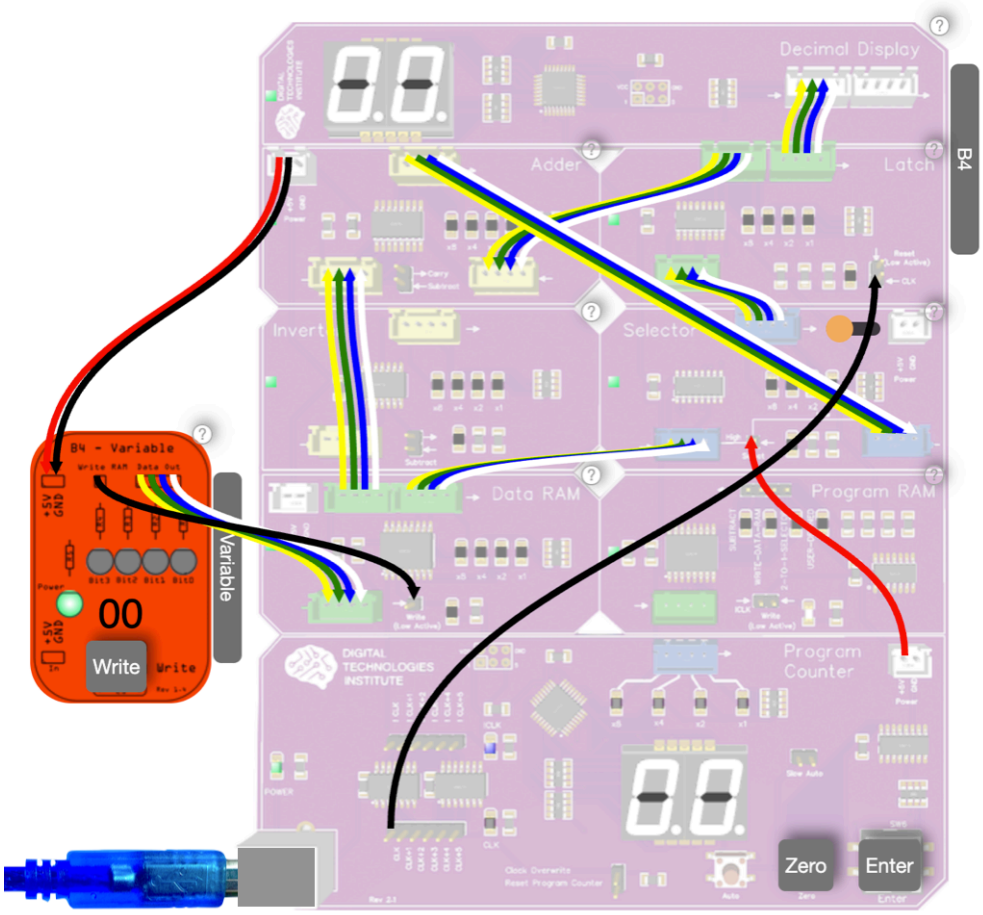
Step 1 – Build the Circuit

You'll need:

- Program Counter (for power , clock, and program step counting)
- Data RAM
- Adder
- Latch
- Selector
- One Variable module (for programming the Data RAM)
- (Optional) Decimal Display

Connections:

1. Left Variable → Data RAM input
2. Data RAM → left Adder input
3. Data RAM → left Selector input
4. Selector output → Latch input
5. Latch output → right Adder input
6. Adder output → right Selector input
7. Variable Write RAM pin → Data RAM Write pin
8. Latch CLK In → Program Counter CLK (so Enter stores the Selector's output)
9. Selector Select control:
   - Select = HIGH → The output of the Data RAM will be latched.(we start with it)
   - Select = LOW → The output of the Adder will be latched.



Setup of Mission 7b

Again, we try to compute 1+2+4:

Step 2 – Programming the Data RAM
First, we program the values 1, 2, and 4 into our Data RAM. We've already learned how to do this in MIssion 5. Follow the steps below: 1. Set the Program Counter to 0000. That's step 0 of our program. 2. We enter our first data. Set the Variable to 0001 and then press the button on the Variable. 3. For our next data, we progress the Program Counter to 0001, which is step 1. 4. Then, on the Variable, enter 0010 and then press the button on the Variable. 5. Progress the Program Counter to 0010 and enter 0100 into the Variable. Click its button to store this data into the Data RAM. 6. 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 until the Program Counter shows 15. 7. Press the Zero button on the Program Counter, so that the Program Counter shows 00

Step 3 – Addition (1 + 2 + 4)
1. Press Enter on the Program Counter. This loads the number 1 into the Latch. 2. Move the Select wire from +5V to GND. This makes the Slector pay attention to the output from the Adder. 3. Press Enter on the Program Counter — the Latch stores the output 3 (1 + 2). 4. Press Enter on the Program Counter — the Latch stores the output 7 (3 + 4).

Observe how the values 1, 2, and 4 come out of the Data RAM. Other than moving the Select wire just once (after the value 1 has been latched). We just keep pressing the Enter button on the Program Counter, and the Adder keep adding. **Wow!**

Checkpoint Challenges 7c	
?	You just computed 1 + 2 + 4 = 7. Now add another number from RAM (e.g., 5). What do you expect the new total to be? Try it and see if your prediction was correct.
	Why do we only need to flip the Select wire once, after the first number is loaded into the Latch?
	In a real CPU, why is it useful to separate the LOAD step (into a register) from the ADD step?

A Thought Experiment…

Imagine if computer designers wanted to save a chip and just remove the Selector. Couldn't we just let the Latch and RAM feed the Adder directly? That would make the wiring simpler, right?

It does work — at least at first. But as soon as we try to store results back into RAM, things start to go wrong in surprising ways (hint: numbers can suddenly double!).

To see why this "Accumulator-only" approach looks tempting but ultimately fails, check out **Appendix D – Design Debate**. There, we compare the two designs head-to-head and show why real CPUs stick with the Selector model.

🟡 Part 2 – Automation with Program RAM

In Part 2, the B4 takes its first steps away from manual wiring and towards automation. Instead of you moving wires, the Program RAM begins to take charge. You'll learn how to write your first program and extend the system to perform both addition and subtraction under program control.

By the end of part 2, you'll add the ability to write results back into Data RAM so that variables can be reassigned and reused later. This is a key milestone: your B4 is no longer just a calculator, but a program-driven machine that can update its own memory while running.

Mission 8 – Automating the Selector with Program RAM

Objective

Learn how to use Program RAM to control the Selector automatically. This is the first step toward letting the computer run instructions on its own, without you flipping switches by hand.

The Story

In Mission 7c, you had to move the Selector yourself to choose between loading a value from RAM or adding to the Latch. That worked, but it still needed you in the loop.

Now it's time to let the computer take over. Real CPUs store a list of instructions in **Program RAM**, and the Program Counter steps through them one by one. Each instruction tells the computer what to do next.

Here's the plan:

- Program RAM holds tiny "control words" (like "Selector = HIGH" or "Selector = LOW").
- The Program Counter moves through Program RAM automatically.
- Instead of you flipping the Selector, Program RAM tells it what to do at the right time.

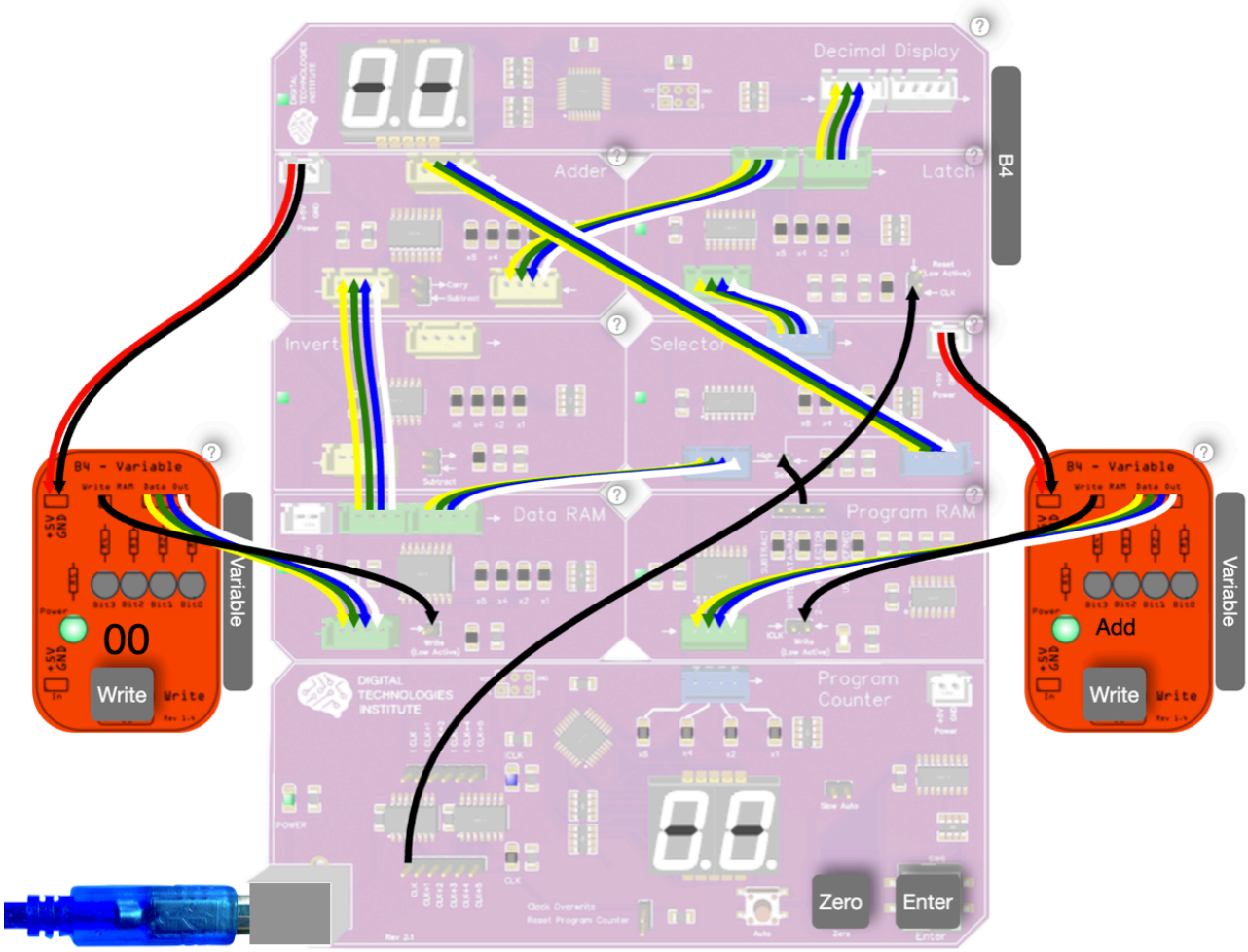
This is the beginning of true automation: the computer is no longer just a collection of parts you control directly — it's starting to follow a stored program.

Step 1 – Build the Circuit

Start with the exact same setup you built in Mission 7c. The only two changes are this:

- Take the end of the **Select wire** away from + 5V/GND and move it to the **second pin from the right** of the Program RAM output. It is labelled as 2-To-1 Selector.
- Place the second variable to the right side of your B4. We use it to program the Program RAM later
- Connect the right Variable to power and to the Program RAM input
- Right Variable Write RAM pin → Program RAM Write pin

That's it! Now Program RAM, not you, decides when the Selector switches between LOAD and ADD.



Setup of Mission 8

Step 2 – Programming the Data RAM
First, we program the numbers that we want to add into the Data RAM. These are the values 1, 2, and 4. We do this exactly like in mission 7c. Flip back a couple of pages and look it up if you have forgotten. Use the left variable to program the Data RAM

Step 3 – Programming the Program RAM
Now, using the right variable, we program a single instruction into the Program RAM. 1. Set the Program Counter to 0000. That's step 0 of our program. 2. We enter our first instruction (LOAD). Set the Variable to 0010 and then press the button on the Variable. 0010 means that the Program RAM's second output port from the right will be HIGH. And this is the port where you have attached the Select wire from the Selector. (Code and hardware always go hand in glove) 3. We clear the remaining Program RAM. Set the Program Counter to 0001, set the right Variable to 0000 and press the button on the right Variable. Repeat this until the Program Counter shows 15. 4. Press the Zero button on the Program Counter, so that the PC shows 00

Step 4 – Addition (1 + 2 + 4)
1. Press Enter on the Program Counter. This loads the number 1 into the Latch. 2. Press Enter on the Program Counter — the Latch stores the output 3 (1 + 2). 3. Press Enter on the Program Counter — the Latch stores the output 7 (3 + 4).

Observe how the values 1, 2, and 4 come out of the Data RAM. Our single LOAD instruction at the start of the program pushes the first number (1) from the Data RAM via the selector into the Latch. And there it waits until ~~it can take over the world~~ the next number (2) becomes available. We just keep pressing the Enter button on the Program Counter, and the Adder keep adding. **Congratulations! We have made a computer !!!!!!**

Step 5 – Why This Matters
Up until now, you were still part of the circuit — flipping the Selector at just the right time. That worked, but a real computer can't rely on a human to pull the levers. By letting Program RAM control the Selector automatically, the computer is starting to follow instructions on its own. This is the heart of the stored-program idea: instead of wiring the computer differently for every task, we just change the program in memory. This is what makes modern computers so powerful — they can switch from adding numbers, to running a game, to browsing the web, all by following different instructions stored in memory.

Checkpoint Challenges 8	
?	Right now, our Program RAM has just one instruction: LOAD the first number into the Latch. After that, the Selector stays on ADD, so every new number from Data RAM is added correctly. But here's the puzzle: 👉 What would happen if you accidentally programmed two LOAD instructions in a row at the start of Program RAM? • Would the additions still work? • Why or why not?
	So far, our computer always reuses the result from the Latch for the next addition. That's why it can keep chaining numbers like 1 + 2 + 4. But what if you wanted to do two separate additions instead? For example: • Add 1 + 2 (and get 3) • Then separately add 4 + 5 (and get 9) 👉 Can you think of a way to program this so the second addition does not reuse the first result?

Mission 9 – Program Tables

Objective
Learn how to write programs in a clear table format, making it easier to describe, run, and share your programs.

The Story
Up to now, we've been giving you programming instructions step by step: "Load this number… now add that one…". That works fine for short tasks, but as soon as programs get longer, it's easy to lose track. This is exactly why real programmers invented notation — a structured way of writing instructions so that other humans can follow them without confusion. From now on, we'll write our programs as tables. Each row shows: • the step number, • the value it uses from Data RAM, • the instruction (LOAD or ADD). Think of the table as your program's "recipe card." Anyone who looks at it will know which value needs to go into which RAM module, at which program step.

