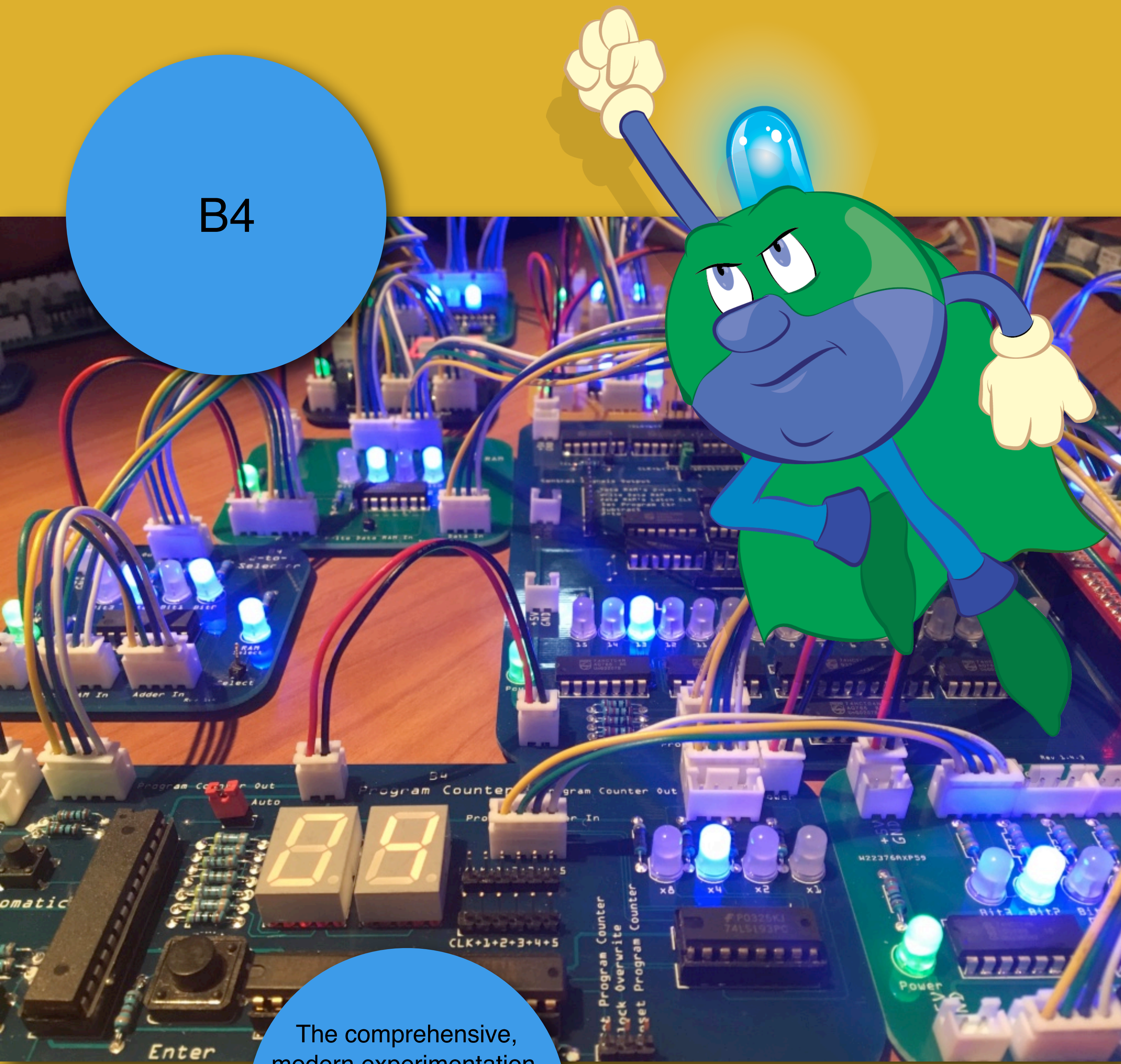
B4



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

Arithmetics Extension Kit

0010
0000
0001
1001

Digital Technologies Institute

“All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can’t get them together again, there must be a reason. By all means, do not use a hammer. ”

(IBM Manual, 1925)

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 electric device and should be handled as such. We recommend to treat is with care, and to keep it on a dry and level surface. Do not scratch the surface of the printed circuit boards with sharp or metallic instruments, as this might damage the wires.

Acknowledgements

We would like to thank Charles Petzold, the author of ‘Code: The Hidden Language of Computer Hardware and Software’, published in 1999. His book has both inspired and guided the design of the B4. We also recommend reading it either during or after students have been working through this experiment plan.

We would further like thank Henrik Maier from proconX for his guidance and feedback on the electrical engineering design, fabrication and component selection, which has been invaluable to transform 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 the Arduino community. Two of the B4’s module deploy an Atmega processor which run Arduino programs. Keep up the great work !

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

Included Parts

1x Control Unit
1x 2-Line-to-1-Line Selector
1x Random Access Memory 16x4 bit
3x Latch
1x Automatic Programmer Arduino Shield (Extended Version)
1x Splitter

10 x 4 Pin Wires
10 x 2 Pin Wires
10 x 1 Pin Wires

1x Student Handbook
1x B4 Arduino Library (available for download)

Not included
B4 Base Kit, sold separately
Arduino Uno or compatible (required for the full function of the Automatic Programmer Arduino Shield - it is already included in the B4 base kit)

Power Consumption:
5V, 200mA DC.

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

Welcome back, Parents and Teachers

The B4 Computer Processor (base) Kit explored the fundamental operation of computers with core functions, such as data storage, addition and subtraction with the underlying algorithmic design. This extension kit offers an exciting opportunity to dive deeper into the inner workings of a digital system by expanding the arithmetic capabilities of the B4 towards multiplication, division and beyond. In the process of implementing these capabilities from the ground up in a computer, students learn about loops, conditional jumps, data pointers and memory addresses. This curriculum follows the B4-style bottom-up motivational approach, in which computing concepts are introduced by need and motivated by context. Students gain an understanding of the practical necessity of computing concepts and learn about them from a hardware and a software perspective. The result is a deeper and more natural understanding of Digital Technologies.

Again, this kit has been designed with the new Australian Curriculum: Digital Technologies in mind.

Welcome back, Students

Congratulations on graduating from the B4 base course. By completing the experiments from the previous course, you have gained a solid understanding of how a computer really works inside. It is now time to apply our knowledge to teaching the B4 a few more tricks. With this new kit in front of you, we will expand the B4's capabilities in performing more algorithmic operations. We'll start with multiplication and division and go forward from there.

Besides following this study guide, we again encourage you to conduct your own experiments and try things that are not in this handbook. You never know what you might discover.

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

B4's Arithmetics Extension Kit Parts

The B4 Arithmetics Extension Kit consists of seven modules which extend the B4 base kit that you already have. The Automatic Programmer replaces the Automatic Programmer from the base kit.

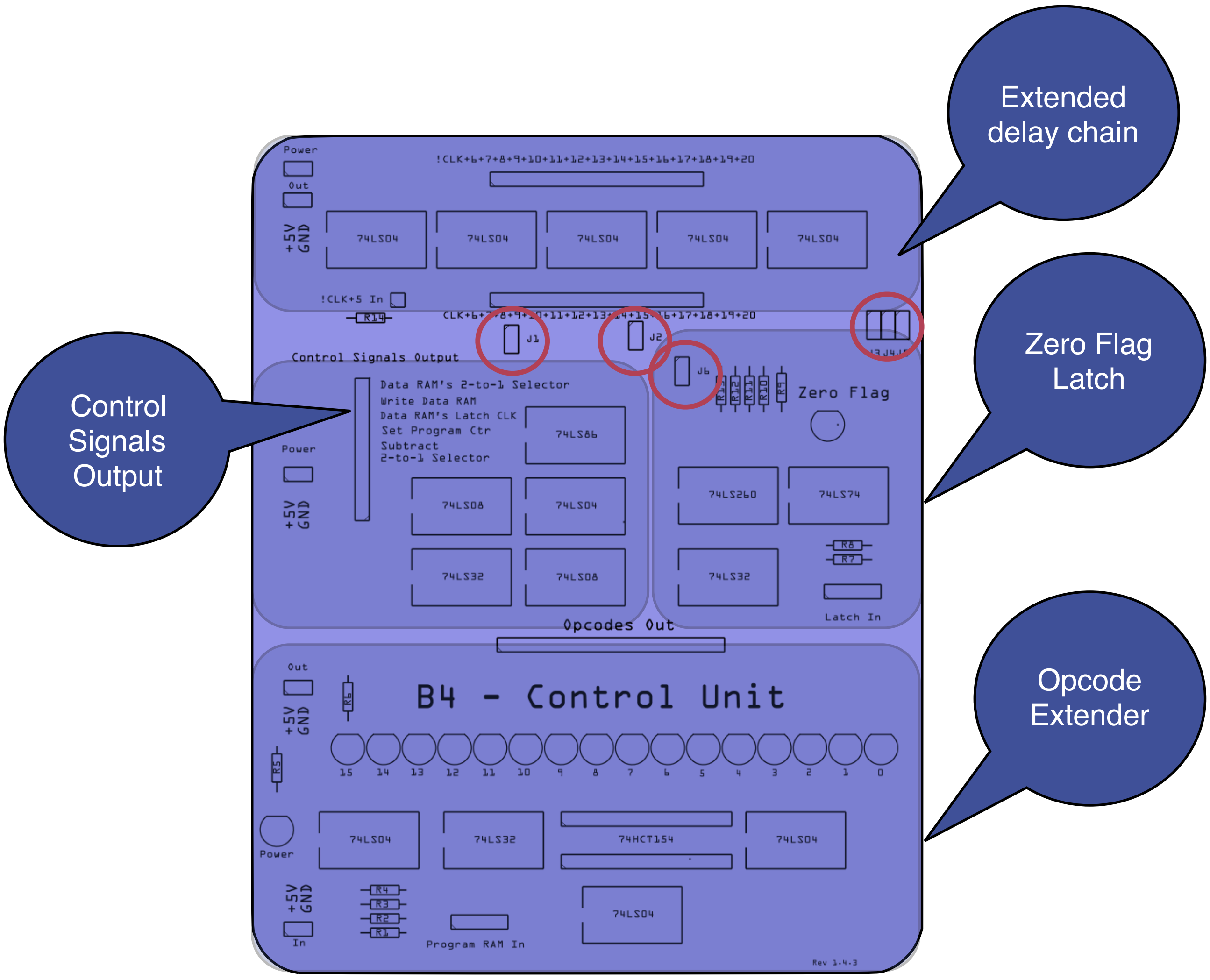
We can classify the B4 Arithmetics Extension Kit modules into control, memory & storage and programming. All modules within a category are of the same colour. The control modules are blue and the memory & storage modules are green. The programming module is red.

	Function	Color	Modules
Core Modules	Control	blue	Control Unit, 2-to-1 Selector, Splitter
	Memory & Storage	green	Data RAM, 3x Latch
Helper Module	Programming	red	Automatic Programmer

You might wonder “Hang on, this is an Arithmetics Extension Kit. Where are the yellow modules?”. This is a very valid question. As you will see, we will build a lot of clever logic around the existing arithmetic modules, namely the Adder and the Inverter.

The modules are all labeled. Take them out of the box and find each of the modules as we describe them below.