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 to express the program 1+ 2

	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								
Step 1	0	0	1	0	0	0	0	0
Step 0	0	0	0	1	0	0	1	0

You already know from Mission 8 that the Program RAM's 2nd output pin from the right controls the Selector. Let's update our table and name this **SEL** (for select). While we're at it, we add a comments column — adding comments is a good practice that software engineers follow! Since there are so many empty rows, we collapse them to save space.

Here is the revised table for a program that computes 1+ 2.

	Data RAM				Program RAM				Comment
Step #	**3**	**2**	**1**	**0**	**A**	**B**	**SEL**	**D**	
Steps 2-15	0	0	0	0	0	0	0	0	do nothing
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

Just so that we don't forget, let's start a little table to help us remember how select works.

	Bit 1 (SEL)
Name	Select
When it's 1…	Selector takes input from Data RAM
When it's 0…	Selector takes input from Adder

Checkpoint Challenges 9	
?	So far, we've shown you how to write the program for 1 + 2 in table form. Now it's your turn! Create a Program Table for the addition 3 + 4. • Fill in the Data RAM values. • Decide what the Program RAM's SEL bit should be for each step. • Write a short comment for each row explaining what happens.
	Bonus: Can you extend your table so that the computer first calculates 1 + 2, and then separately calculates 3 + 4 (two additions, not chained together)?

What's Next

Your computer just learned how to follow a recipe card — the program table — and it can already add numbers all on its own. But every great computer has more tricks up its sleeve. In the next missions, we'll teach it two powerful new skills: first, how to **work the Inverter** so it can subtract as well as add, and then how to **write results back into Data RAM** so it can remember answers for later. Step by step, you're unlocking the CPU's secret language.

Mission 10 – Subtraction with the Inverter

Objective

Extend your program table so the Program RAM can control the Inverter. This allows your computer to perform subtraction automatically.

The Story

Back in Mission 3, you used the Inverter to flip every bit of a number. On its own, that didn't feel very useful. But here's the twist: when you combine inversion with a +1 signal into the Adder, subtraction becomes possible.

This is how real computers do it. They don't have a separate "subtract" circuit — instead, they turn subtraction into addition by cleverly inverting one number and adding 1. It's called two's complement arithmetic.

To make this work, we extend the wiring:

- Place the Inverter between the Data RAM and the Adder.
- Connect a new control wire from the Program RAM to the Inverter.
- Use the Inverter's +1 output to feed into the Adder's carry-in pin.

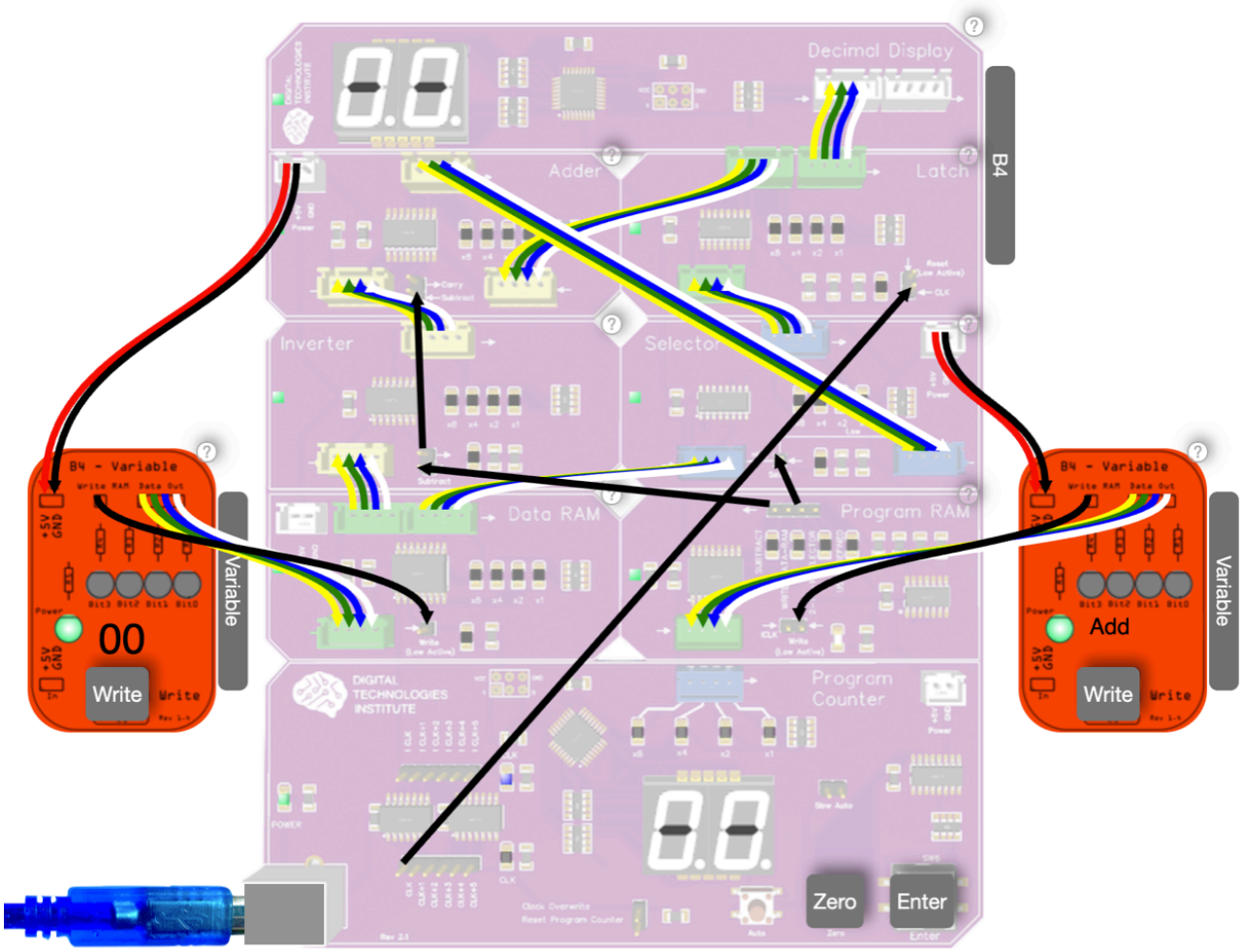
Now, when the Program RAM sets the SUB bit high, the Inverter flips the number and sends the +1 into the Adder. If SUB is low, nothing changes and the Adder just adds normally.

Step 1 – Build the Circuit

Start with your Mission 8 build

- Insert the Inverter between Data RAM and Adder.
- Run a new 1-pin wire from the Program RAM's leftmost output pin (SUBTRACT) to the Inverter's control input.
- WIth another 1-pin wire, connect the Inverter's SUBTRACT pin to the Adder's SUBTRACT pin.

That's it! Now Program RAM decides when the Inverter switches between addition and subtraction.



Setup of Mission 8

Step 2 - Designing our program.
Let's say we wanted to compute 8 + 4 - 2. How could our program look like?

	Data RAM				Program RAM				Comment
Step #	**3**	**2**	**1**	**0**	**SUB**	**B**	**SEL**	**D**	
Steps 3-15	0	0	0	0	0	0	0	0	do nothing
Step 2	0	0	1	0	1	0	0	0	Activates the Inverter. Sends the binary complement 1101 to the Adder. Result = 10
Step 1	0	1	0	0	0	0	0	0	Send 4 to the Adder, which adds it to the 8 stored in the Latch. Result = 12
Step 0	1	0	0	0	0	0	1	0	Load 8 from the Data RAM into the Latch. This is the first number for the adder

Step 3 – Programming the Data and Program RAM

Just like in Mission 8, you need to load both your Data RAM and Program RAM before running.

1. Set the Program Counter to 0.
2. Use the left Variable to set a value for the Data RAM, and the right Variable for the Program RAM.
3. For each value:
   - Push the Variable's button to transfer its value into the RAM module.
   - Then press Enter on the Program Counter to move to the next step.
4. Repeat this process — set values, push the buttons, press Enter — until your whole program and data are entered.

Step 4 – Running 8 + 4 - 2

1. Press Enter on the Program Counter. This loads the number 8 into the Latch.
2. Press Enter on the Program Counter — the Latch stores the output 12 (8 + 4).
3. Press Enter on the Program Counter — the Latch stores the output 10 (12-2).

Observe how the values 8, 4, and 2 are read out of the Data RAM.

- The single **LOAD** instruction at the start pushes the first number (8) through the Selector into the Latch, where it waits.
- Next, the second number (4) arrives, and the Adder combines it with the latched value.
- **Now comes the interesting part**: in program step 2, the Program RAM activates the **Inverter**. This flips the third number from 0010 (2) into 1101 — the two's complement representation of –2.

Step 5 – Why This Matters

Back in Mission 8, you saw how the Program RAM could control the **Selector**, saving you from manually moving wires each time. In this mission, the same idea is extended: the Program RAM now controls the **Inverter** as well.

This means subtraction no longer requires you to unplug wires or reroute signals by hand — the Program RAM does it automatically at the right moment. Imagine if you had to flip the Inverter wire by hand every time — your program would grind to a halt.

Each new control wire you connect gives the Program RAM more "power" to manage the circuit, turning your setup into something closer to a real CPU.

Let's just update our Program RAM cheat sheet table to help us remember.

- Bit 3 (SUB): Decides whether the Adder adds (0) or subtracts (1).

That's all you need for now — no need to memorise, just check the table when you're writing your programs. In the next mission, we will discover what Bit 2 does.

	Bit 3 (SUB)	Bit 2 (?)	Bit 1 (SEL)	Bit 0 (?)
Name	Subtract	?	Select	?
When it's 1…	The Inverter is active → Adder subtracts	?	Selector takes input from Data RAM	?
When it's 0…	Adder adds	?	Selector takes input from Adder	?

Checkpoint Challenges 10	
?	Your current program shows how subtraction works with the Inverter. Now, change it to calculate 11 – 3 – 2.
	Imagine you didn't have Program RAM. Which wire would you have to manually move in order to make the subtraction work? Why is letting Program RAM handle this better?

Mission 11 – Storing Calculation Results in the Data RAM

Objective

Learn how to let Program RAM control writing into Data RAM. This allows the computer to store results permanently in memory, so variables can be assigned and re-assigned with new values from calculations — just as you would expect when programming.

The Story

Up to now, your computer could only hold results temporarily in the **Latch**. That was fine for the very last answer, but as soon as the next operation ran, the old result was gone. The **Program Counter** only moves forward, so once you've lost a value, you can't get it back.

But when we write programs, we expect something more powerful:

- **Variables can change**. A value can be updated by a calculation and then reused later.
- **Results can be kept**. Once you compute an answer, it doesn't vanish — it's stored for the next steps.

Think of a game score: every time you earn points, the computer adds to the score and then **stores the new value back into memory**. Or imagine a bank balance: when you spend or deposit money, the balance gets recalculated and re-assigned with the updated amount.

This is where writing back to Data RAM becomes important. By saving results into memory, your computer can **keep them** instead of losing them. That's a big difference between a simple calculator and a real computer:

- A calculator only ever shows the latest answer.
- A computer can **store, update, and reuse answers** in memory as the program runs.

In this mission, you'll take the next step and give your B4 the ability to write results back into Data RAM automatically during runtime. This is the final ingredient that starts to make your B4 behave like a true CPU.

Step 1 – Build the Circuit

Use your Mission 10 build.

Step 2 - Designing our program.

We wanted to compute 8 + 4 - 2 and store the result (10) into the Data RAM. We re-use the program from Mission 10 and add Step 3.
0100 is the code to write to the Data RAM. We call it WRT.

	Data RAM				Program RAM				Comment
Step #	**3**	**2**	**1**	**0**	**SUB**	**WRT**	**SEL**	**D**	
Steps 4-15	0	0	0	0	0	0	0	0	do nothing
Step 3	0	0	0	0	0	1	0	0	Stores the contents of the Latch in the Data RAM
Step 2	0	0	1	0	1	0	0	0	Activates the Inverter. Sends the binary complement 1101 to the Adder. Result = 10.
Step 1	0	1	0	0	0	0	0	0	Send 4 to the Adder, which adds it to the 8 stored in the Latch. Result = 12
Step 0	1	0	0	0	0	0	1	0	Load 8 from the Data RAM into the Latch. This is the first number for the adder

Step 3 – Programming the Data and Program RAM

Just like in Mission 10, you need to load both your Data RAM and Program RAM before running. If you have forgotten, skip back to Mission 10 and look up the programming instructions

!!! Step 4 – 🔧 Re-wiring for Runtime !!!

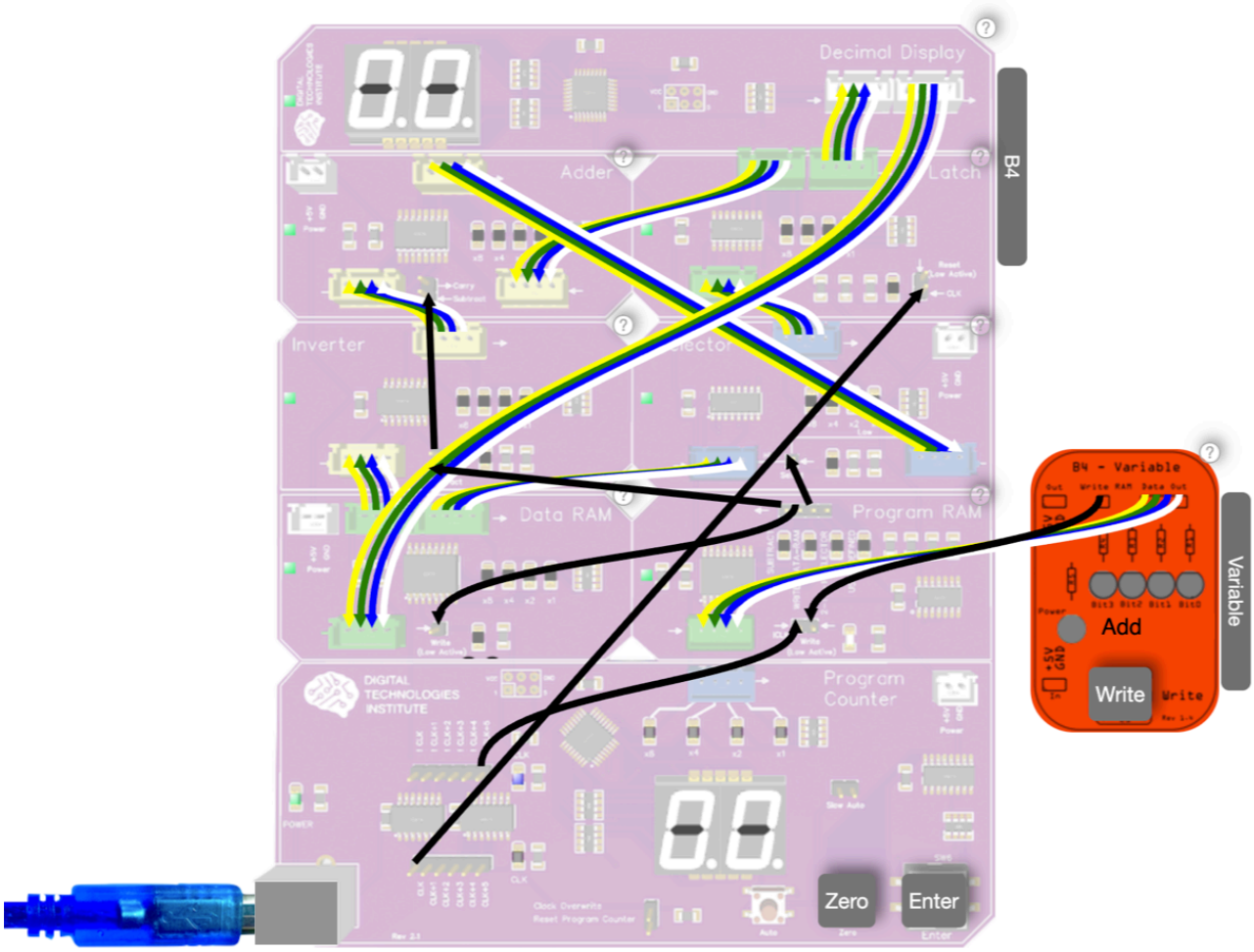
Once you've finished programming the Data RAM and Program RAM (just like in earlier missions), it's time to prepare the computer for runtime. We hand over the data and control paths from the left Variable to the Latch and the Program RAM. **The Data RAM will take its data input from the Latch, and the WRT command will come from the Program RAM.**