Core Modules



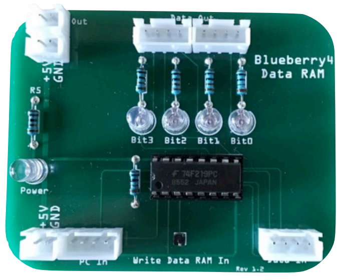
The **Control Unit**, shown above, is the largest of the B4 core modules. It performs four main functions:

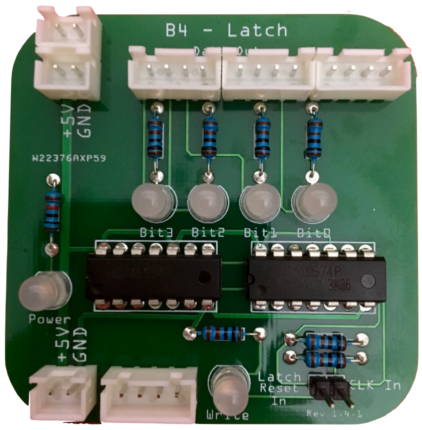
1) The **Opcode Extender** grows the number of opcodes from four to 15. This means that B4 can now support up to 15 different commands, allowing us to jump around in code, access data directly from memory, and more. Where we only needed 3 opcodes (LOAD, SELECT, and WRITE) in the base kit, we will need a few more for the experiments in this book.
2) The **Zero Flag Latch** is a single Flip Flop that remembers when its input was zero. This is useful to determine when a loop has run its course and should stop.
3) The **Extended Delay Chain** provides important support services. This includes an extended chain of inverter circuits to delay the CLK and !CLK signals, giving the B4 more time to synchronise its many new functions internally. For a refresher of these signals, see experiment #8 in the B4 Base Kit Handbook.
4) The **Control Signals Output** is the place where many of the opcodes are being translated into electrical signals. We will connect them to the other B4 modules during the various experiments.

There are six little electrical connectors called jumpers on this board, which we have circled in the figure above. They connect the various circuits to the Extended Delay Chain. They ensure that low-level functions are performed in the right order. The precise calibration is depends on the exact timing of the chips, which can vary between production runs. At the bottom right of the Control Unit you find a revision number, such as 1.4.3.

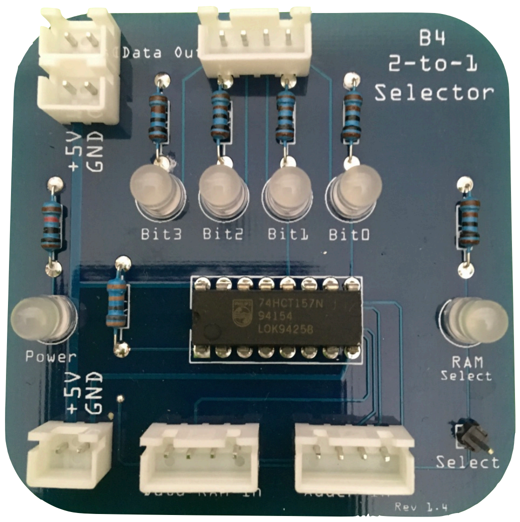
Please go to https://www.digital-technologies.institute/handbooks to download the Control Unit Calibration document, which contains the exact jumper settings for your Control Unit. You will need it for the experiments 8 and higher. All other experiments can be conducted with the jumpers installed.

The **Random Access Memory (RAM)** replaces the **Program RAM Module** from the B4 Base Kit. It is identical to the Data RAM Module from the B4 Computer Processor kit.

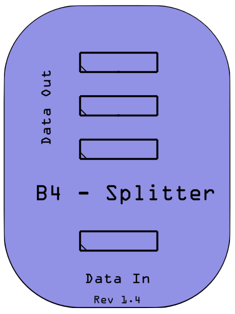




A **Latch** has the function of short-term memory in a computer. It stores (or 'buffers') some data before that data can be further processed. This Latch module is like the one in the B4 base kit.



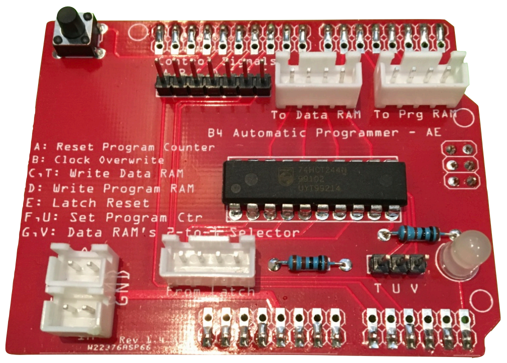
The **2-to-1 Selector** is a switch that can change the data paths in our computer. Ours is identical to the one from the base kit. We will use the second 2-to-1 Selector to enable the B4 to do relative data addressing.



The **Splitter** is a little passive module that allows us to split a single signal into three outputs. We use the splitter when we don't have enough 4-pin output ports on a module. Unlike the other modules, the Splitter does not require power.

Helper Module

The **Automatic Programmer** is the extended version of the Automatic Programmer from the Base Kit. Its main function of temporarily taking control of the B4 hasn't changed, but it can now also influence the Program Counter.



We now have a basic understanding of the new modules of our B4 Arithmetics Extension Kit. Don't worry if you haven't understood everything yet. We will revisit each module in more depth during the following experiments. In the box, there are some additional 1, 2, and 4 pin wires, that you are already familiar with from working with the base kit.

As a reminder, in the diagrams in this book, we use the following wiring notation. A solid line denotes a 2 pin power or 4 pin data wire. A line with two arrows denotes a 1 pin control wire. This is just to make the setup a little bit easier for you.

Symbol	Meaning
———	2 pin or 4 pin wire
◄———►	1 pin wire

Please look after me

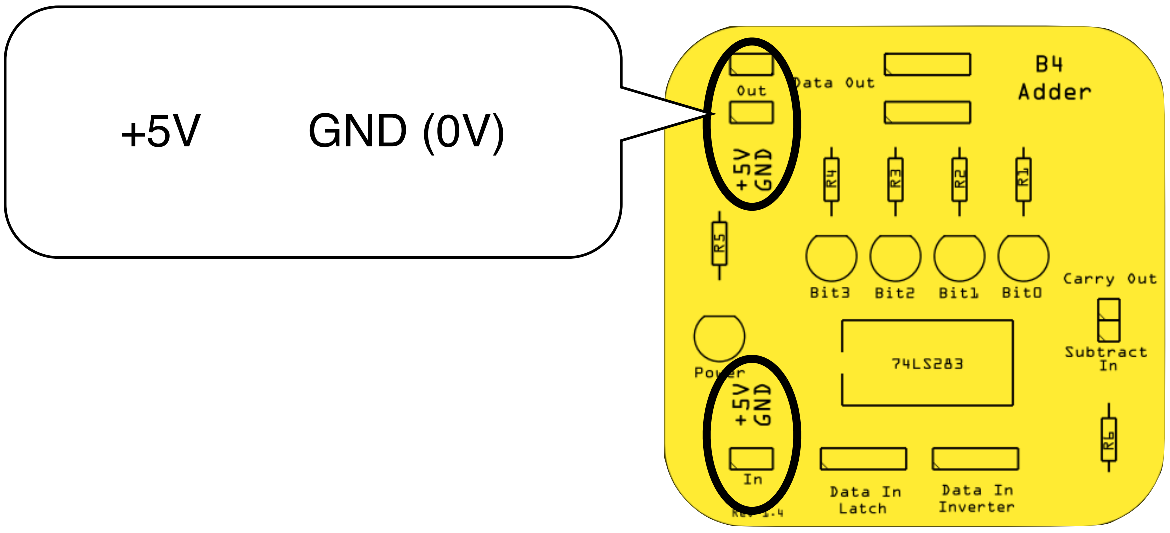
The B4 is fairly robust and will last a long time with proper care. As long as you don't plug wires into connectors they are not designed to go in and as long as you don't drop the modules, step on them or use them as a doorstopper, things should be just fine.
Always only plug the 2 pin wires into 2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

About Numbers

To be clear about the distinction of binary and decimal numbers, we add a capital ‘B’ to binary numbers. This way we can distinguish for example 11 (decimal eleven) from B0011 (decimal 3), or 10 (decimal ten) from B0010 (decimal 2).

A Word about Power

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



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

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

Experiments

Overview
In this handbook, we have prepared several experiments that will help you to build a computer that can perform multiplication, division and more. You will learn how loops work inside a computer and what data pointers are.

We recommend that the experiments be taken 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 is written in this handbook. Feel free to explore and try out different things as you like.

Experiment	Title	Learning Objectives
1	Multiplication - Exploring the Challenge	How a computer performs multiplication. Simple approach. Discussion of limitations. Design of a general-purpose approach to multiplication
2	Loops	Repetition of commands through unconditional jumping.
3	Knowing when to stop	Function of the Zero Flag Latch module. Teaching the B4 when to to stop a loop.
4	A Multiplication Program	Moving from absolute data values to data pointers. Extension of the B4's opcode repertoire.
5	Data Pointers	How a computer addresses memory by pointing at it.
6	Automating Data Pointers	Introduction of a Latch to enable data pointers at runtime of a program.
7	Automating Loops	Introduction of a Latch to enable loops at runtime of a program.
8	We are Building the B4/A	Hardware setup of a machine that can perform multiplication, division, averages, and more.
9	Programming the B4/A	We design, document and run a number of exciting algorithm, which apply our learnings from the previous experiments. This includes multiplication, division, and average algorithms. We conclude with an algorithm that computes Fibonacci numbers.
10	Summary and Conclusions	We reflect on what we have learned in this course

Experiment 1: Multiplication - Exploring the Challenge

Our goal is to extend the B4 architecture from the B4 base kit handbook so that our B4 can perform general multiplication and division. With general, we mean multiplications of any two natural numbers, within the limits of our 4 bit architecture. So we are limited to integers and the multiplication result must be less than 16. As multiplication and division are closely related, we assume that we can initially focus on multiplication and then look at what we need to do to let the B4 do division.

We have previously learned that a computer multiplies two numbers by performing a series of additions. For example, 3x5=5+5+5=15. So, to multiply three with five, we have to perform two additions: five *plus* five *plus* five.

How could we make this happen with a computer?

First attempt: We write a program that adds:

Line	Code
0	LOAD(5);
1	ADD(5);
2	ADD(5);
3	WRT();

This will definitely produce 15. All done, class dismissed. Everyone go home.

But what if we want to compute 4x3? We have to write another program:

Line	Code
0	LOAD(3);
1	ADD(3);
2	ADD(3);
3	ADD(3);
4	WRT();

Since 3x4 is equivalent to 4x3, we can make the program a bit shorter:

Line	Code
0	LOAD(4);
1	ADD(4);
2	ADD(4);
3	WRT();

Hang on, that's not very practical. We have to write a different program every time the numbers change. Also, the many ADD() statements are very confusing, as we have to manually keep count of how many we need. When we want to compute 2x5 we need 1 ADD() command, but for 3x4 we need 2 of them. What if we could do the following to calculate 3x4?

Line	Code
0	LOAD(0);
1	ADD(4). Do this 3 times
2	WRT();

If we then wanted to calculate 2x5, we would write:

Line	Code
0	LOAD(0);
1	ADD(5). Do this 2 times
2	WRT();

We see that we re-use the entire program and all we need to do is to replace the numbers in line 1.

When we look at the program, we see that our B4 can already carry out the LOAD(), ADD() and WRT() commands. However the 'do this 2 times' is new. We need to find out what, for example, 'ADD(5) 2 times' means for a computer and how we could make it happen.
Let's break this into pieces:

1) ADD(5)
2) 2
3) times

Let’s pause here for just a moment to learn some important terminology.

In 3x4, the 3 is the *multiplicand* and 4 is the *multiplier*. You can remember this simply as follows: In a multiplication, the first number is the multiplicand and the second number is the multiplier.