Follow these steps carefully:

1. Reset the Program Counter to 0.
2. Remove the left Variable from the Data RAM module. Unplug the 4-pin and 1-pin wires that connect the Variable and the Data RAM.
3. Connect the Latch to the Data RAM.
   - Use a 4-pin wire to connect the Latch's output to the Data RAM's input.
4. Connect the Program RAM to the Data RAM's write pin.
   - Use a 1-pin wire from the 2nd left Program RAM output (labelled WRITE_DATA_RAM) to the Write-pin on the Data RAM.
   - This lets the Program RAM decide when to store a new value in memory.
5. Connect a 1-pin wire from the Program Counter's !CLK+5 pin to the !CLK pin of the Program RAM
   - This enables the Program RAM to communicate with the Data RAM.

Now your computer is ready to execute the program with the new wiring in place.

Yes, unplugging and re-plugging wires feels clunky — but don't worry, in the next mission we'll add the Automatic Programmer to get rid of this step.



Setup of Mission 11 (Runtime)

Step 5 – Running 8 + 4 - 2 + WRT
1. Press Enter on the Program Counter. This loads the number 8 into the Latch. 2. Press Enter on the Program Counter — the Latch stores the output 12 (8 + 4). 3. Press Enter on the Program Counter — the Latch stores the output 10 (12 - 2). 4. Press Enter on the Program Counter — the Data RAM stores the contents of the Latch, 10. Observe how the Data RAM holds 1010 at the end of the program.

Step 6 – Why This Matters
With Program RAM now in charge of the write line, your B4 can save values from the Latch back into Data RAM — automatically, without you moving wires. This step turns the B4 from a “one-shot calculator” into a true computer that can remember its own results.

Let’s just update our Program RAM cheat sheet table to help us remember.

- Bit 2 (WRT): Decides whether to write the current Latch value into Data RAM (1) or not (0).

	Bit 3 (SUB)	Bit 2 (WRT)	Bit 1 (SEL)	Bit 0 (free)
Name	Subtract	Write	Select	(free)
When it's 1…	The Inverter is active → Adder subtracts	Store the Latch value in Data RAM	Selector takes input from Data RAM	Reserved for student extensions
When it's 0…	Adder adds	No write into Data RAM	Selector takes input from Adder	Not used (for now)

Checkpoint Challenges 11	
?	Program the Data RAM so the computer calculates 6 + 3 and stores the result in Data RAM. • Which step should WRT be set to 1? • What value ends up in the RAM at that step's address?
	Run your program with WRT = 0 for every step. • What changes in Data RAM? • What does this tell you about the role of WRT?

📖 Here's a question: How does Program RAM tell the hardware what to do?

The answer: **each bit in Program RAM has a special meaning**. These bits control different parts of the computer through the 1-pin wires we attach to them. When we say that a bit is **1**, or **HIGH**, it means there is a voltage pushing electrons through a wire. We call this a current. This means that tiny amounts of electricity flow through the computer to switch different components on and off as needed.

- **Bit 3 → SUB**
  If this bit is 1, the Inverter is switched on. That makes the Adder perform subtraction instead of addition.
- **Bit 2 → WRT**
  If this bit is 1, data is written back into Data RAM.
- **Bit 1 → SEL**
  If this bit is 1, the Selector points to Data RAM. If it's 0, it points to the Latch/Adder. (You've already used this signal in Mission 7!)
- **Bit 0 → Extension**
  This one is left for you! You can later extend the B4 with your own instruction. For now, we won't use it.

Together, these bits form what we call **instruction codes** (or **operation codes**, shortened to **opcodes**). They tell the computer what action to perform at each step. Every computer processor has different opcodes. The next table summarises them for our B4 CPU.

B4 Opcodes at a Glance

	Bit 3 (SUB)	Bit 2 (WRT)	Bit 1 (SEL)	Bit 0 (free)
Name	Subtract	Write	Select	(free)
When it's 1…	The Inverter is active → Adder subtracts	Store the Latch value in Data RAM	Selector takes input from Data RAM	Reserved for student extensions
When it's 0…	Adder adds	No write into Data RAM	Selector takes input from Adder	Not used (for now)

❓ But what about the Latch and Adder?

- We don't need a special opcode for the **Latch**. It always stores its input whenever the **Clock (CLK)** ticks, which happens automatically when the Program Counter advances.

- We don't need an opcode for the **Adder** either. The Adder is always "on" — it continuously adds whatever data it receives. The only decision we need to make is: do we use its output or not? That's exactly what the **SEL** signal controls.

📖 Here's another question: Why does everything work in just the right order?

When our program stores a result back into Data RAM, how do we make sure the calculation is finished first? If the computer tried to write back too early, it could end up saving the wrong value — or even nonsense!

This is where timing becomes critical. Different parts of the computer need to act in a specific order:

- The **Data RAM** must put its value on the wire before the Latch can store it.
- The **Latch** must capture the result before the Data RAM is told to write it back.

So how does the B4 keep everything in sync?

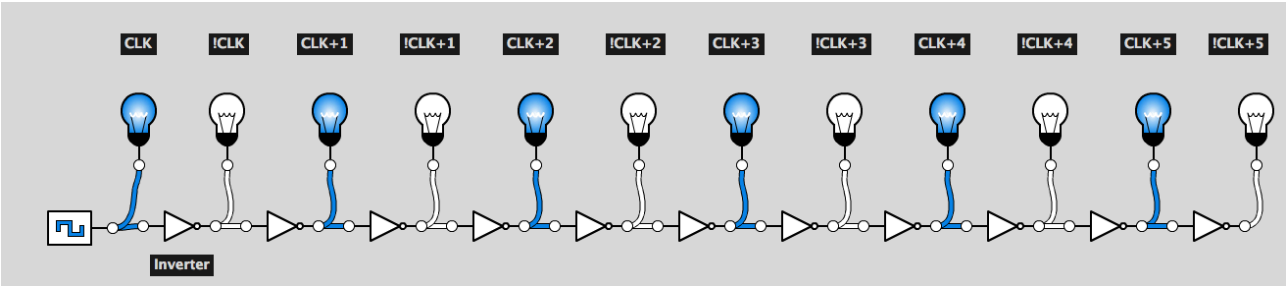
⏱ CLK and !CLK Signals

The Program Counter doesn't just create one clock "tick." It also creates slightly delayed and inverted versions of that tick. These are labelled CLK+1 … CLK+5 and !CLK+1 … ! CLK+5.

Here's how it works:

- When you press Enter, the Program Counter sends a CLK signal, just like in Mission1.
- A simple circuit called an inverter flips this into a !CLK signal.
- Because inverters take a tiny moment (about 5 nanoseconds) to react, the !CLK version is slightly delayed. By chaining more inverters, we can get !CLK+5, which happens a little later still.

This gives us fine control over when each module "listens."



CLK and !CLK Delay Chain

To delay and invert a signal, we require a circuit. It inverts a 1 signal into a 0 signal and a 0 signal into 1.

🔧 Putting It Together

We found the B4 runs smoothly when:

- The Latch is triggered by CLK (so it captures data right away).
- The Program RAM is triggered by !CLK+5 (a later moment), so writing back to memory only happens after the Latch has its result.

In other words: the delayed signals make sure the computer always writes back after the calculation is ready, not before.

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

🔴 Part 3 – Toward a Real CPU

In the final part, the B4 grows into a fully functioning miniature computer. You'll introduce the Automatic Programmer, which removes the last manual wiring steps, and then explore how timing and synchronisation keep everything in the right order. You'll also look at how higher-level instructions are designed and even try out cyber security experiments that reveal what happens when systems are pushed in unintended ways.

By the end of Part 3, your B4 won't just look like a toy computer — it will behave like a true CPU, giving you a hands-on understanding of how every computer you use really works.

Mission 12: Automatic Programming

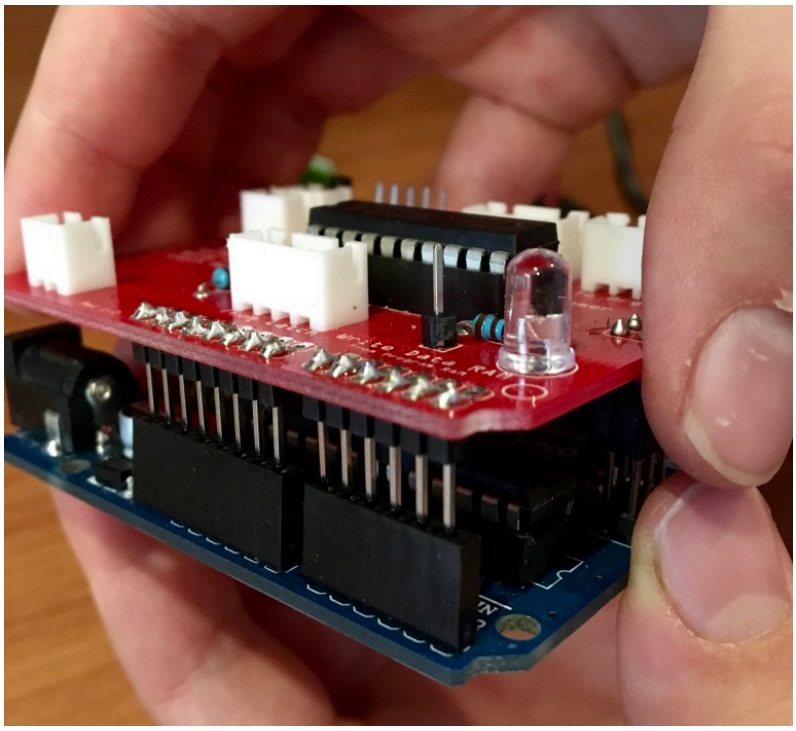
You will probably agree that entering data and program code into the B4 isn't very convenient. In the previous experiments, we have used the Variable modules to get an understanding of coding on the lowest possible level. To make the programming process more elegant, we will introduce the Automatic Programming (AP) shield. The AP can take full control of the B4 during the programming phase, so 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 Master Programmer kit.
2) The B4 Automatic Programmer Shield
3) The setup from experiment 11.
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:



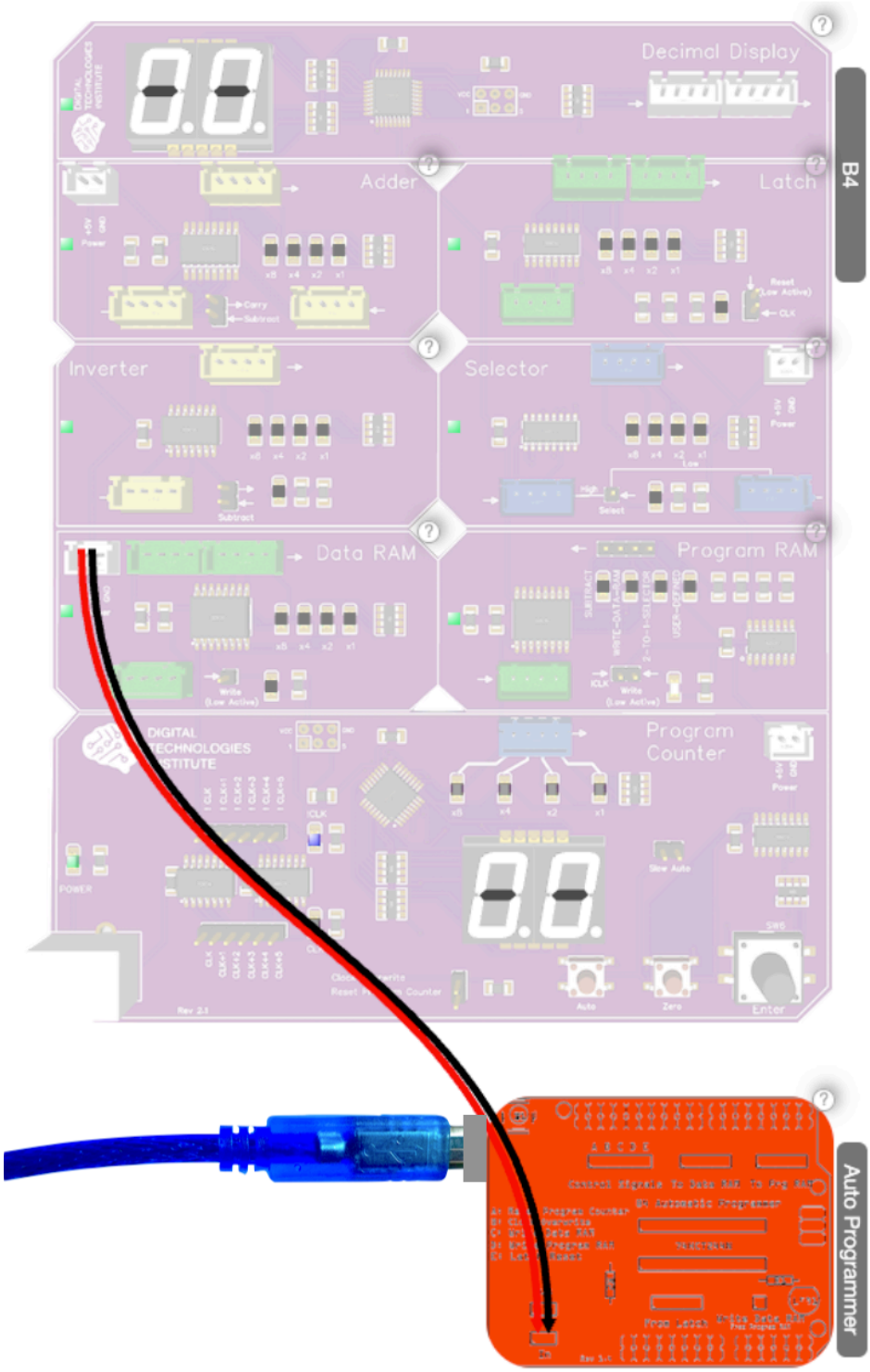
Installing the Automatic Programmer Shield on an Arduino

Step 2: Modules and their Wiring

Place the Automatic Programmer at the bottom of the B4. Because of the complexity of the Setup of this mission, we do it in multiple stages:

Stage 1: Modules and Power Wires:

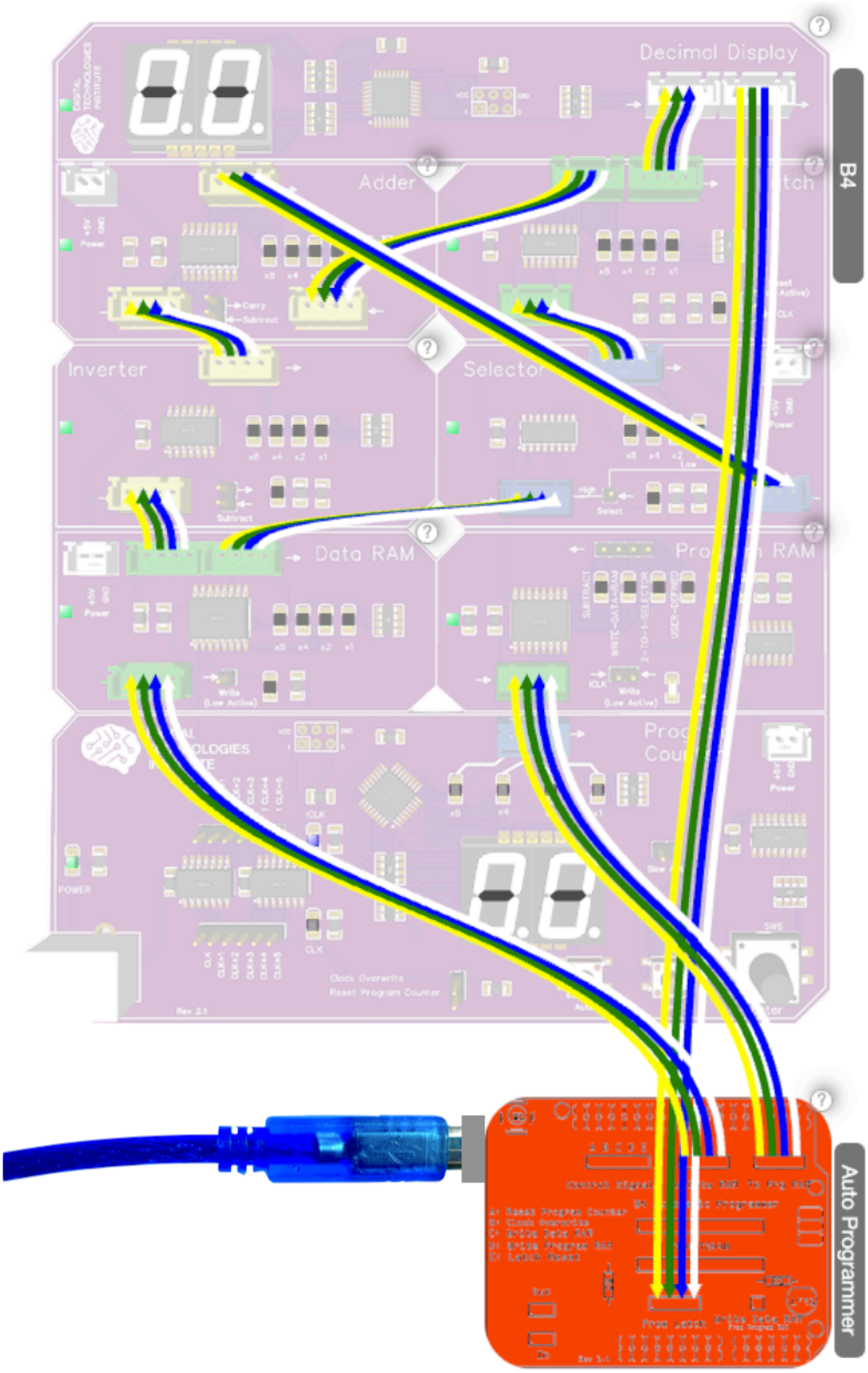
First, let's arrange the modules as shown below. Then, connect the power wire as shown. You will need to run a wire to the Automatic Programmer to supply it with electricity.



Setup of Mission 12:Power Wiring only

Stage 2: Data Wires

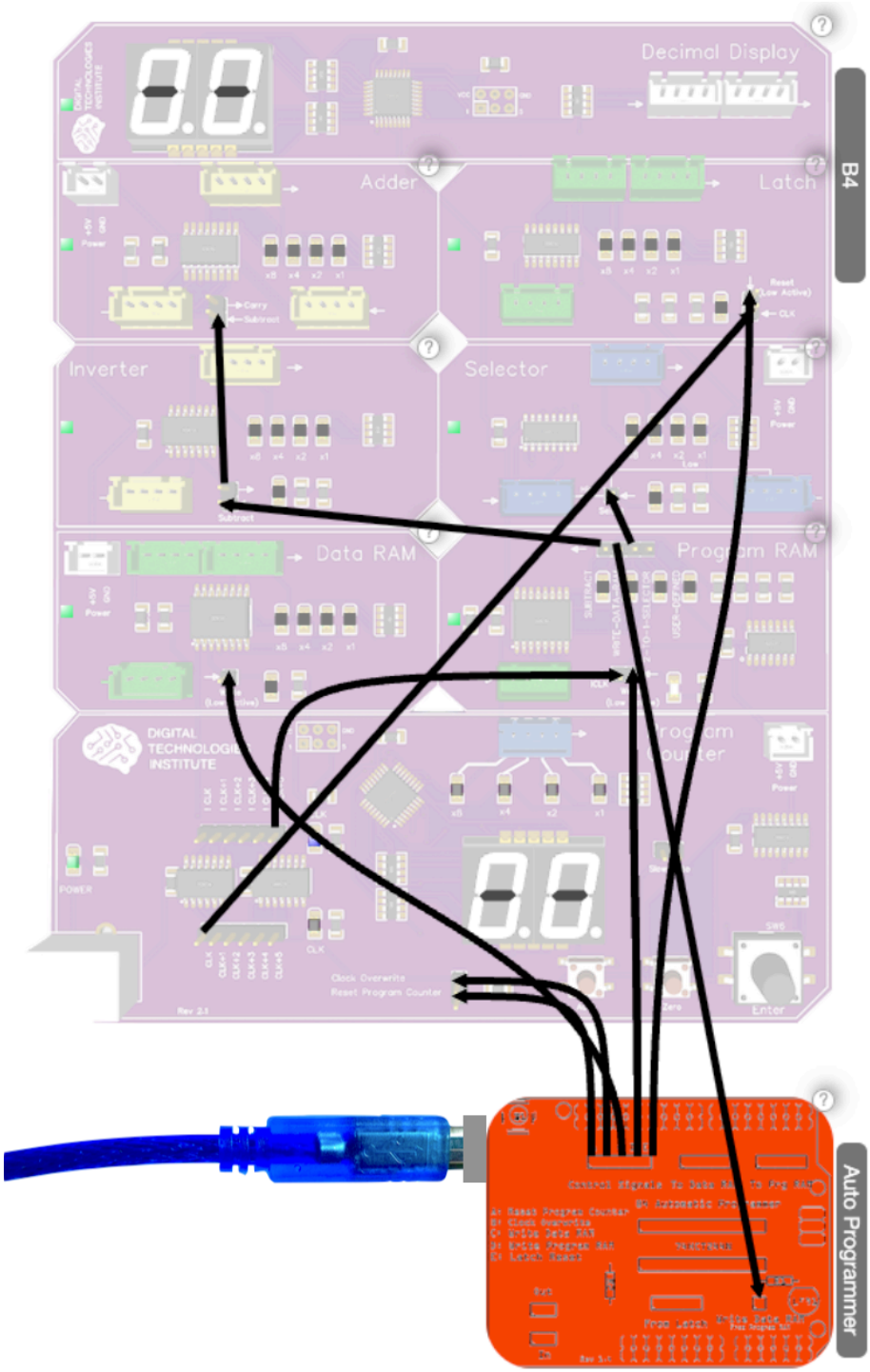
Then, we connect the data wires as shown in the following Figure. Most of the wiring will look familiar. The significant change is that the output of the Decimal Display (which is really the Latch content) is now routed to the Automatic Programmer. The Automatic programmer now also controls the Data and Program Ram Modules.



Setup of Mission 12: Data Wiring only

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.

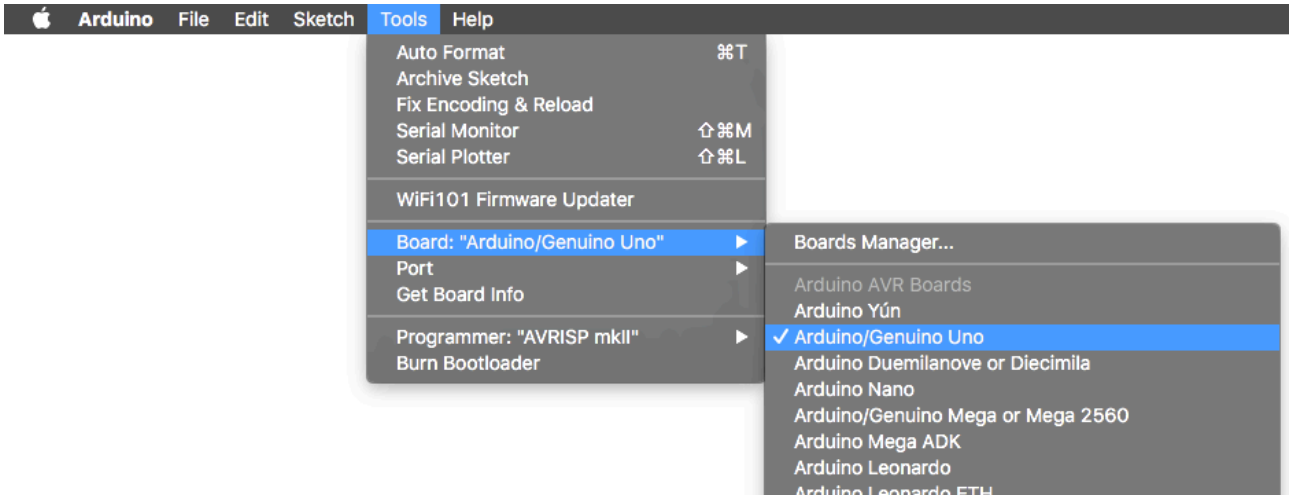


Setup of Mission 12:1-Pin Control Wiring only

Congratulations, we are done. 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.

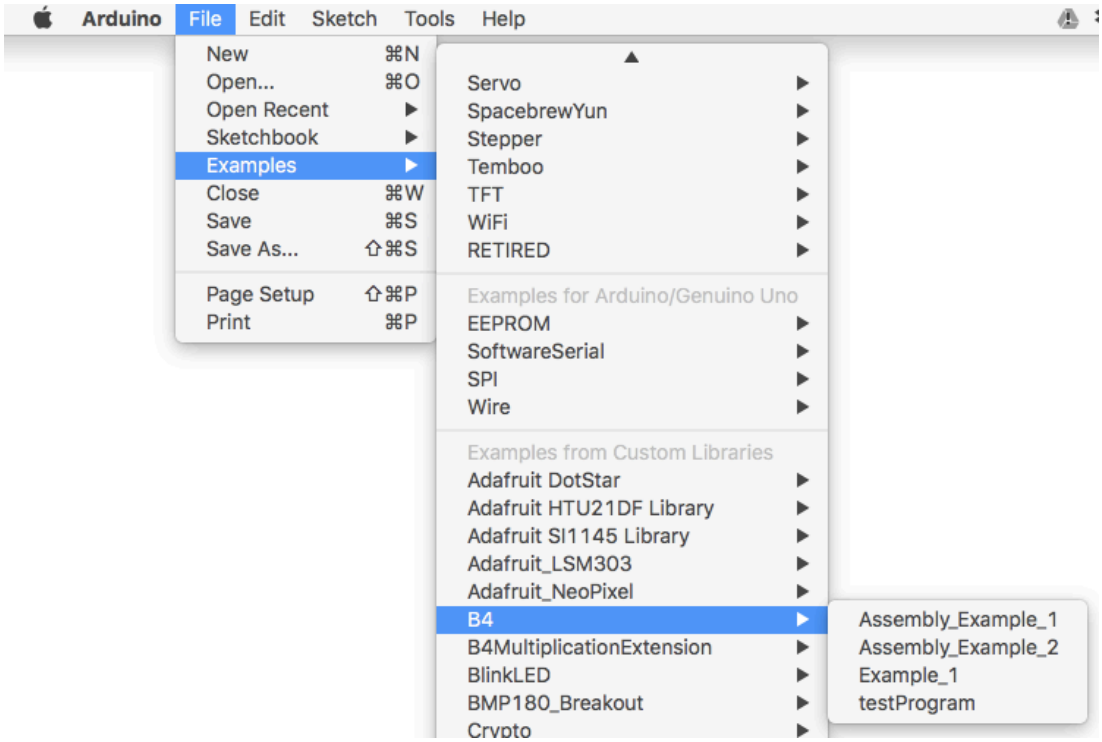


Arduino IDE: Selection of the Board

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

1. Download the B4 library from http://www.digital-technologies.institute/downloads
2. Locate the folder called B4-master and rename it to B4.
3. 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.
4. Then, restart the Arduino IDE and go into the File menu.
5. 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

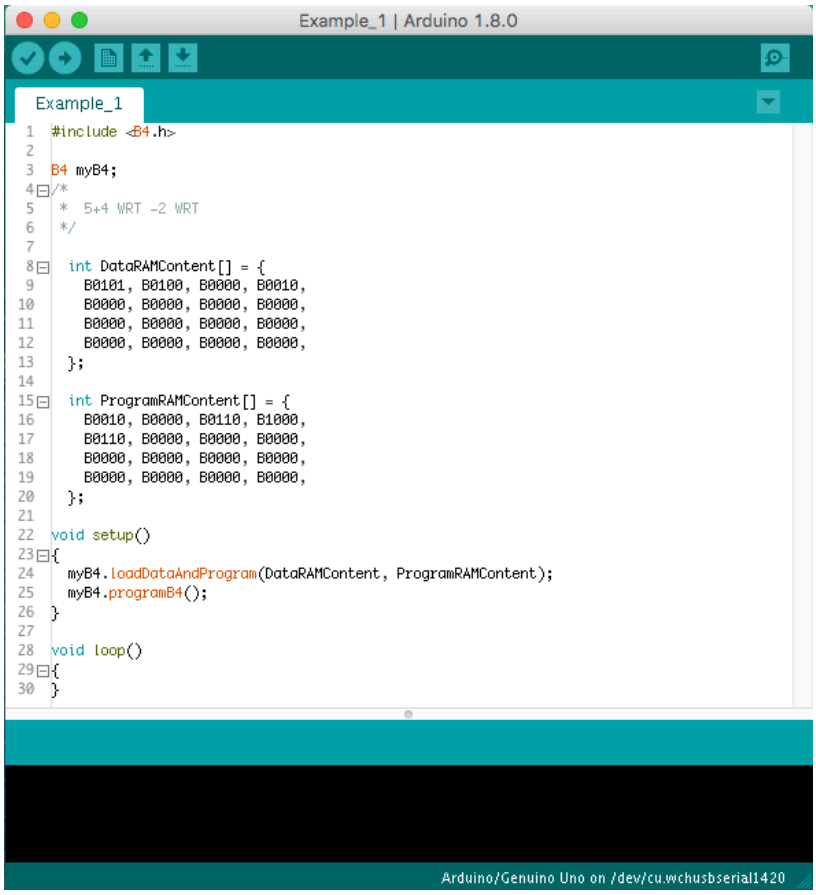
You just need to click on the second button from the top  to compile the code and upload it to the Arduino. Sit back and watch the LEDs of the B4 flashing as the program gets uploaded.