We can generalise this to:

1) ADD(multiplier)
2) multiplicand
3) times

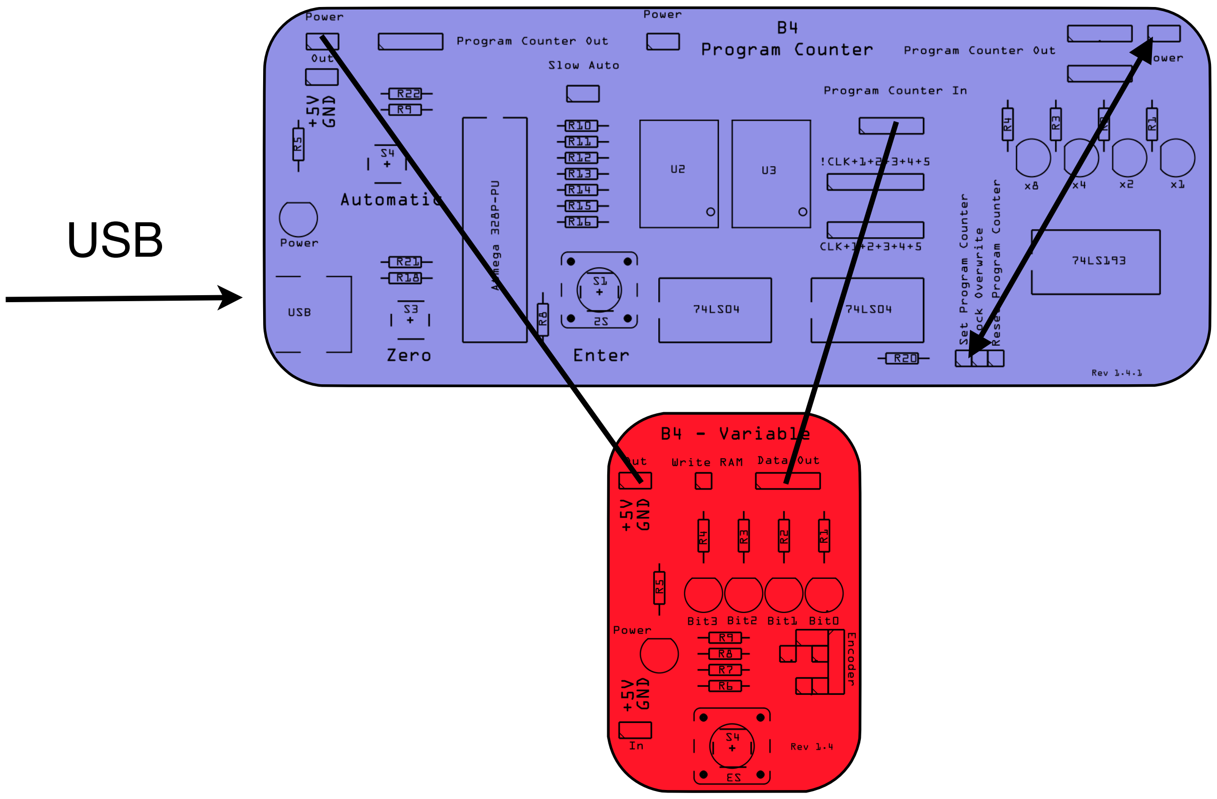
Luckily, our B4’s Adder module can already add. Therefore, we don’t need to worry about point 1). Point 3) refers to a repeated action, meaning some sort of loop. Point 2) refers to a defined number of times that the loop is run, until the algorithm is complete.

In the following chapters, we will explore 2) and 3) further. We start with 3), the loop.

Experiment 2: Loops

Modules Required: Program Counter, 1x Variable, Data RAM, Program RAM

Connect the Variable to the Program Counter as shown. In particular, run a 4 pin wire from the Variable's output to the Program Counter In port. **Connect a 1 pin wire to the Program Counter's Set Program Counter pin and connect the other end of that wire to GND for the moment. We will move this later.**



Setup of Experiment 2, Part 1

1. Turn the knob on the Variable until it shows B1010 (Decimal 10).
2. Press the Enter button on the Program Counter until it displays for example 03.
3. Then, move the end of the 1 pin wire from GND to +5V.

What happens?

The Program Counter will jump to position 10, which is B1010 in binary.

Let's try this again, but with another value.

4. Move the end of the 1 pin wire from +5V to GND
5. Set the Variable to another value, let's say B0110 (Decimal 6)
6. Then, connect the wire to +5V again and the Program Counter will jump to 6.

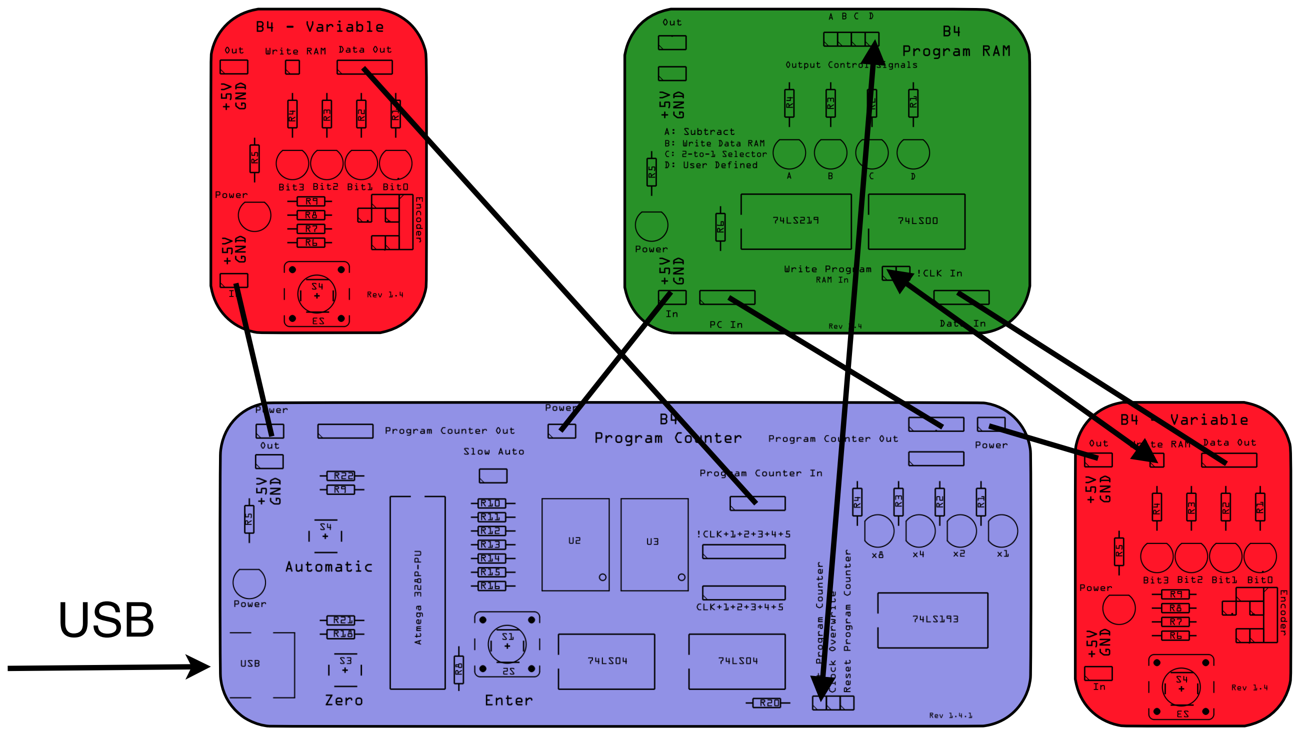
We have just discovered that we can influence the Program Counter so that it will jump to any step in the program that we want it to go to. Of course it would be entirely impractical if we wanted to influence a program by manually setting values on a Variable and then connecting and disconnecting wires. There has to be a better way than that. We want automated jumping, ideally by making it part of the program itself. Luckily, the B4 from the base kit has a spare opcode, which is called User Defined. The plan is to set a flag in the program to indicate that we want to make a jump and store a value in the Data RAM to indicate where we want to jump to.

Let's now run an experiment in which we program the Program RAM to send a signal to the Program Counter to jump to a defined position as set by the Variable. In our experiment, illustrated below, the left Variable plays the role of the Data RAM**. Leave the 1 pin wire from the Program RAM's *port D* to the Program Counter's *Set Program Counter* disconnected for the moment.**

In our experiment, we want to automatically jump from step 7 to step 3 in our program. For this, two things need to happen during step 7:

1) The Program RAM has to contain the value B0001. This sends an electrical impulse to the Set Program Counter Pin of the Program Counter, which will then jump to whatever value is present at the Program Counter In port.
2) Data RAM (in our experiment the left Variable), has to provide the binary value B0011 to the Program Counter.

For this experiment, we only want to experiment with the loop, so the rest of the program is not important. We simply set all other RAM values to 0.



Setup of Experiment 2, Part 2

Set the left Variable to B0011. That's the value we want the Program Counter to jump to. Then, with the right Variable, we will program the Program RAM with the code to make the jump happen. Set the right Variable B0000. Then, set the Program Counter to 0000 as well.

First, we want to clear the Program RAM.

1. Repeat

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

Now that you have cleared the Program RAM you can advance the Program Counter to Step 7 (B0111) and program the value of B0011 into it. This is the address we later want to jump to. Then, press the *Zero* button on the *Program Counter* to go back to Step 0. Our Program RAM should now look like in the table below. In the table, we have left fields of 0's empty to improve the readability. So, basically we want our program to jump to step 3 once we reach step 7, so we will have a step sequence of 0, 1, 2, 3, 4, 5, 6, 7, 3, 4, 5, 6, 7, 3, 4, 5, 6, 7, 3, ...

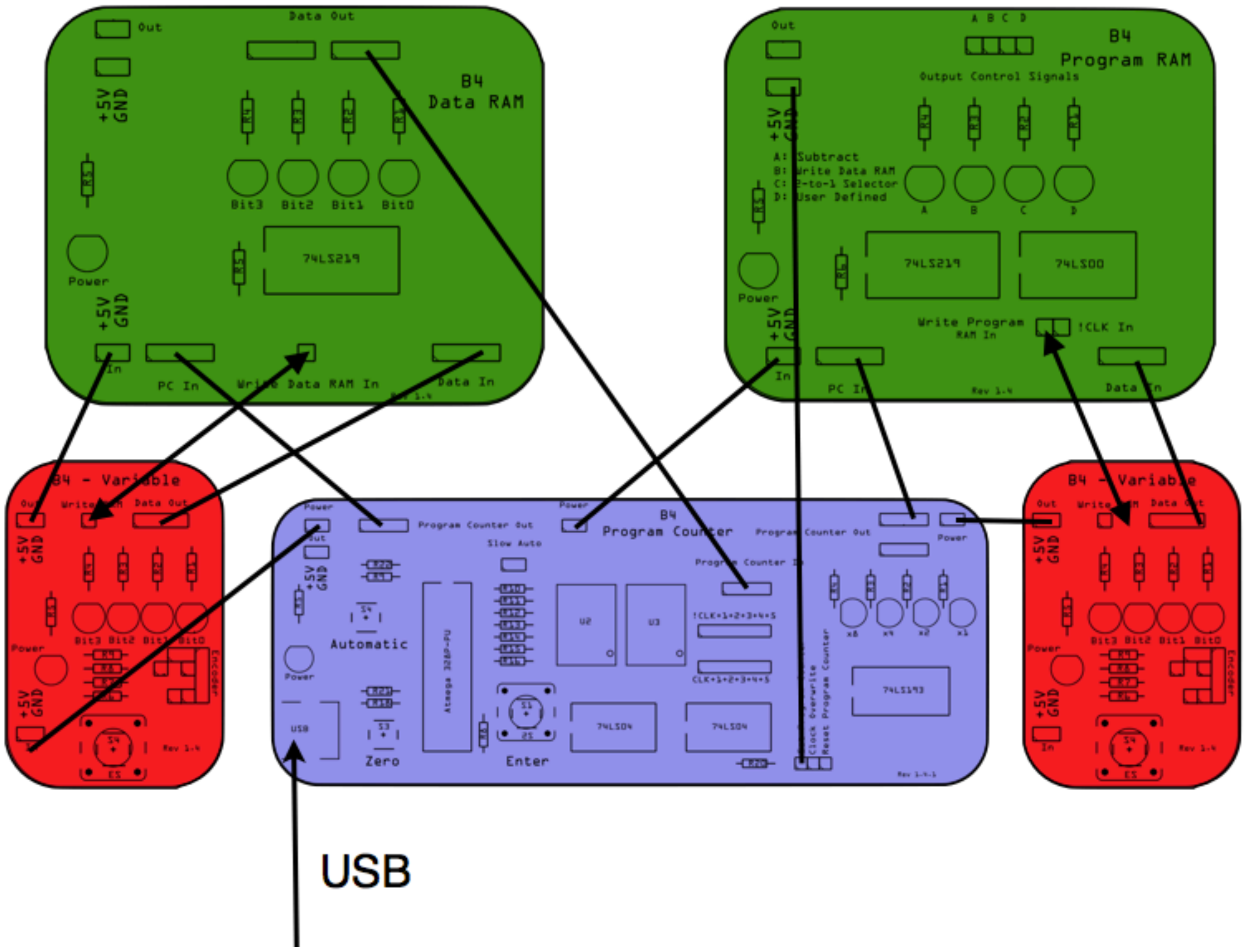
	Data RAM				**Program RAM**				**Description**
Step #	**3**	**2**	**1**	**0**	**SUB**	**WRT**	**SEL**	**USR**	
Step 15									
Step 14									
Step 13									
Step 12									
Step 11									
Step 10									
Step 9									
Step 8									
Step 7	0	0	1	1	0	0	0	1	Jump to step 3
Step 6									
Step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