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 to verify that the wiring is all correct. 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 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 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.

Mission 13: 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. 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

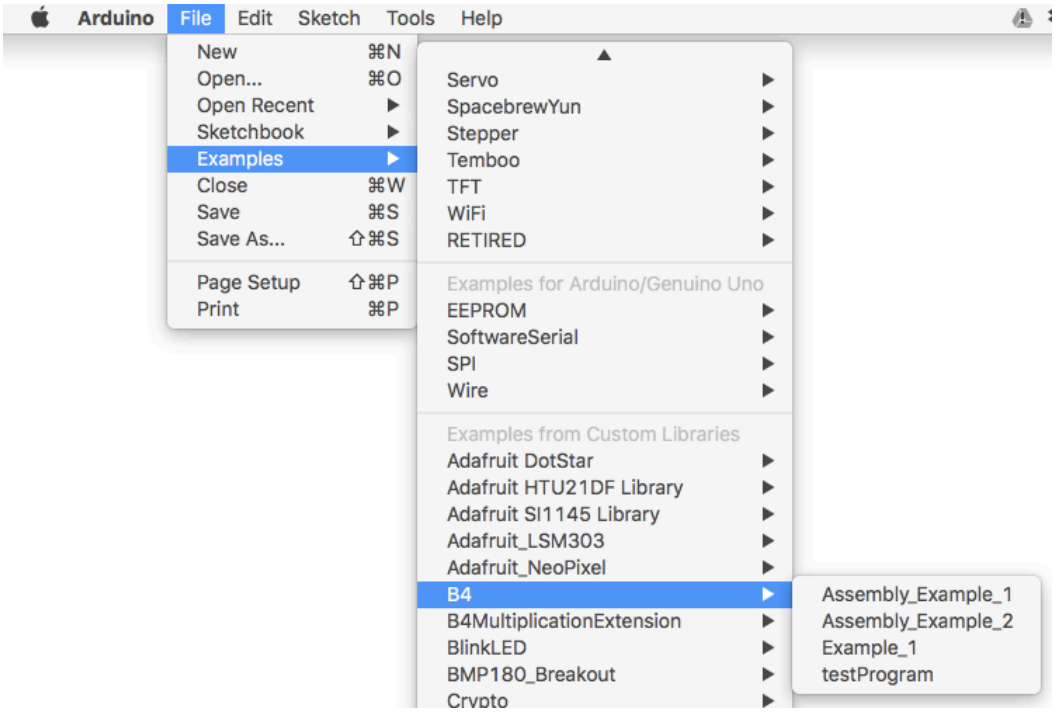
Matching Assembly Language with Machine Code

You notice that LOAD, ADD, and SUB have just one active bit, whilst WRT has two (B0**11**0). 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 Selector, we apply a clever little trick that 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 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.

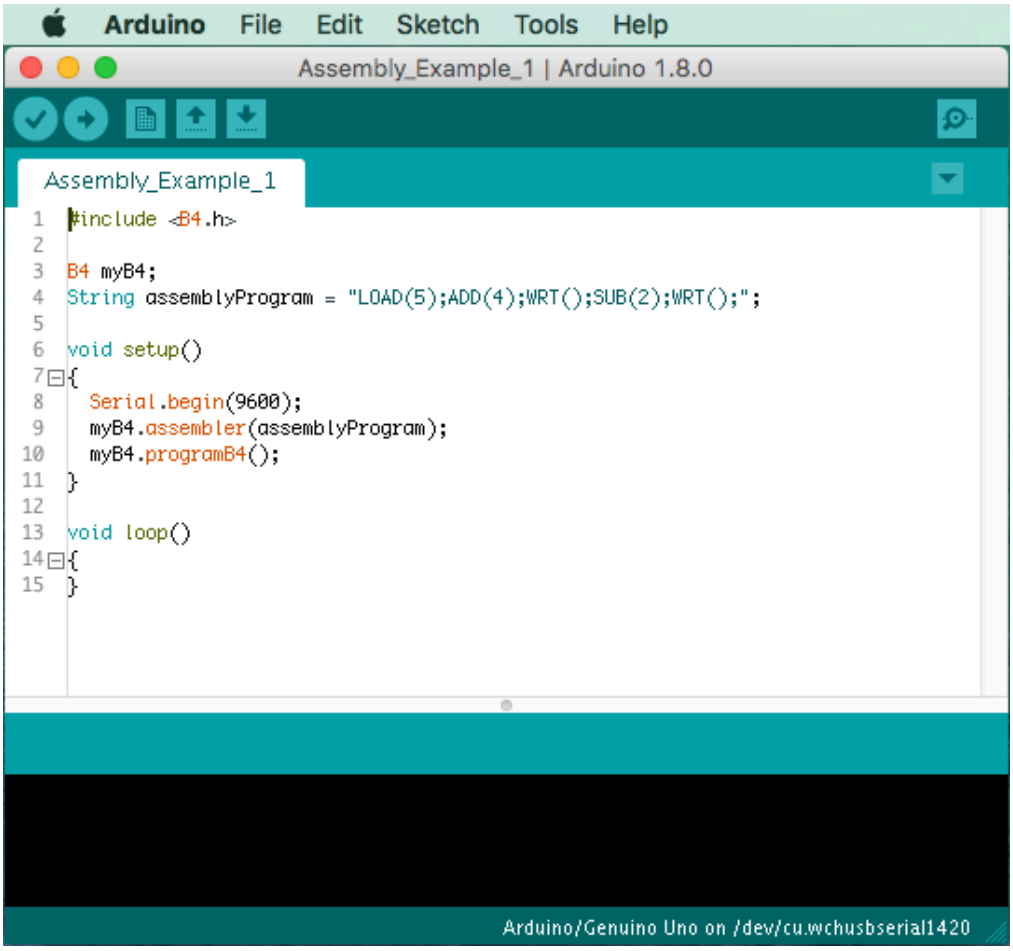
The function that performs the above-mentioned tasks (and more) is called *assembler* and is part of the B4 library. If you would like to learn more about its details, you can open the file B4.cpp in the Arduino/libraries/B4 folder.

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:



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 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.

Checkpoint Challenges 12	
?	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 mission, 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 Selector's output is from the Data RAM.

This mission 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.

Mission 14: On the Role of Timing

In the previous missions, 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 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, 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 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 by 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, the length of wires decreases.

Mission 15: 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 mission 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

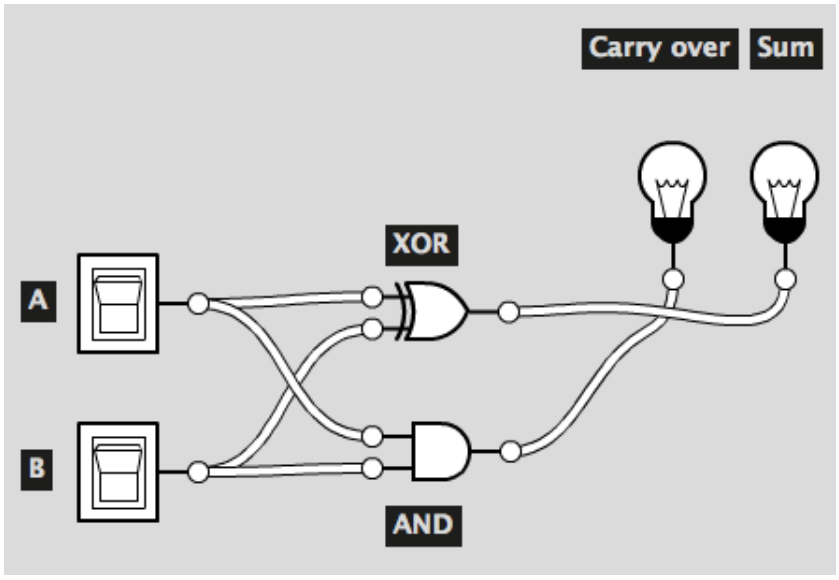
A	B	sum (higher bit) carry over	sum (lower bit)
1	0	0	1
0	1	0	1
0	0	0	0

Adding two binary numbers

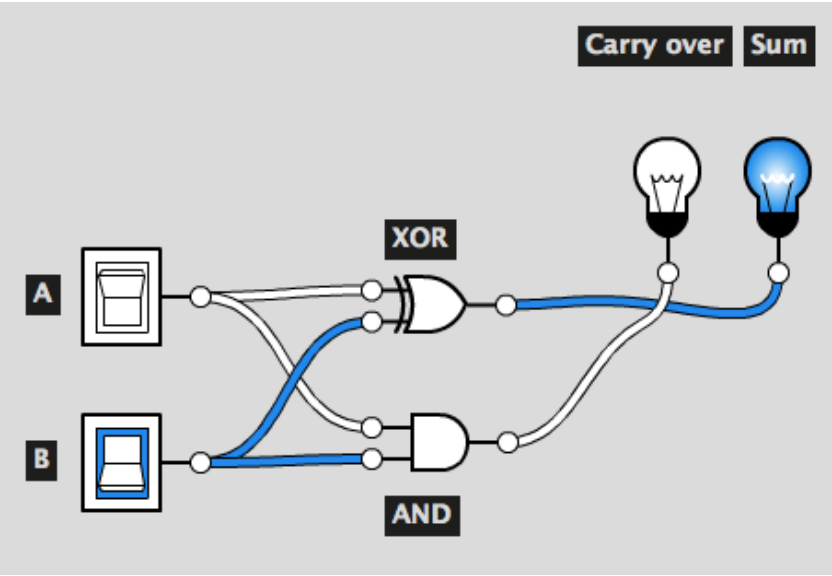
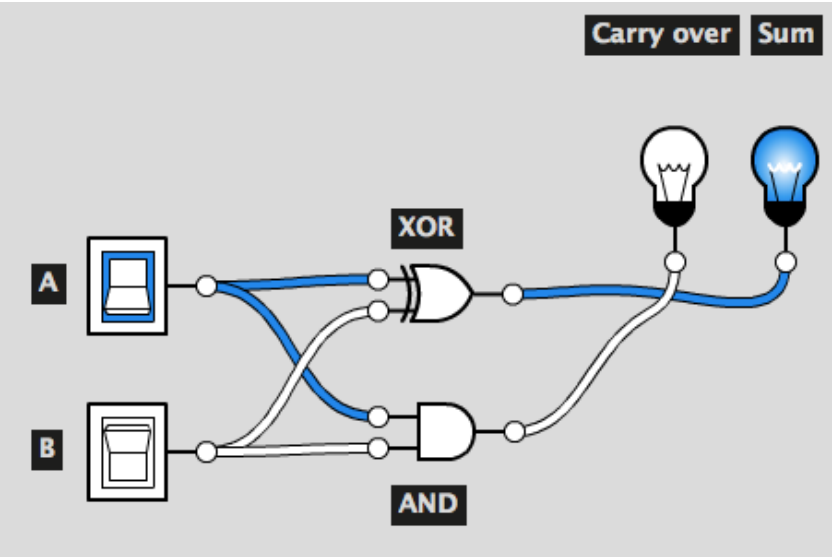
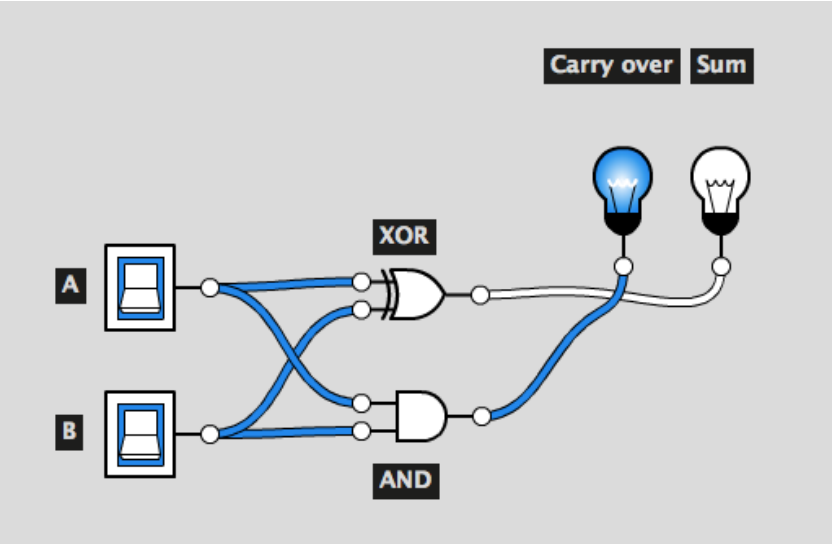
So, the sum's lower bit is 1 when either A OR B are 1, but not when both or none of them is 1: So we write: A XOR B

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+0=00

	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

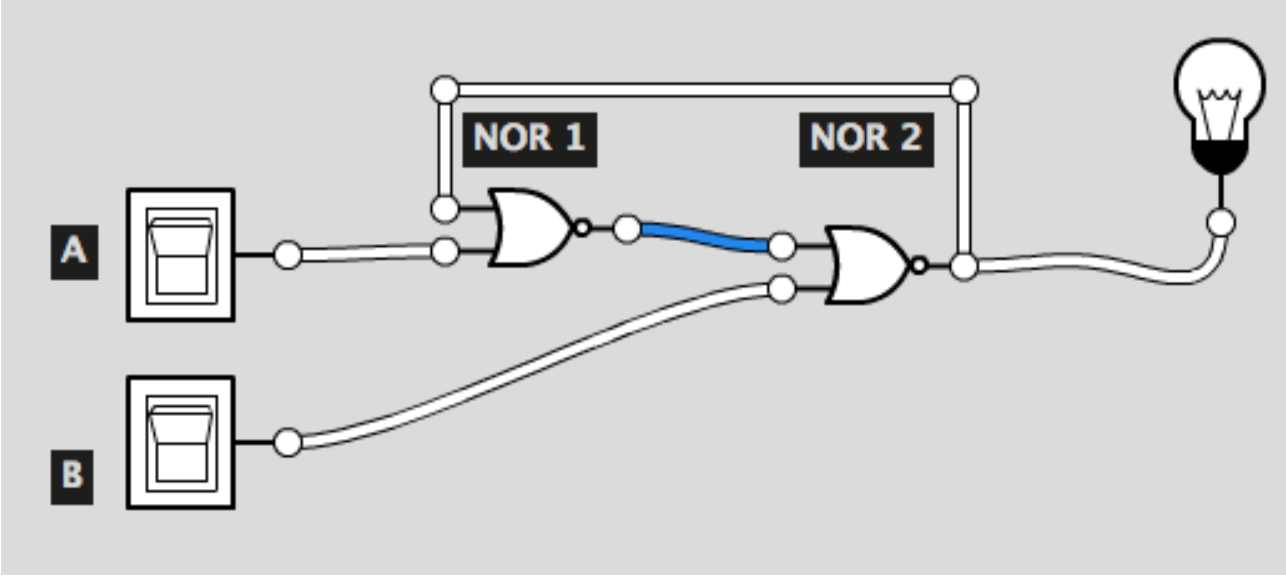
Logic gates can not only add, but also remember. Memory, as you have seen throughout this handbook, when we explored the Latch and RAM modules, is a fundamental function of a computer.

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

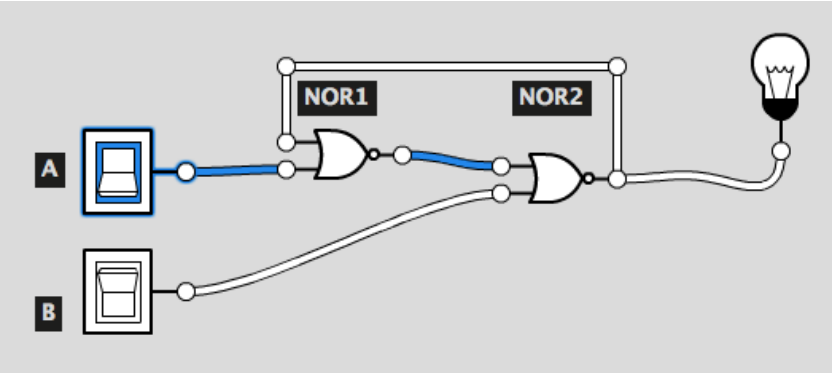
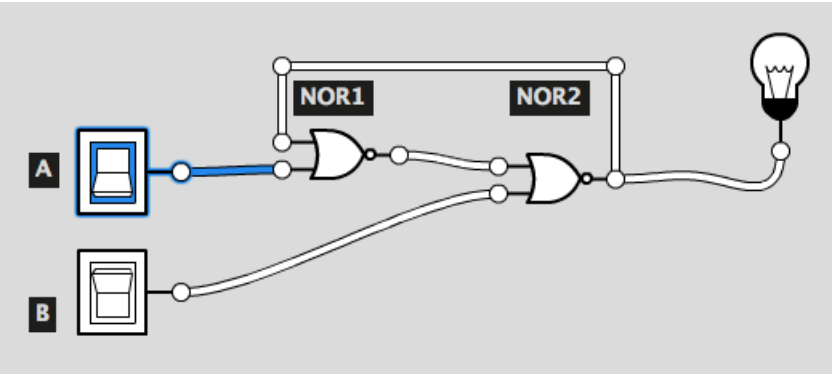
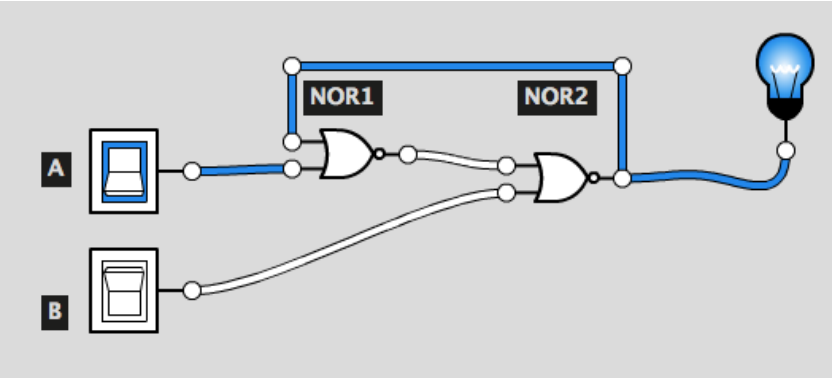
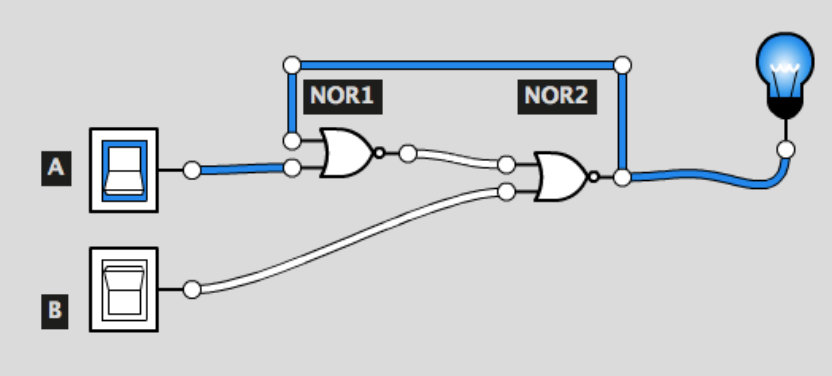
NOR truth table

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.

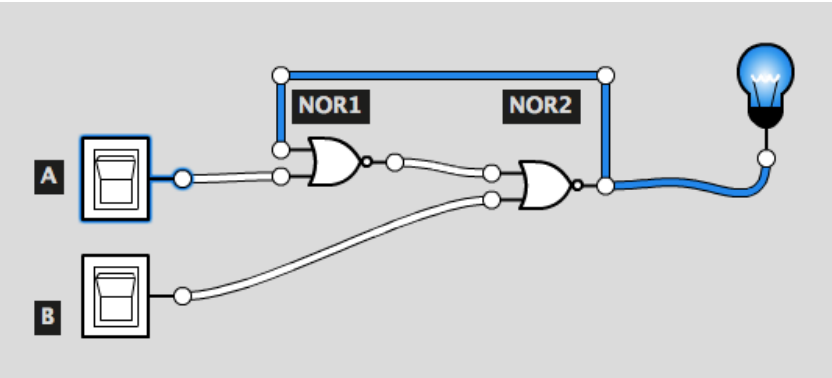
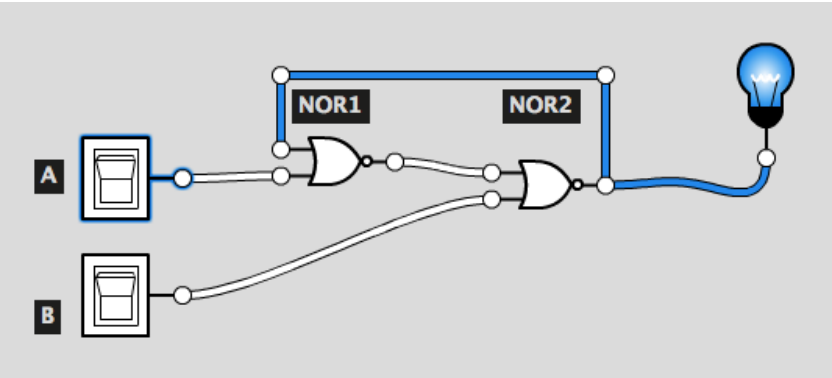


A Feedback Circuit with two NOR gates

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

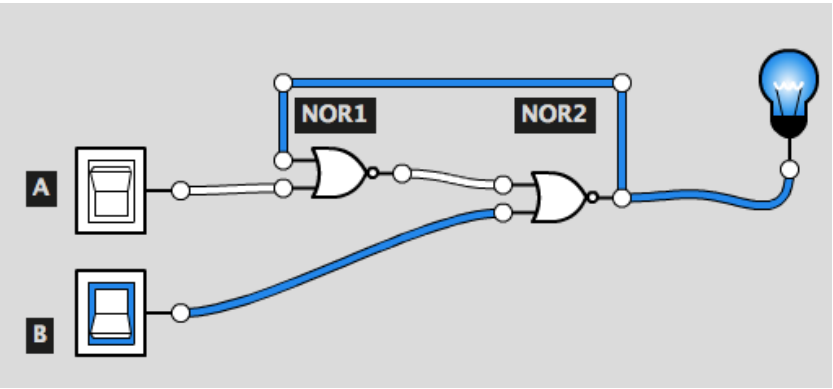
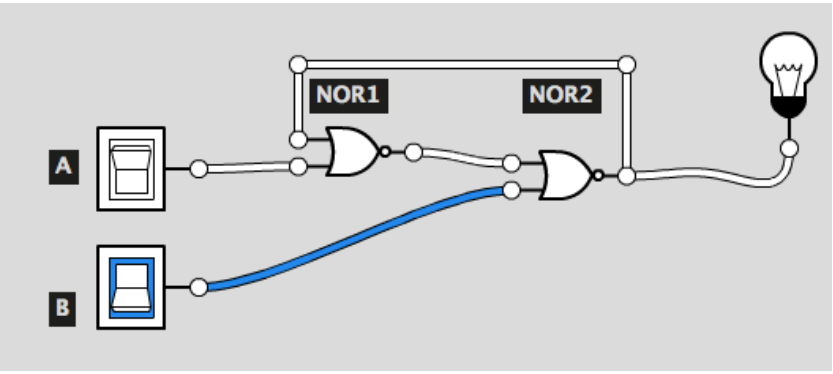
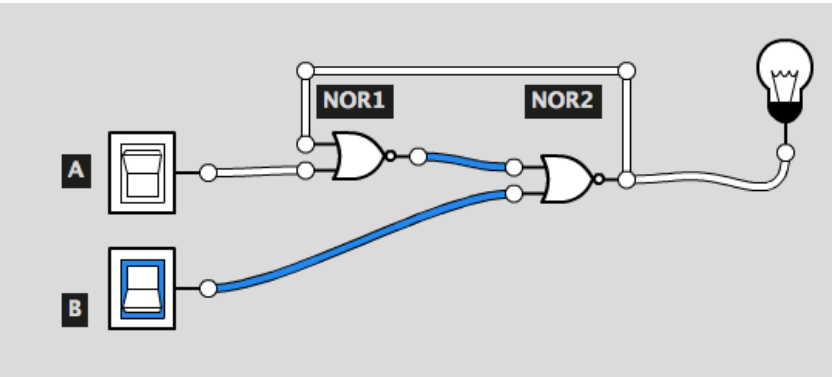
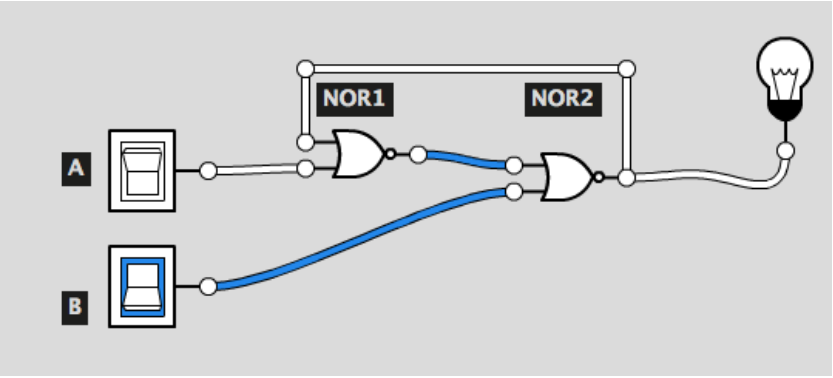
	Button A has been pressed. NOR1 received a 1 from button A and a 0 from NOR 2 ...
	... 0 NOR 1 is 0, which the NOR1 gate now gives to the NOR2 gate ...
	... 0 NOR 0 is 1, which the NOR2 gate now gives to the light bulb and to the NOR1 gate ...
	... and because 1 NOR 1 is 0, the NOR1 gate does not change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

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

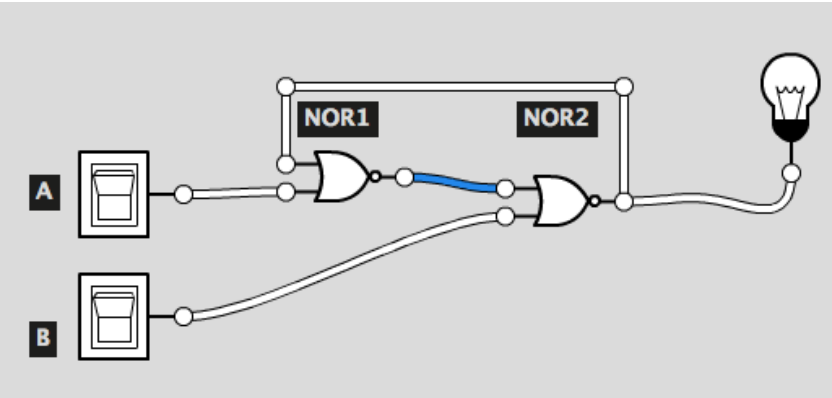
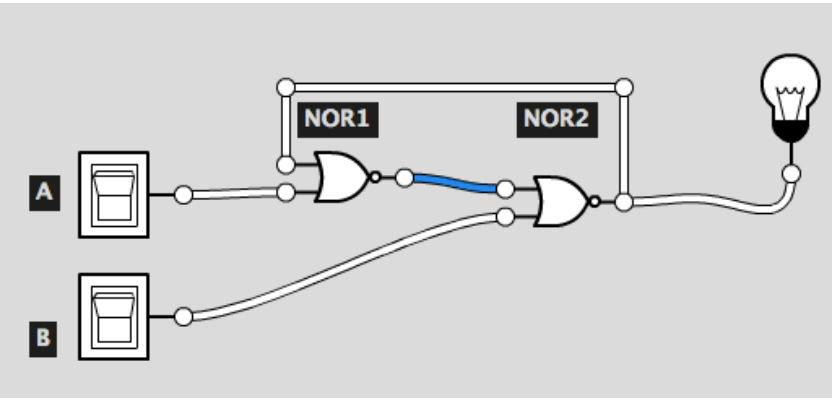
	Button A has been released. NOR1 receives a 0 from button A and a 1 from gate NOR2 ...
	... 0 NOR 1 is 0, which is the same result as 1 NOR 1 when the button A was still pressed. As a result, the output of NOR1 does not change and therefore the output of NOR 2 stays constant as well. The light stays 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.