Program for Experiment 2, Part 2

Now is the time to reconnect the 1 pin wire from the Program RAM's *port D* to the Program Counter's *Set Program Counter, as shown in the figure above.*

Now press the Enter button on the Program Counter until you reach step 6. Then press the button again and the Program Counter will jump straight to step 3. Keep pressing the Enter button and observe how you are now in a loop: Step 3, 4 , 5, 6, 7, 3, ... In fact the

Program Counter does reach step 7. The jumping from 3 to 7 happens too fast for our eyes. How do we know that step 7 is reached? Simply, if step 7 weren't reached, the Jump command would never have been executed.
Still, one practical issue remains: The left Variable acting as Data RAM substitute. Let's replace it with a real Data RAM as shown in the next figure. **Again, leave the 1pin wire from the Program RAM's port D to the Program Counter's Set Program Counter disconnected until you have programmed the RAM modules**.



Setup of Experiment 2, Part 3

As we can see, the left Variable is now only used to program the Data RAM. The Data RAM Module's output is connected to the Program Counter In port of the Program Counter module. Following the procedures that you are already familiar with, program the Data and Program RAM modules with the program from the previous experiment (part 2). Then, press the Zero button on the Program Counter to go back to Step 0. **Finally, reconnect the 1 pin wire from the Program RAM's port D to the Program Counter's Set Program Counter**. Now press the Enter button on the Program Counter until you reach step 6. Then press the button again and the Program Counter will jump straight to step 3. Keep pressing the Enter button and observe how you are now in a loop: Step 3, 4 , 5, 6, 7, 3.

In Summary

In this experiment, we have successfully demonstrated that a program, which is stored in the Program RAM can influence the Program Counter module to jump to a program step as preset in the Data RAM module. We now know how to loop code. This is called an unconditional jump. It is unconditional, because it will always jump. We abbreviate jump as **JMP**. What we need next is a conditional jump that is only executed as long as a condition is true (or false).

Questions

Question 2.1	
?	What was the reason for leaving the 1 pin wire from the Program RAM's port D to the Program Counter's Set Program Counter disconnected during the programming phase?
	What would have happened if we had not disconnected the wire. Explain your thinking and conduct an experiment to verify it.

Experiment 3: Knowing When to Stop

In experiment 1 we found that one of the challenges in multiplication of numbers is to know when to stop. One of the examples we used was 2x5, which we wrote as.

Line	Code
0	LOAD(0);
1	ADD(5). Do this 2 times
2	WRT();

There are different ways of going about keeping track of how many times a loop has been executed. One way of doing this is to count the number of loops. When our program above is executed (runs), then something like this would happen

Command	Result Value	Loop Counter Value
Set Loop Counter Value to 0		0
LOAD(0);	0	0
ADD(5);	5	0
Add 1 to the Loop Counter Value	5	1
Repeat the loop (JMP) if the value of the loop counter is less than 2	5	1
ADD(5);	10	1
Add 1 to the Loop Counter Value	10	2
Repeat the loop (JMP) if the value of the loop counter is less than 2	10	2
WRT();	10	2

This program will perform two ADD(5) operations, which will result in 10. At the end of the program, the Loop Counter Value will be 2.

Now here comes a thought from the engineering perspective: To determine if some number is of a certain value, a piece of hardware (a circuit) is required. And here is the challenge: We would need different circuits for each different multiplicand (**3**x4, or **5**x8, or **7**x7, or **12435768973**x2, etc.). And because there is an infinite number of multiplicands, we would require an infinite number of circuits, which would make our computer infinitely large and therefore infinitely heavy. An infinitely heavy computer would collapse under its own weight and tear a hole in the space time continuum which would result in a black hole which would swallows the Earth, our solar system, and possibly our entire galaxy. This would probably ruin our day ...

Is there a better way which would not destroy planet Earth? Turns out there is. Instead of counting up, we could just as well count down (remember, B4 can subtract). So, we count down the multiplicand and stop jumping when its value is down to zero. With this approach, we only require one circuit, which can determine if a number is zero and remember it. Our new and improved program would then look like something like this in pseudo code.

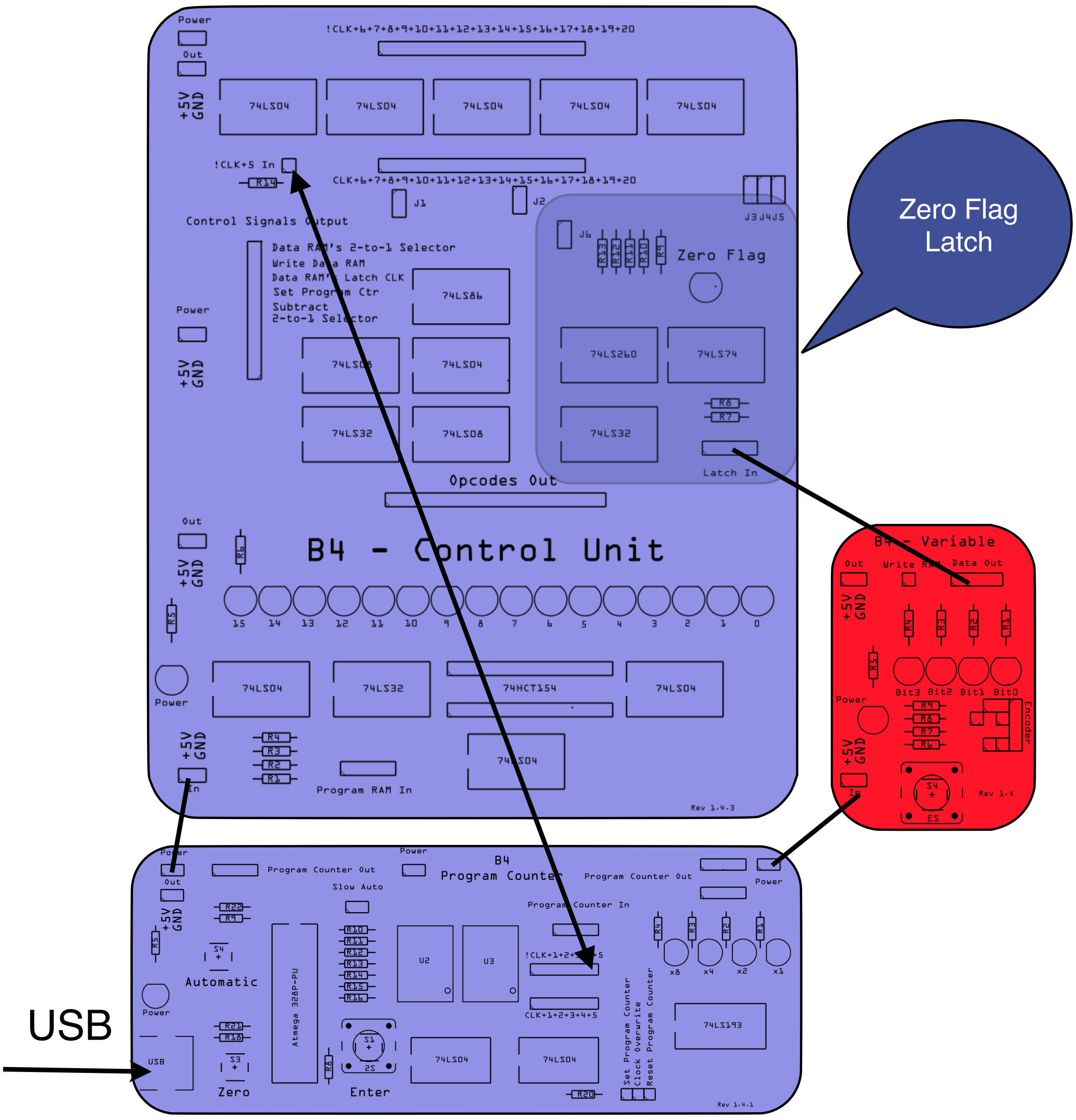
Command	**Result Value**	**Loop Counter Value**
Set Loop Counter Value to 2		2
LOAD(0);	0	2
ADD(5);	5	2
Subtract 1 from the Loop Counter Value	5	1
Repeat the loop (JMP) if the value of the loop counter is not 0	5	1
ADD(5);	10	1
Subtract 1 from the Loop Counter Value	10	0
Repeat the loop (JMP) if the value of the loop counter is not 0	10	0
WRT();	10	0

Ok fantastic, this seems to work just fine. Let’s run an experiment in which we explore the **Zero Flag Latch**.

Modules Required: Program Counter, Control Unit, Variable.

Control Unit Jumper Settings: All Jumpers set.

We connect the Control Unit module to the Program Counter as shown below. We run a 1 pin wire from the !CLK+5 pin of the Program Counter to the corresponding !CLK+5 In pin of the Control Unit module. We connect the output of the Variable to the Adder In port of the Control Unit module. Don't forget the power wires and we are ready to experiment.