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

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

	Button B has been released. NOR2 receives a 0 from button B and a 1 from gate NOR1 ...
	... 0 NOR 1 is 0, which is the same result as 1 NOR 1. As a result, the input of NOR2 does not change and therefore the output of NOR 2 stays constant as well. The light stays off.

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

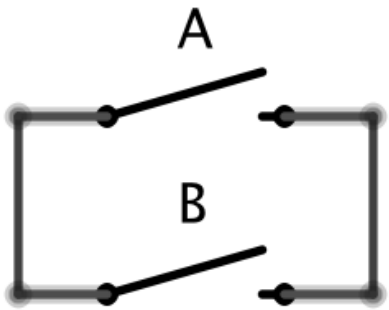
To this point, we have learned that we need different types of gates (AND, XOR) to make an adding machine, and other gates (NOR) to build memory. Each of these gates can be constructed of a cleverly-arranged set of little switches, called transistors. They have been around since the 1920’s, but developed 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

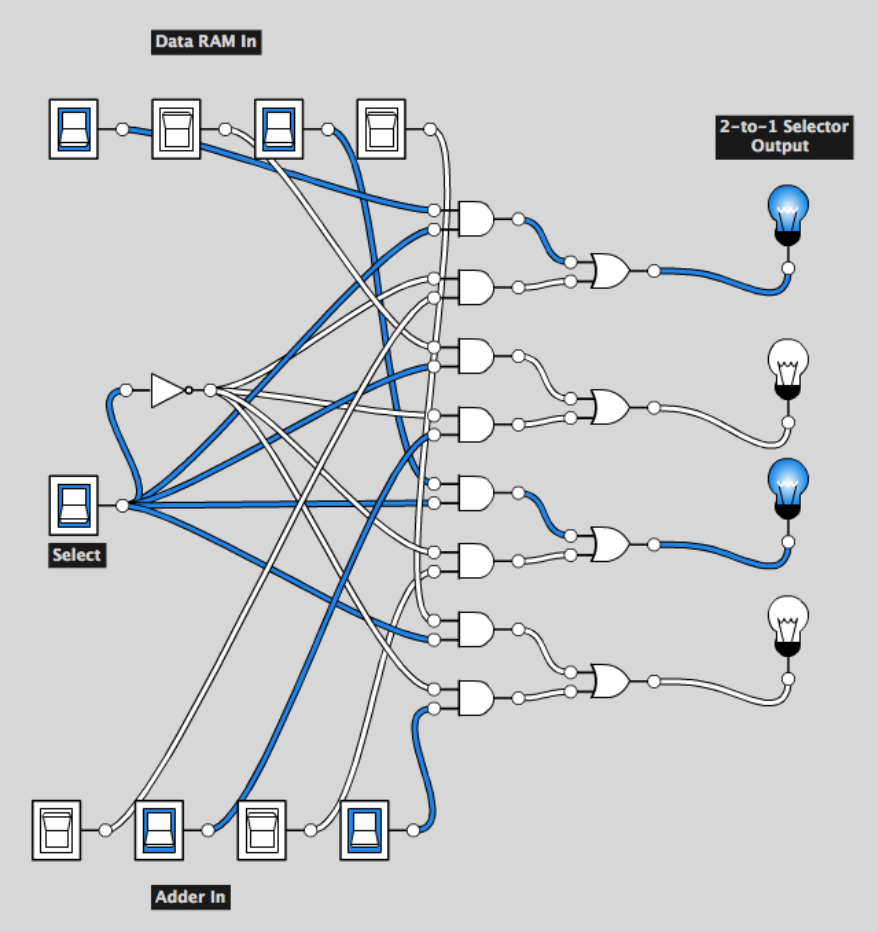
The circuit can only be closed by closing the switches A and B simultaneously.

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:



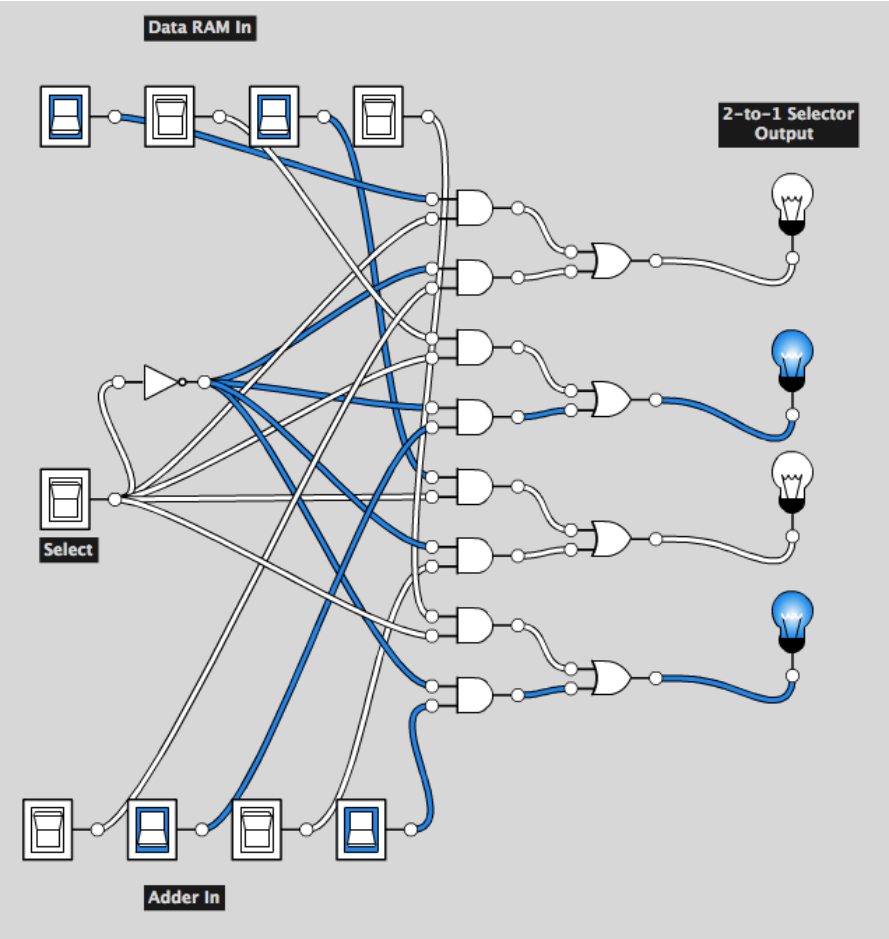
Realising an OR gate with two switches

Let's look at a concrete example of AND and OR gates: The B4'sSelector 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 Selector. Let's take a look:



Inside the 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 Selector. These are the same lights you see on the 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 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 the off position:



2-to-1 Selector (Select switch off)

According to Wikipedia, the largest transistor count in a commercially available single-chip processor in 2016 was 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 microcontroller, 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 mission, we have learned how the 2,500-year history of logic has led to a method of Boolean algebra, in which we can define logical functions known as 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.

Checkpoint Challenges 14	
?	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?

Mission 16: 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 the safe and reliable operation of the connected laptop computers.

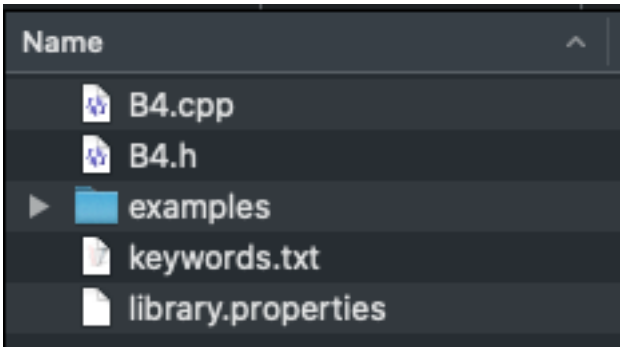
In this section, we are investigating two strategies of hacking into the B4. We call such strategies attack vectors. The better you understand a system, the better you can hack it. We begin by understanding the B4's software and then proceed to the functions of its hardware.

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 Mission 12, you installed the B4 Arduino library on your laptop or PC. If you haven't done this yet, go back to Mission 12 and follow the installation procedure. You will need the B4 library for this mission. 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.

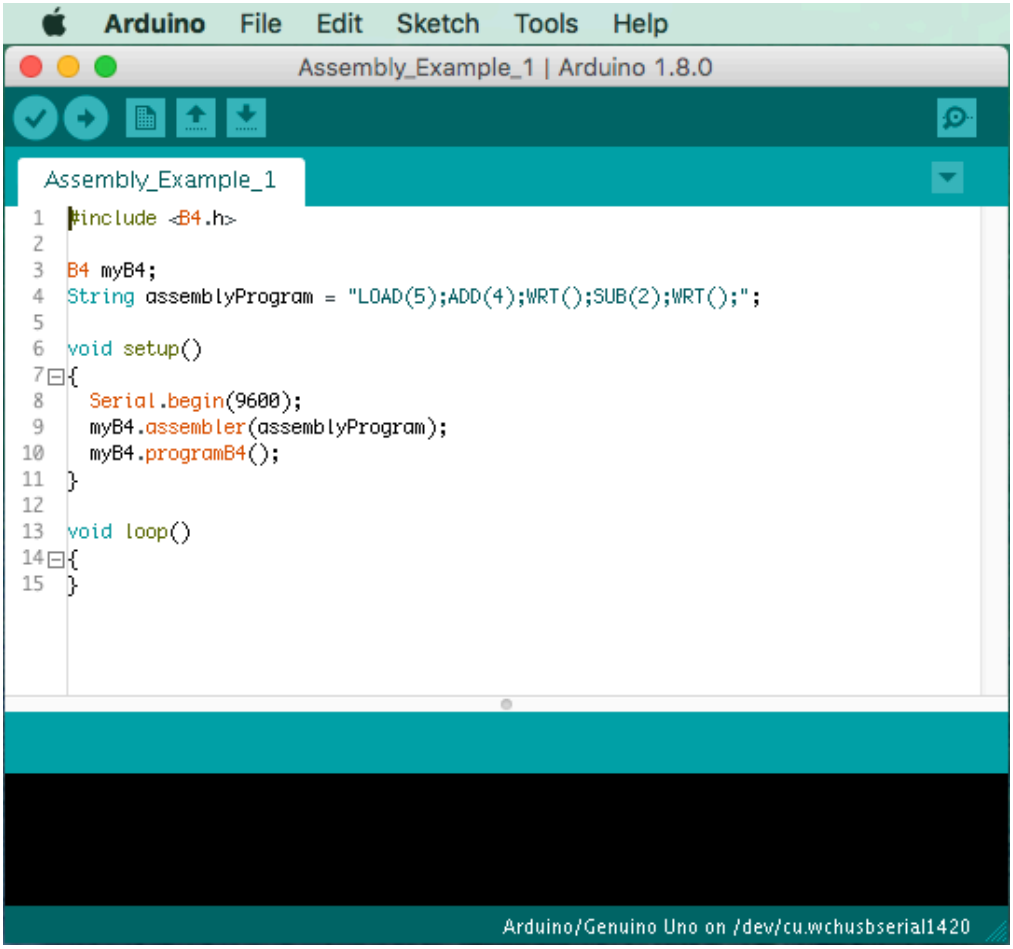
Find the Arduino libraries folder. On Windows and Macintosh machines, the default name of the folder is "Arduino/libraries" and is located in your Documents folder. In there you find a folder called B4. Open it.



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 Mission 13, you notice that we send the *assemblyProgram* string to the *assembler* function with myB4.assembler(assemblyProgram)



Assembly example from Mission 13

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 Mission 13, 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 MIssion 13 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 semicoli (;) 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

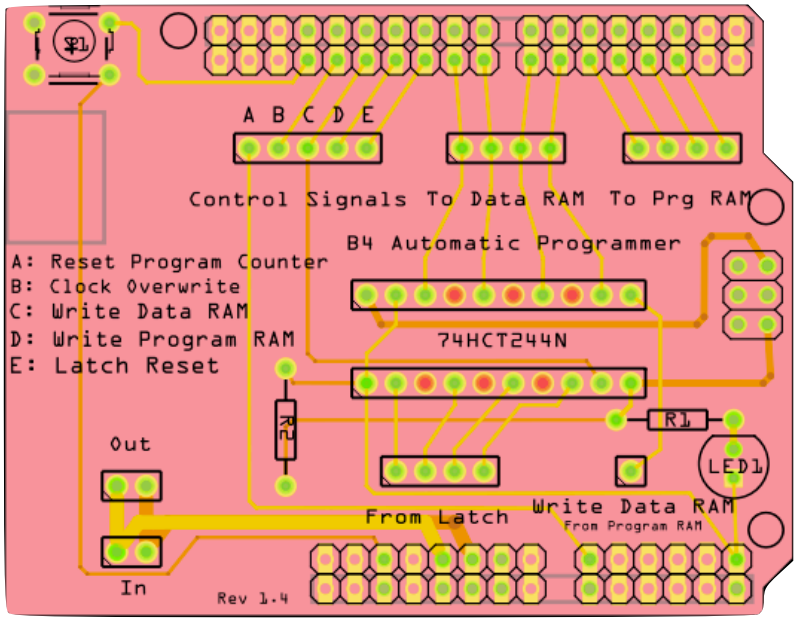
```
LIZARD(5)!APPLE(4)!WOMBAT()!SAUSAGE(2)!WOMBAT()!
```

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.