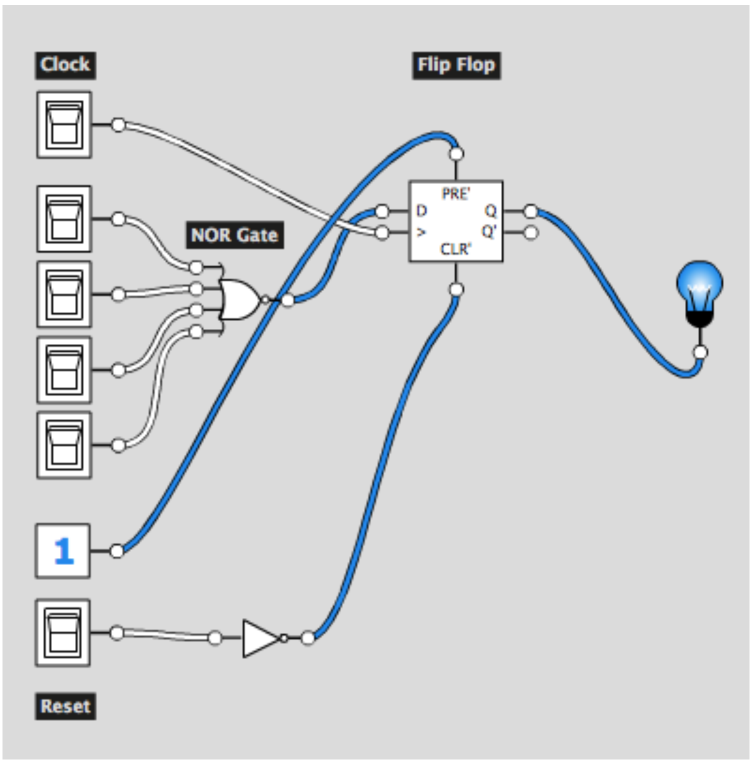
Setup of Experiment 3

Here is how the Zero Flag Latch works. Every time the Enter button is pressed, the clock signal rises to HIGH. At this time, the Zero Latch Flag looks at the input of the Latch In port. If the value is B0000, then the Zero Flag LED will light up. For any value other than

B0000, the LED will remain off. The Zero Flag Latch will remember the result of this analysis until the next clock cycle when the process repeats.

Let’s now set the value of the Variable to B0001 and press the Enter button on the Program Counter. We observe that the Zero Flag LED will remain off. Now change the value of the Variable to B0000. The Zero Flag LED will remain off. Only when we press the Enter button on the Program Counter will the Zero Flag LED turn on. If we then change the value of the Variable to anything else than B000, for example, B1010, the LED will remain on until we press the Enter button. The CLK signal will trigger a new analysis. Try this with a couple more values to get a better feeling how the Zero Flag Latch works.

Internally the Zero Flag Latch consists mainly of an OR gate, a NOT gate and a Flip Flop. As we remember from the B4 base kit handbook, the OR truth table is such that the output is HIGH when one the input signals is HIGH. If we then negate the output, then it will be HIGH when all the inputs of the OR gate are LOW. This is called a NOT-OR or, in short, NOR gate. So a NOR gate will produce a HIGH output when all of its input is LOW. And this is what we want. We then simply feed the output of our NOR gate to our Flip Flop to remember the result. Here is an illustration !



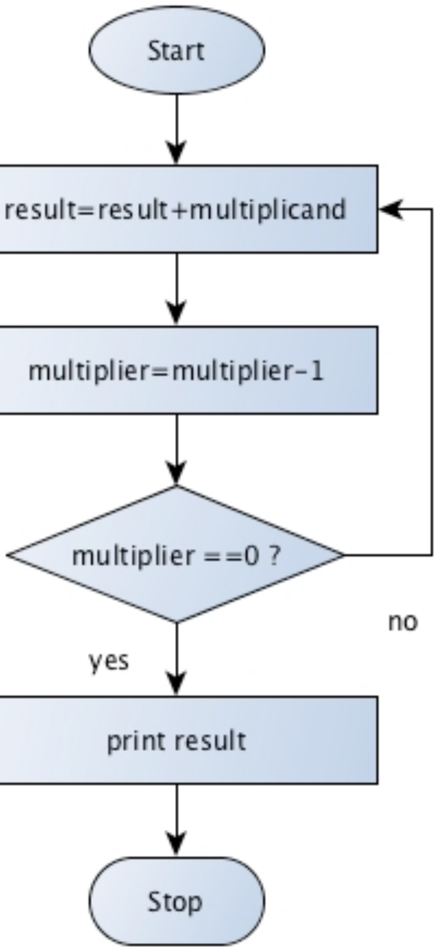
Inside the Zero Flag Latch

The output of the Zero Flag Latch is an electrical signal which can be used to influence the Program Counter in its conditional jumping. We will explore this in the coming chapters, but first we have to upgrade the opcodes of the B4. We’ll do this in the next chapter.

Experiment 4: A Multiplication Program

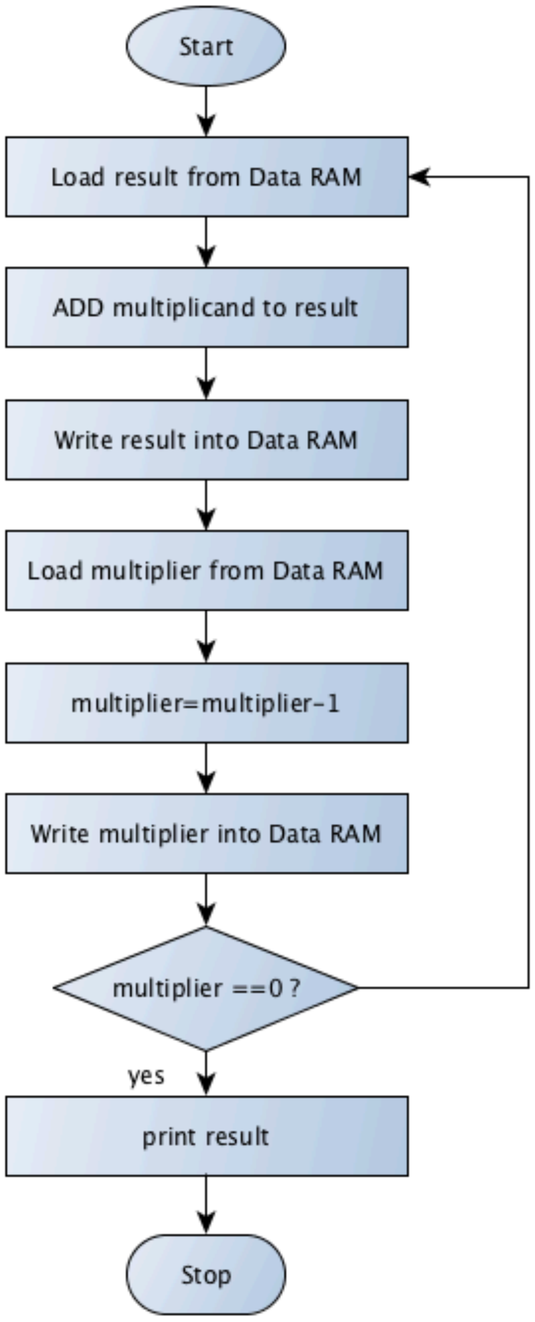
In experiment 1 we explored the challenge of building and programming a machine that is capable of multiplying any two positive natural numbers within the scope of our 4-bit computer architecture, which limits us to small numbers between 0 and 15. We determined that loops were a fundamental challenge, which we explored in experiment 2. In experiment 3 we generalised the loop concept further towards conditional loops, which know when to stop. In this chapter, we want to extend the high-level programs from the previous experiments and design a general purpose multiplication program. As we do this, we revisit the opcodes of our computer, which, from now on we call the B4/A, where A stands for arithmetics. We will soon see why it makes sense to distinguish the B4 and the B4/A.

Here is an example of a general-purpose multiplication program in flowchart representation:



Multiplication Program: High-Level Flowchart

If we look into this program a bit deeper from the perspective of the inner workings of a computer that we are already familiar with, we obtain the following flowchart:


Start
Load result from Data RAM
ADD multiplicand to result
Write result into Data RAM
Load multiplier from Data RAM
multiplier=multiplier-1
Write multiplier into Data RAM
multiplier ==0 ?
yes
print result
Stop

The principle is that we load the result variable from memory, add the multiplicand to it and then write it back into memory.

Then, we load the multiplier, decrement it and also write it back into memory

Then, we check if the multiplier is zero. If not, we repeat the loop. If yes, we print the result.

Multiplication Program: Detailed Flowchart

Let’s run this program, at least on paper for now until we have built the machine that can do this. We have done this in the next table:

	Command	Result Value	Multiplier (Loop Counter) Value
First Iteration	Load result	0	3
	Add multiplicand to result	2	3
	Store the result	2	3
	Load multiplier	2	3
	Subtract 1 from multiplier	2	2
	Store the multiplier	2	2
	Jump to step 0 if multiplier not zero	2	2
Second Iteration	Load result	2	2
	Add multiplicand to result	4	2
	Store the result	4	2
	Load multiplier	4	2
	Subtract 1 from multiplier	4	1
	Store the multiplier	4	1
	Jump to step 0 if multiplier not zero	4	1
Third Iteration	Load result	4	1
	Add multiplicand to result	6	1
	Store the result	6	1
	Load multiplier	6	1
	Subtract 1 from multiplier	6	0
	Store the multiplier	6	0
	Jump to step 0 if multiplier not zero	6	0

Table: Runtime of the 3x2 Program.

Let’s now decompose this into instructions to our machine that we want to build. We begin with result=0, multiplier=3 and multiplicand=2.

	Data RAM				Opcode	Description
Step #	**3**	**2**	**1**	**0**		
Step 15	0	0	0	0		result
Step 14	0	0	1	1		multiplier
Step 13	0	0	1	0		multiplicand
Step 12	0	0	0	1		constant (1)
Step 11						
Step 10						
Step 9						
Step 8						
Step 7						
Step 6	0	0	0	0	JUMP if not zero	Jump to step 0 if multiplier not zero
Step 5	1	1	1	0	WRITE to address	Store the multiplier
Step 4	1	1	0	0	SUB from address	Subtract 1 from multiplier
Step 3	1	1	1	0	LOAD from address	Load multiplier
Step 2	1	1	1	1	WRITE to address	Store the result
Step 1	1	1	0	1	ADD from address	Add multiplicand to result
Step 0	1	1	1	1	LOAD from address	Load result

LOAD from address means that we want the data that is at that particular address. This is like: "Give me the box on the top shelf". So instead of saying *what* we want we say *where* we want it from. So, *LOAD from address B1111* means that we want the data that is stored at address B1111 (decimal 15). That's initially a 0. *ADD from address B1101* correspondingly means that we want to add the data that is stored at address B1101 (decimal 13), which is a 2. In the same way, we can *SUB from address* or *WRITE to address*. And this is extraordinarily handy, because now we can perform many operations on a single piece of data, just by referencing it. This means that our opcode arsenal is growing - which is good - because our computer gets more powerful. Here it is:

Name	Mnemonics	Machine Code	Set 2-to-1 Selector	Activate Inverter	Set 2-to-1 Selector of Data RAM (PC values)	Set Program Counter	Output Latch
Load from address	LOAD_A	1111	1		1		
Add from address	ADD_A	1110			1		
Subtract from address	SUB_A	1101		1	1		
Store at address	WRT_A	1100			1		
Jump to address	JMP	1011				1	
Jump if not zero to address	JNZ	1010				1	
Load (Absolute)	LOAD	1001	1				
Add (Absolute)	ADD	1000					
Subtract (Absolute)	SUB	0111		1			
Print on Decimal Display	PRINT	0110					1
Do Nothing	NOP	0000					

Expanded Opcode ‘Arsenal’

From Absolute Data to Data Pointers

In all our previous programs we have worked with absolute data. LOAD(3), ADD(7), and SUB(2) loaded a three, added a seven or subtracted a two. This was all right for simple addition and subtraction programs. However, absolute data is quite limiting when we want to work with loops where we want to keep count of the number of times a loop has run (iterations) and the computational result the loops is producing. This requires the reading of data in one step, its change in a second step and the storage back into the data RAM in a third step. When we work with absolute data, we can only read and write data into the Data RAM address which corresponds to the value of the Program Counter. So we need a way to read and write data from or to a particular address. And then we access data not by referencing its value, but through its address instead.We explore this in the following experiments.

Experiment 5: Data Pointers

Modules Required: Program Counter, Control Unit, Data RAM, Latch, 2-to-1 Selector, Variable

Control Unit Jumper Settings: All Jumpers set.

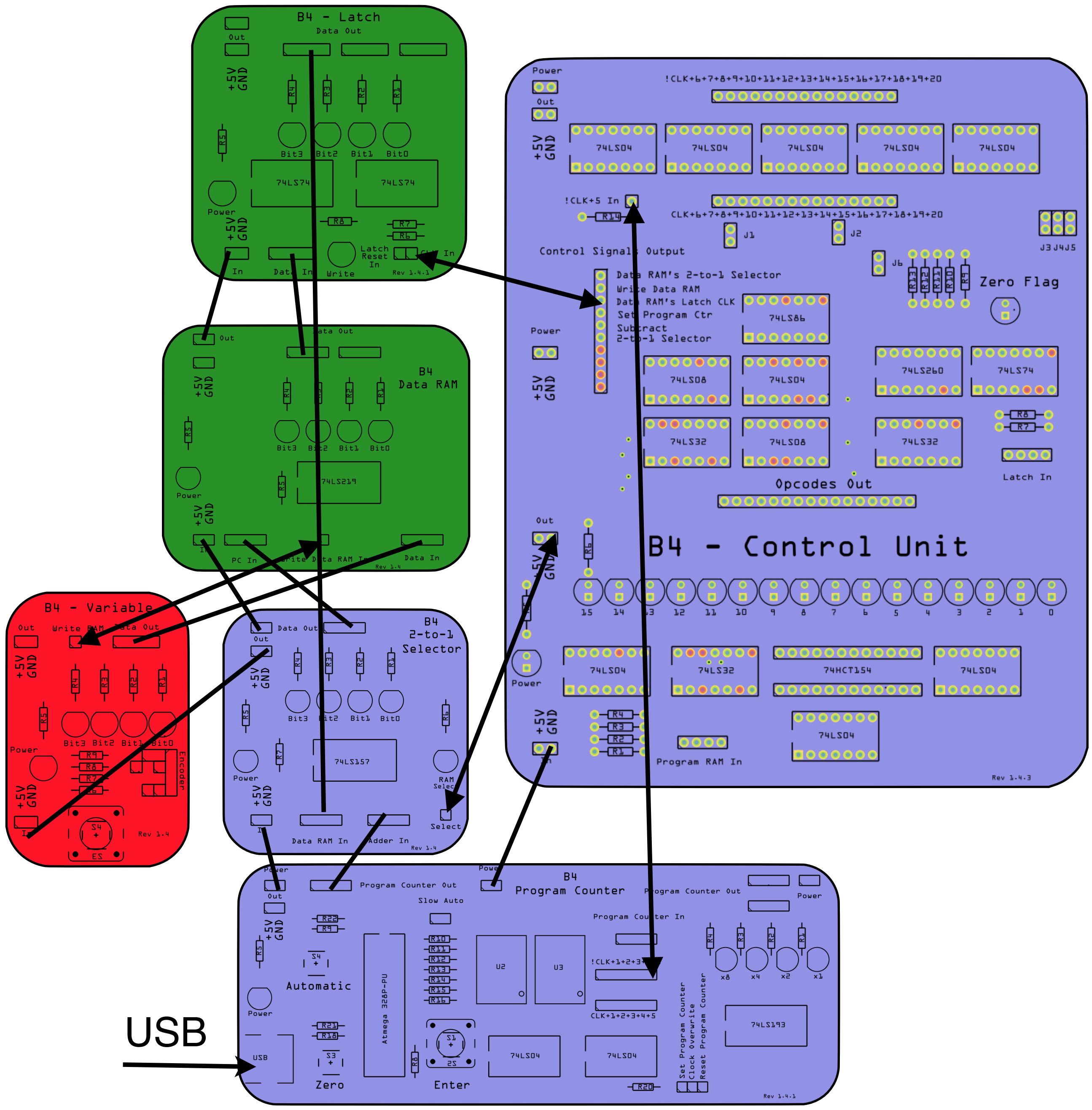
In the previous experiment, we laid the theoretical foundations for data pointers. In this experiment, we want to find out how we actually make them with our hardware. With data pointers we have a bit of a problem that comes in two parts:

1) The data pointer itself is stored in RAM, but the program has to know how to interpret the data - that is, as an address, rather than a data value. So when the Data RAM releases this address needs to be fed back to the Data RAM\'s Program Counter In port (abbreviated as PC In). So, we need to cut the Data RAM loose from the Program Counter and insert a **2-to-1 Selector** between the Program Counter and the Data RAM, so that we can choose at runtime, whether we want a data value or a data pointer. And here is the second part of the problem:
2) When the Data RAM listens to itself, it forms a feedback loop.To solve this, we insert a Latch between the output of the Data RAM and one of the inputs of the 2-to-1 Selector is the solution.

	address Value in Binary				Data RAM Content			
address #	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**
address 15	1	1	1	1	0	0	0	0
address 14	1	1	1	0	0	0	0	1
address 13	1	1	0	1	0	0	1	0
address 12	1	1	0	0	0	0	1	1
address 11	1	0	1	1	0	1	0	0
address 10	1	0	1	0	0	1	0	1
address 9	1	0	0	1	0	1	1	0
address 8	1	0	0	0	0	1	1	1
address 7	0	1	1	1	1	0	0	0
address 6	0	1	1	0	1	0	0	1
address 5	0	1	0	1	1	0	1	0
address 4	0	1	0	0	1	0	1	1
address 3	0	0	1	1	1	1	0	0
address 2	0	0	1	0	1	1	0	1

	address Value in Binary				Data RAM Content			
address #	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**
address 1	0	0	0	1	1	1	1	0
address 0	0	0	0	0	1	1	1	1

All this is shown in the following figure.



Setup of Experiment 5

Connect the 2-to-1 Selector's Select pin to GND. This way, the Data RAM will listen to the Program Counter.

Next, program the Data RAM with the values from the table above. You would know how to do this.

When you have done this, click the Enter button on the Program Counter repeatedly. You should see the normal behaviour, that is, the Data RAM will show the value at the address from the Program Counter. Did you notice that the Latch will always show the same output as the Data RAM? We say that the Latch shadows the Data RAM.

Before you continue, Set the Program counter to step 1.

If you now move the Selector control wire from GND to +5V, what will happen? The 2-to-1 Selector also displays B1110 on its output, which makes sense, because it now listens to the Latch. The 2-to-1 Selector output is now the address to which the Data RAM listens, and the Data RAM consequently displays the value, which is stored at address B1110, which is B0001. Note that the output of the Latch does not change when you change the Selector wire from GND to +5V. The Latch is breaking the circuit. It will only update its value when it receives a signal from the Control Unit signal, and this signal is tightly coupled to the Program Counter's CLK signal which can only be generated when we press the Enter button on the Program Counter module.

Move the selector wire to GND again, advance the Program Counter to address 2. The Data RAM will display the value stored at this address, which is B1101. If you now connect the **Selector control wire from GND to +5V**, then the 2-to-1 Selector also displays B1101 on its output. The 2-to-1 Selector output (coming straight from the Latch) is now the address to which the Data RAM listens, and the Data RAM consequently displays the value which is stored at address B1101, which is B0010.

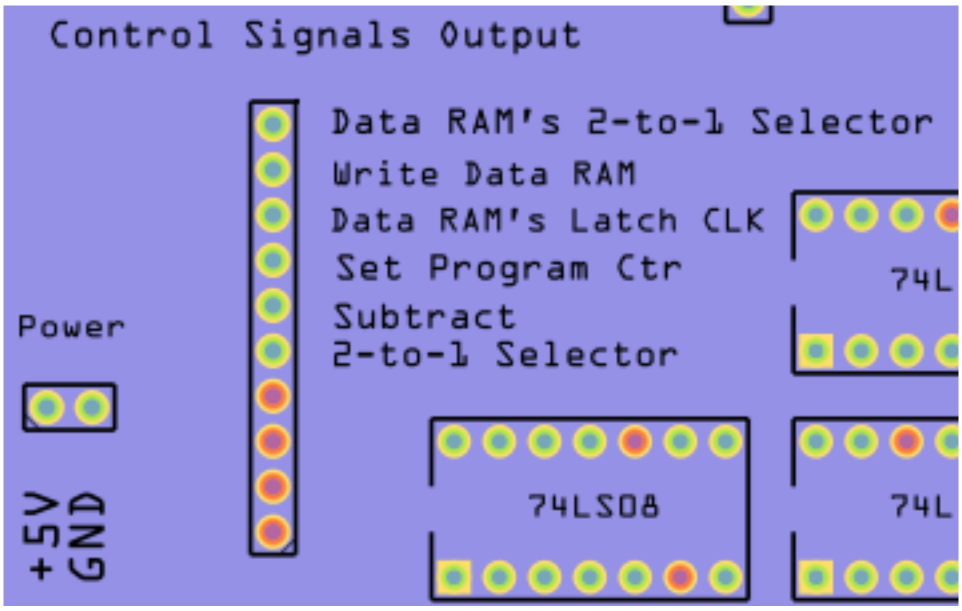
By the way, did you notice that the Latch is a bit laggy? It is just a little bit behind the Data RAM when it changes its output. You have to pay close attention to notice it. The reason for this is, that the Latch CLK is delayed by a very long chain of delay circuits. We have done this to ensure that the Latch does not latch onto the Data RAM output before the Data RAM has done everything it needs to do to show the data value stored at the new address. Clever timing, isn't?

Experiment 6: Automating Data Pointers

Modules Required: Same modules as in experiment 6, plus Variable

Control Unit Jumper Settings: All Jumpers set.