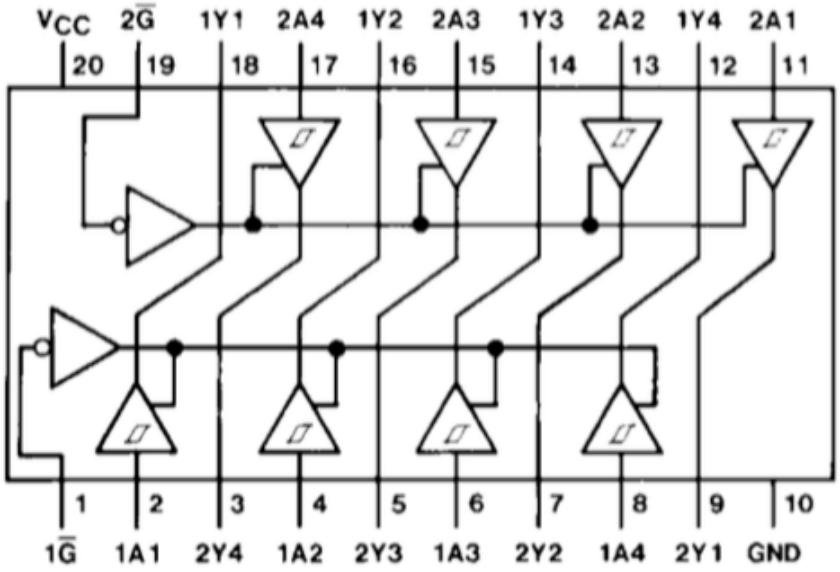
Automatic Programmer circuit board

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 seperate 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 the corresponding functions 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 clockCyle 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)
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 mission.
	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.

Checkpoint Challenges 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).

Checkpoint Challenges 2		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.

Checkpoint Challenges 3	Calculate in binary	Solution
?	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)

	2 minus 3. What do you see?	The binary complement of 3 (0011) is 1100
		  0010
		+1100
		-------- 	
		  1110
		+ 1
		-------- 	
		  1111
		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.

Checkpoint Challenges 5		Solution
?	Store the number 0111 (decimal 7) on floor 5. Move away and come back to check it. Does it remain?	Yes, the value remains as long as the Data RAM is powered or the address gets overwritten by another value
	If you wanted to store a high score in a game, would you use a Latch or the Data RAM? Why?	You would use the Data RAM, because the Latch's content is changed on every program step. It is not meant to hold data for a long time.
	If you wanted to store the value 0 on floor 0, value 1 on floor 1 … until the value 15 on flor 15, how would you do this automatically, just using the Ptrogram Counter and the Data Ram Module?	1) Remove the Variable. Connect the output of the Program Counter to the input of the Data RAM with a 4-in wire. The Data RAM is activated by a LOW signal, which is what the !CLK signal provides. 2) Connect the !CLK signal of the Program Counter to the Write pin of the Data RAM, using a 1-pin wire. Now, every time you press the Enter-button on the Program Counter, the value will get written into the memory cell at that address. 0->0, 1->1, … 15->15

Checkpoint Challenges 7a		Solution
?	Try 2+6+7. What's the final result?	1) First sum: 2 + 6 = 8 (binary 1000) → press Enter to store in the Latch. 2) Reuse a Variable: set it to 1000 (8), set the other Variable to 7 (0111). 3) Final sum: 8 + 7 = 15 (binary 1111). ✅
	What happens if you forget to store the first sum in the Latch before changing a Variable?	You lose the intermediate result. This means you have nothing to enter into the variable. The process stops. Moral: press Enter to clock the Latch before updating the Variables.
	Could you extend this method to add four numbers? How?	Extending to four numbers (A+B+C+D) with the same setup We repeat the store-and-copy cycle: 1) Set Variables to A and B → press Enter → Latch = D_1 = A+B. 2) Copy D_1 from Latch into one Variable, set the other to C → Adder shows D_1+C → press Enter → Latch = D_2. 3) Step 3: Copy D_2 into one Variable, set the other to D → Adder shows R = D_2 + D (i.e., (A+B)+C+D). 4) It's just "add two, store, copy, add next, store, copy, …" until you're done.

Checkpoint Challenges 7b	Try adding four numbers. 1+2+4+5.	Solution
?	What's the final result?	First Addition: 1+2=3 Second Additon: 3+4=7 Third Addition: 7+5=**12**
	How often do you need to move the endpoint of the select wire?	Only once — after the first addition. • During the first addition, the Adder must take its right input from the Variable (so Select = LOW). • After storing that first sum in the Latch, flip Select to HIGH. • From then on, all subsequent additions reuse the intermediate result (from the Latch) to be routed to the right input port of the Adder.

Checkpoint Challenges 7c		Solution
?	You just computed 1 + 2 + 4 = 7. Now add another number from RAM (e.g., 5). What do you expect the new total to be? Try it and see if your prediction was correct.	• Prediction: After 1 + 2 + 4 = 7, adding 5 should give 12. • Try it: The Adder computes Latch(7) + RAM(5) = 12. ✔ Final result = 12.
	Why do we only need to flip the Select switch once, after the first number is loaded into the Latch?	• During the first addition, the Adder must take its right input from RAM, while the Latch is still empty. That's why Select = HIGH (so the Latch loads the RAM value). • After that, the Latch always holds the running total. Now the Adder must take its right input from the Latch instead. That's why we flip Select = LOW once, and never touch it again.
	In a real CPU, why is it useful to separate the LOAD step (into a register) from the ADD step?	• If we didn't separate LOAD and ADD, every RAM value would be combined with whatever was already in the Latch — even when we wanted a "clean" new number. • Worse, if we wrote results back into RAM, we'd end up doubling values (RAM + Latch showing the same thing). • By forcing LOAD → ADD, CPUs avoid these errors and can safely write results into RAM without corrupting future calculations.

Checkpoint Challenges 8		Solution
?	Right now, our Program RAM has just one instruction: LOAD the first number into the Latch. After that, the Selector stays on ADD, so every new number from Data RAM is added correctly. But here's the puzzle: 👉 What would happen if you accidentally programmed two LOAD instructions in a row at the start of Program RAM? • Would the additions still work? • Why or why not?	If you LOAD twice, the second LOAD will overwrite the first number already sitting in the Latch. That means the first number gets lost, and the addition won't work properly. This shows why programs have to be written carefully: each instruction has a purpose, and even a small mistake (like an extra LOAD) can break the calculation.
	So far, our computer always reuses the result from the Latch for the next addition. That's why it can keep chaining numbers like 1 + 2 + 4. But what if you wanted to do two separate additions instead? For example: • Add 1 + 2 (and get 3) • Then separately add 4 + 5 (and get 9) 👉 Can you think of a way to program this so the second addition does not reuse the first result?	To do two independent additions, you need to start the second calculation with a new LOAD instruction. • The first LOAD puts "1" into the Latch, then the program adds "2" to get 3. • A second LOAD then replaces the Latch with "4", so the next ADD brings in "5" to make 9. To do this, you program the Data RAM with 1, 2, 4, and 5. The Program RAM is programmed with 0010, 0000, 0010, 0000. That's LOAD, ADD, LOAD, ADD.

Checkpoint Challenges 9

Create a Program Table for the addition 3 + 4.

	Data RAM				Program RAM				Comment
Step #	**3**	**2**	**1**	**0**	**A**	**B**	**SEL**	**D**	
Steps 2-15	0	0	0	0	0	0	0	0	do nothing
Step 1	0	1	0	0	0	0	0	0	Send 0100 to the Adder, which adds it to the 0011 stored in the Latch.
Step 0	0	0	1	1	0	0	1	0	Load 0011 from the Data RAM into the Latch. This is the first number for the Adder

Bonus: Can you extend your table so that the computer first calculates 1 + 2, and then separately calculates 3 + 4 (two additions, not chained together)?

The trick here is to have a separate SEL=HIGH at program step 2. This loads the latch with 3 and therefore interrupts the chain of additions.

	Data RAM				Program RAM				Comment
Step #	**3**	**2**	**1**	**0**	**A**	**B**	**SEL**	**D**	
Steps 4-15	0	0	0	0	0	0	0	0	do nothing
Step 3	0	1	0	0	0	0	0	0	Send 0100 to the Adder, which adds it to the 0011 stored in the Latch.
Step 2	0	0	1	1	0	0	1	0	Load 0011 from the Data RAM into the Latch. This is the first number for the Adder
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

Checkpoint Challenges 10

Question: Your current program shows how subtraction works with the Inverter. Now, change it to calculate 11 – 3 – 2.

Answer: Here is the program:

	Data RAM				Program RAM				Comment
Step #	**3**	**2**	**1**	**0**	**SUB**	**B**	**SEL**	**D**	
Steps 3-15	0	0	0	0	0	0	0	0	do nothing
Step 2	0	0	1	0	1	0	0	0	Activates the Inverter. Sends the binary complement 1101 to the Adder. 8-2=6
Step 1	0	0	1	1	**1**	0	0	0	Activates the Inverter. Sends the binary complement 1100 to the Adder. 11-3= 8
Step 0	1	0	1	1	0	0	1	0	Load 11 from the Data RAM into the Latch. This is the first number for the Adder

Question: Imagine you didn't have Program RAM. Which wire would you have to manually move in order to make the subtraction work? Why is letting Program RAM handle this better?

Answer: You would have to manually move the Inverter control wire each time you wanted to switch between addition and subtraction. Program RAM does this automatically at the right program step, making the process reliable and freeing you from manual rewiring.

Checkpoint Challenges 11

Question: Program the Data RAM so the computer calculates 6 + 3 and stores the result in Data RAM. Which step should WRT be set to 1? What value ends up in the RAM at that step's address?

Answer: The write should happen after the addition at step 2 of our program. The value will be 6 + 3 = 9.

	Data RAM				Program RAM				Comment
Step #	**3**	**2**	**1**	**0**	**SUB**	**WRT**	**SEL**	**D**	
Steps 3-15	0	0	0	0	0	0	0	0	do nothing
Step 2	0	0	0	0	0	1	0	0	Stores the contents of the Latch in the Data RAM
Step 1	0	0	1	1	0	0	0	0	Sends 3 to the Adder. 6+3=9
Step 0	0	1	1	0	0	0	1	0	Load 6 from the Data RAM into the Latch. This is the first number for the Adder

Question: Run your program with WRT = 0 for every step. What changes in Data RAM? What does this tell you about the role of WRT?

Answer: **Nothing** in Data RAM changes; the Latch updates but Data RAM does not. **WRT** is the only signal that authorises a store—no WRT, no write.

Checkpoint Challenges 12		Solution
?	If you were to design a calculator, would you design WRT() to be B0110, or B0100?	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 Data Selector keeps feeding intermediate results to the Latch, which provides it to the Adder.

	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?
		To answer this question, we can conduct an experiment by running the following program:
		
		#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()
		{
		}
		After program step (3), the output of the Latch is B0010. This is not B0111 because theData 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.

Checkpoint Challenges 14		Solution
?	Compare your knowledge about transistors that form gates to what you know about biological systems. Can you identify similarities?	Transistors form gates, which form higher-level functions, such as arithmetics, memory, switching, etc. Similarly, cells form organs, which in turn form organisms.
	If transistors were made of mechanical parts that moved, rather than semiconductor materials, what disadvantages would this bring?	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 1cm^3 (the size of a sugar cube) of space and that we need another 1cm^3 of space around each relay for wiring, etc.. So 2cm^3 of space per relay. That would be 2 billion cm^3. for all our 1 billion relays. That's 2,000,000,000 cm^3 = 2,000 m^3, 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. 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.

Checkpoint Challenges 14		Solution
	How much does it cost to manufacture a microprocessor? What would be the price per transistor for this microprocessor?	Let's pick the XBox One processor which has 5 billion transistors. The XBox console'r retail price is about $350. Let's assume that the cost of the processor is maybe $50. So, the price per transistor is $50/5billion=$0.000 000 01 or 0.000001 cents, thats 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. 0.0003 divided by 0.000001 is 300. So, making a transistor in a chip is about 300 times cheaper than printing a letter in a newspaper. 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.

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 :-). 2x6=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 – Design Debate: Accumulator vs. Selector

When building a CPU, there are often many possible designs. Let's compare two popular options:

Option 1: Accumulator-Only Design (no Selector)

- **How it works:**
  - RAM feeds one side of the Adder.
  - The Latch (Accumulator) always feeds the other side.
  - The Adder's output is always fed back into the Latch.
- **Advantages:**
  - Very simple wiring.
  - Minimal number of components.
  - Works fine for repeated additions, as long as you don't write results back to RAM.
- **Problems:**
  - The Latch is always "in the loop." If you write the result back into RAM, the Adder sees the same value from both sides (RAM + Latch) and doubles it.
  - No flexibility: you can't choose between "fresh" values from RAM and intermediate results.
  - Historically, accumulator-only CPUs existed, but they were quickly replaced as programs became more complex.

Option 2: Selector + Latch Design

- **How it works:**
  - The Selector decides whether the Latch should load a fresh value from RAM or the output of the Adder.
  - The Latch then feeds the Adder for further calculations.
- **Advantages:**
  - Clean separation between LOAD and ADD.
  - You can safely write results back into RAM without doubling problems.
  - Much closer to how modern CPUs operate, with explicit control over data flow.
  - Scales better for more complex operations.
- **Disadvantages:**
  - Slightly more complex wiring.
  - At first, harder to understand why the Selector is needed.

So Which One Wins?

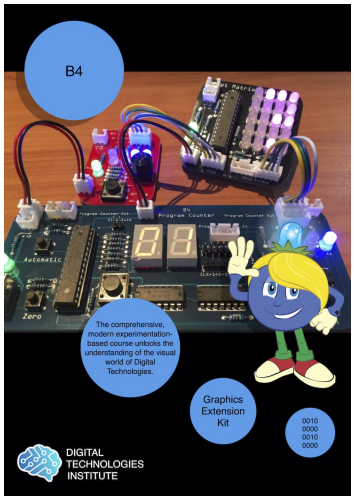
- **Accumulator-only** is like a one-trick pony: neat and minimal, but it quickly runs into problems.
- **Selector + Latch** may look more complicated at first, but it avoids hidden traps and sets you up for more advanced designs.

👉 This is why our B4 computer uses the **Selector + Latch** approach from Mission 7c onwards. It's the design that makes real computers work reliably, millions of times per second.

Appendix E: Extension Kits

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

Graphics Extension Kit



This extension kit adds graphics output capabilities to the B4 Spark. 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.

Computer Memory Kit

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 for the Data RAM module in the B4 Spark Computer Processor kit.

Appendix F: 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

B4 Opcodes at a Glance

	Bit 3 (SUB)	Bit 2 (WRT)	Bit 1 (SEL)	Bit 0 (free)
Name	Subtract	Write	Select	(free)
When it's 1…	The Inverter is active → Adder subtracts	Store the Latch value in Data RAM	Selector takes input from Data RAM	Reserved for student extensions
When it's 0…	Adder adds	No write into Data RAM	Selector takes input from Adder	Not used (for now)