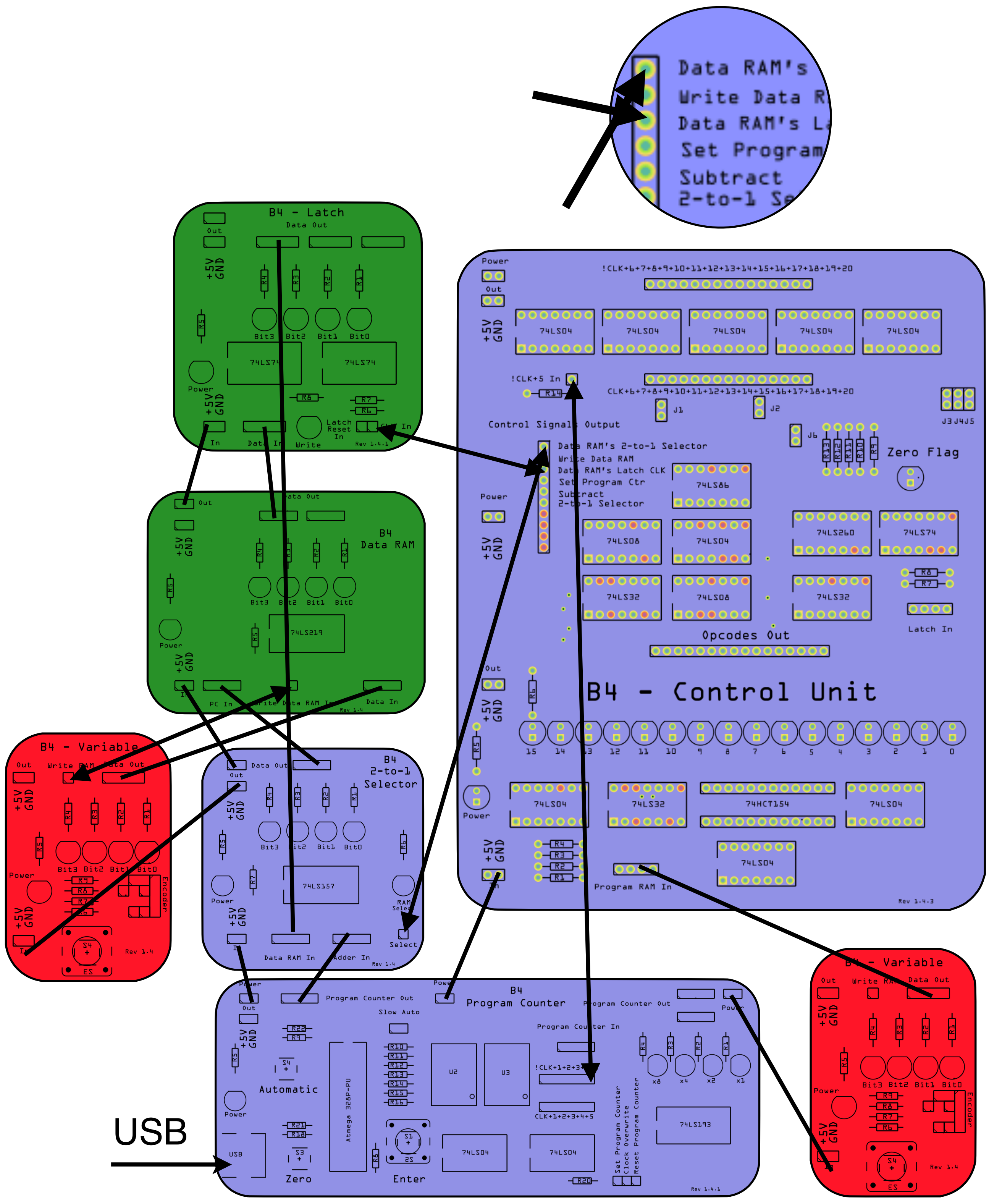
Moving the control wire around to enable data pointers is not the way computers work in practice. So in this experiment, we are using the Control Unit to send an activation signal to the 2-to-1 selector instead. If you take a look at the Control Signal Output section on the Control Unit, you will see that the pin at the very top is labelled as **Data RAM's 2-to-1 Selector**.



Control Signal Output Pins on the Control Unit

Connect the Select control wire from the 2-to-1 Selector to this pin as shown in the following figure. Whilst you are doing this, also wire up a second Variable as shown. We'll use it to generate some opcodes for this experiment.

If you kept the B4/A powered from the previous experiment then you already have the Data RAM module filled with useful data. Otherwise, just program the Data RAM again as shown in experiment 5.



Setup of Experiment 6

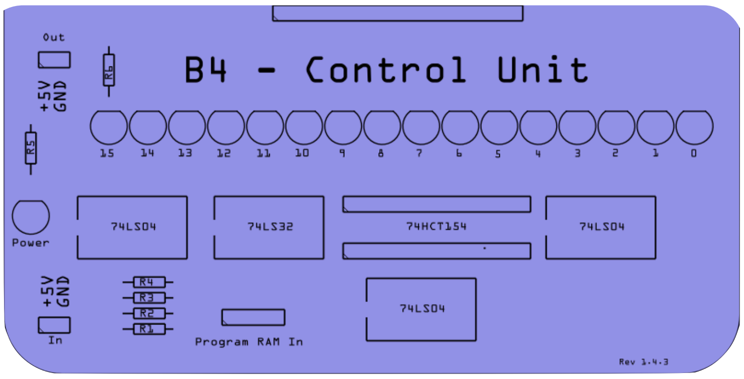
In experiment 4, we talked about the extended opcode ‘arsenal’ of the B4/A. Here is the opcode table again:

Name	Mnemonics	Machine Code	Set 2-to-1 Selector	Activate Inverter	Set 2-to-1 Selector of Data RAM (PC values)	Set Program Counter	Output Latch
Load from address	LOAD_A	1111	1		1		
Add from address	ADD_A	1110			1		
Subtract from address	SUB_A	1101		1	1		
Store at address	WRT_A	1100			1		
Jump to address	JMP	1011				1	
Jump if not zero to address	JNZ	1010				1	
Load (Absolute)	LOAD	1001	1				
Add (Absolute)	ADD	1000					
Subtract (Absolute)	SUB	0111		1			
Print on Decimal Display	PRINT	0110					1
Do Nothing	NOP	0000					

Expanded Opcode 'Arsenal'

Have a look at the highlighted region in the table. It says that the Data RAM's 2-to-1 Selector is activated for the instructions Load from address, Add from address, Subtract from address and Store at address. These instructions represent the Machine Codes B1111, B1110, B1101 and B1100. In decimal, that's the instructions 15, 14, 13 and 12.

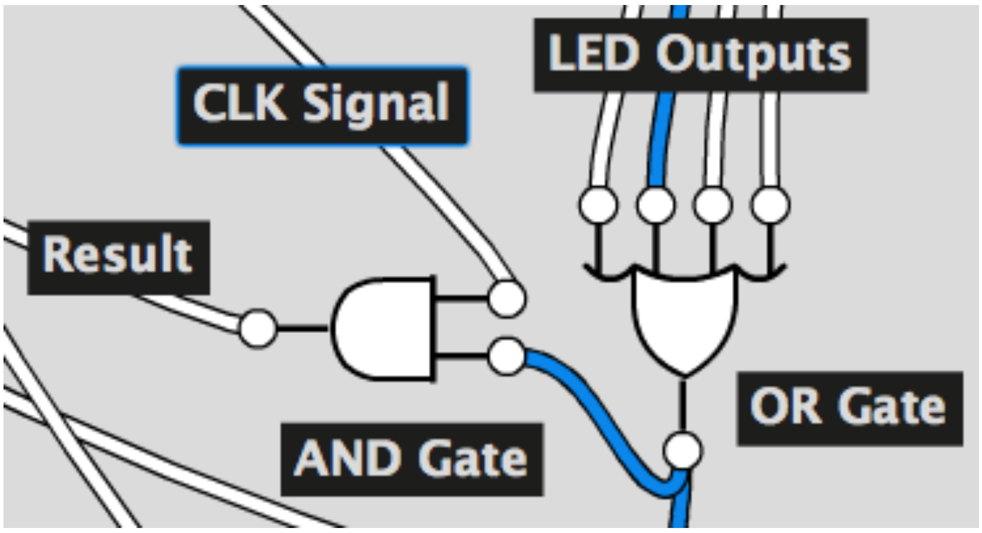
Now, have a look at the Control Unit's long horizontal row of LEDs:



Expanded Opcode 'Arsenal'

These LEDs represent the Machine codes which we feed into the Program RAM In Port - so they will ultimately come from the Program RAM - we'll get to this a little later. So when the machine code is B1000, then the LED 8 will lite up. For machine code B1111 LED 15 will lite up. Try this out by turning the knob of the second Variable you just added to your setup.

Inside the Control Unit, we feed these values from the LEDs to some other circuity, but in the case of the Data RAM's 2-to-1 Selector, this is one OR and one AND Gate. The logical expression is: If ((LED 15==1) OR (LED 14==1) OR (LED 13==1) OR (LED 12==1)) AND (CLK==1) THEN activate the Data RAM's 2-to-1 Selector.



Inside the Control Unit: The Main Logic Gates that Activate the Data RAM's 2-to-1 Selector

Program your data RAM with the values from the following table.

	address Value in Binary				Data RAM Content			
address #	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**
address 15	1	1	1	1	0	1	1	1
address 14	1	1	1	0	0	1	0	1
address 13	1	1	0	1	0	0	1	1
address 12	1	1	0	0	0	0	0	1
address 11	1	0	1	1	0	0	0	0
address 10	1	0	1	0	0	0	0	0
address 9	1	0	0	1	0	0	0	0
address 8	1	0	0	0	0	0	0	0
address 7	0	1	1	1	0	0	0	0
address 6	0	1	1	0	0	0	0	0
address 5	0	1	0	1	0	0	0	0
address 4	0	1	0	0	1	1	1	1
address 3	0	0	1	1	1	1	1	0
address 2	0	0	1	0	1	1	0	1

	address Value in Binary				Data RAM Content			
address #	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**
address 1	0	0	0	1	1	1	0	0
address 0	0	0	0	0	1	1	1	1

Run the experiment:

1. Reset the Program Counter to zero
2. Set the variable to a value of 12 or greater
3. Press the enter button on the Program Counter

The following steps happen very quickly in this order.

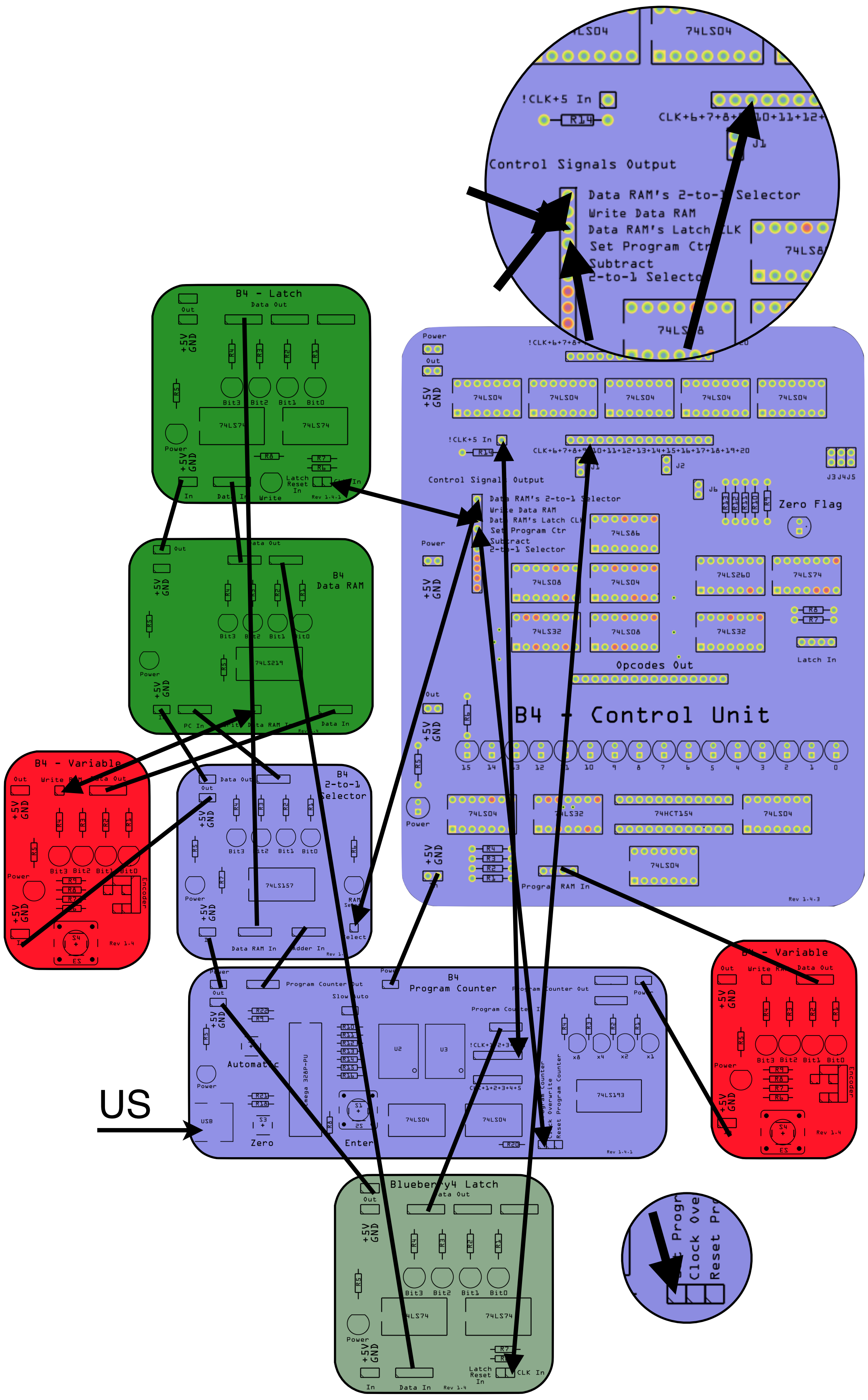
The CLK signal from the Program Counter to the Control Unit flips
The value 1 is sent from the Program Counter to the 2-to-1 Selector, which passes it on to the Data RAM
The Data RAM looks up the value at address 1, which is 13 and presents it at its output
The Latch latches 13 and provides it to the 2-to-1 Selector
The Control Unit sends an activation signal to the 2-to-1 Selector which makes the Selector listen to the Latch.
The 2-to-1 Selector provides the value 13 to the Data RAM
The Data RAM looks up the value at address 13, which is 3 and presents it at its output
The CLK signal from the Program Counter to the Control Unit flips back
The Latch latches and remembers the value 3 from the Data RAM.

This clever little workflow uses the high and low part of the clock cycle quite effectively. It first remembers an address and then goes and gets the value from this address. We will see in later experiments that the value can be used for further computation, for example for addition or subtraction.

Repeat the experiment
Repeat the experiment with the right variable set to a value less than 12. Observe and note the exact steps and compare them with the previous observations. How many times will the Latch latch ? With the variable set to a value less than 12, the Latch will only latch once. That means it will treat data as a value, and not as an address. Hence, there is no need to go and fetch the data from an address in memory.

We have just learned how a computer can fully automatically read data from memory.

If your experiment doesn't work as described, check if you have wired the experiment correctly. Most likely, you have connected the Control Unit's !CLK+5 input not to !CLK+5 on the Program Counter, but to CLK+5 (without the exclamation mark), which has inverted the CLK signal. That happened to us when we designed the experiment - silly mistake :-)



Experiment 7: Automating Loops

In experiment 2 we learned how to do jumps by influencing the Program Counter. We can use the knowledge from experiment 5 to add another Latch to our B4/A and wire it up with the Control Unit so that we can do jumps without changing wires at runtime. The principle is the same as in experiment 5: A Latch is inserted between the output of the Data RAM and the Program Counter's In port. The Latch then acts as a circuit breaker.

For this experiment, we extend the setup from experiment 6 with an additional Latch that we place below the Program Counter as shown.

We then connect its CLK In Pin to CLK+8 and finally wire up the Program Counter's Set Program Counter pin to the Set Program Ctr[1] pin of the Control Unit. We then connect the Latch with power and finally connect the output of the Latch with the Program Counter's In port. From now on we refer to this latch as the **Program Counter Latch** or just the **Jump-Latch**.The Jump-Latch remembers every output of the Data RAM.

Program the Data RAM with the program from Experiment 5.

Let's look at our opcodes again. You can see that the Set Program Counter signal is sent for the jump instructions JMP (that is the unconditional jump) and JNZ (the conditional jump). These correspond to machine codes B1011 and B1010. We have highlighted them in the table below.

Name	Mnemonics	Machine Code	Set 2-to-1 Selector	Activate Inverter	Set 2-to-1 Selector of Data RAM (PC values)	Set Program Counter	Output Latch
Load from address	LOAD_A	1111	1		1		
Add from address	ADD_A	1110			1		
Subtract from address	SUB_A	1101		1	1		
Store at address	WRT_A	1100			1		
Jump to address	JMP	1011				1	
Jump if not zero to address	JNZ	1010				1	
Load (Absolute)	LOAD	1001	1				
Add (Absolute)	ADD	1000					
Subtract (Absolute)	SUB	0111		1			
Print on Decimal Display	PRINT	0110					1
Do Nothing	NOP	0000					

Setup of Experiment 7

[1] Ctr=Counter

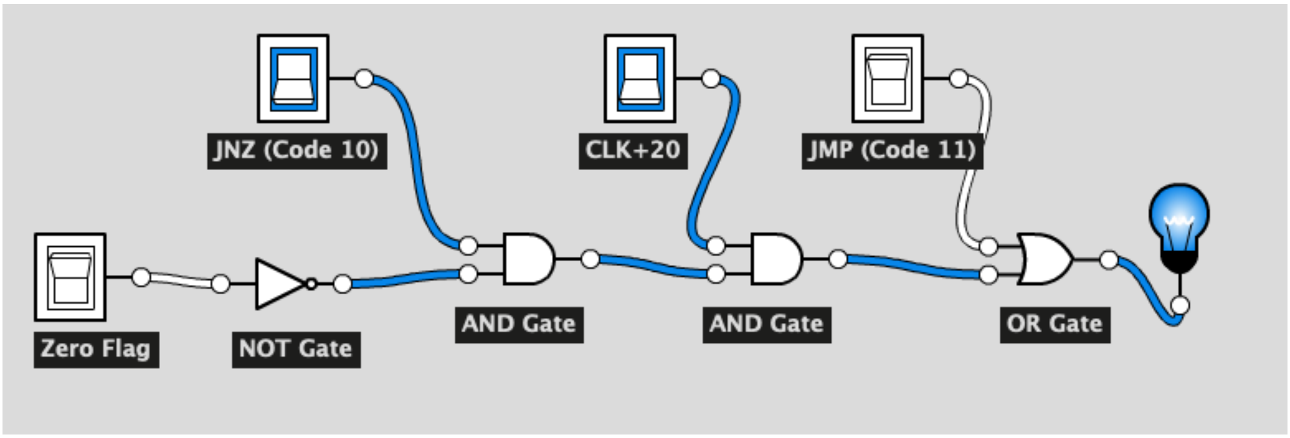
Experiment:

1. Set the right Variable to B0000
2. Press the Enter button on the Program Counter repeatedly until you get to position 6 (B0110). The Data RAM's output should be B1001 and the same value should be in the Jump-Latch.
3. Turn the knob on the right Variable **up** to B1010 (that's the opcode for JNZ)
4. The Program Counter will jump to B1001 (decimal 9).
5. The Data RAM will display the value stored at position decimal 9, which is a B0110 (decimal 6)

You can try the experiment with the JMP opcode, but you will need to come **down**, rather than up with the right Variable, starting at opcode B1111. This experiment looks as follows:

1. Set the right Variable to **B1111**
2. Press the Enter button on the Program Counter repeatedly until you get to position 6 (B0110). The Data RAM's output should be B1001 and the same value should be in the Jump-Latch.
3. Turn the knob on the right Variable **down** to B1010 (that's the opcode for JMP)
4. The Program Counter will jump to B1001 (decimal 9).
5. The Data RAM will display the value stored at position decimal 9, which is a B0110 (decimal 6)

So in both cases we can prompt the Program Counter to jump to a new position. The difference between the unconditional jump JMP and its conditional sister JNZ is how the Set Program Ctr control signal is being generated. JMP will fire always when the machine code is B1011 is present, whilst JNZ requires the machine code B1010 **AND** a passive Zero Flag **AND** a CLK signal. Below you see the corresponding logic diagram. Have a go and trace it.



The entire chain to generate the Set Program Counter control signal is encoded in hardware with logic circuits inside the B4 Control Unit.

We have now reached the point at which we have learned everything that is in addition to the B4 from the base kit. We are now ready to build a computer capable of multiplication and division. Let's get started. Turn over the page and start with experiment 8.

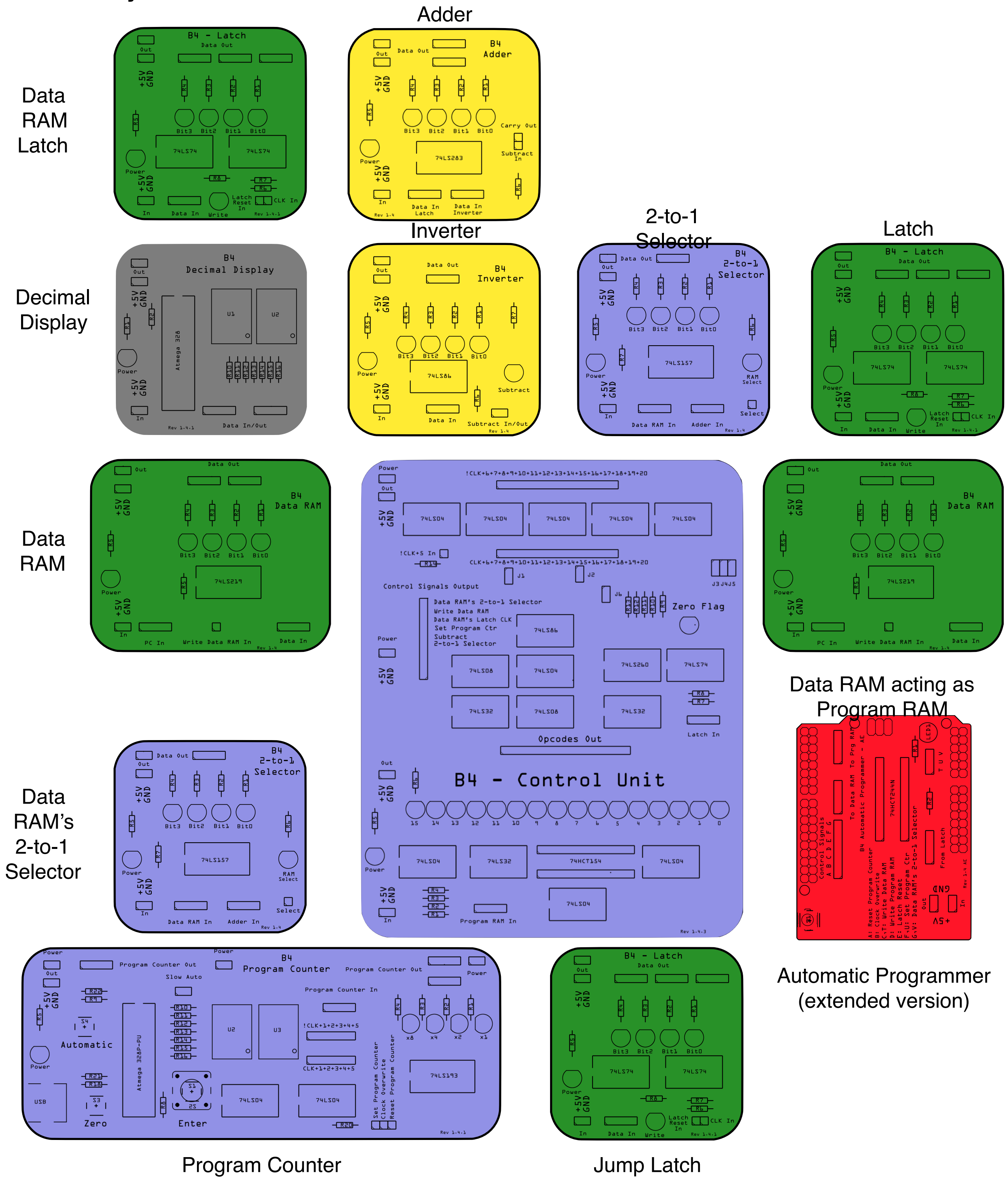
Experiment 8: We are Building the B4/A

This will be our biggest build and we build on top of experiment 7. However, because of the complexity of this experiment, we will do this step by step. We start by placing the modules in front of us. We then connect the power wires. After that we connect the data wires and finally the control wires. In Experiment 9, we will then write software for our computer.

Contrary to the way we built the B4 before, we will move straight to the Automatic Programmer and not use the Variables. Make sure you use the Automatic Programmer's extended version that came with this extension kit.

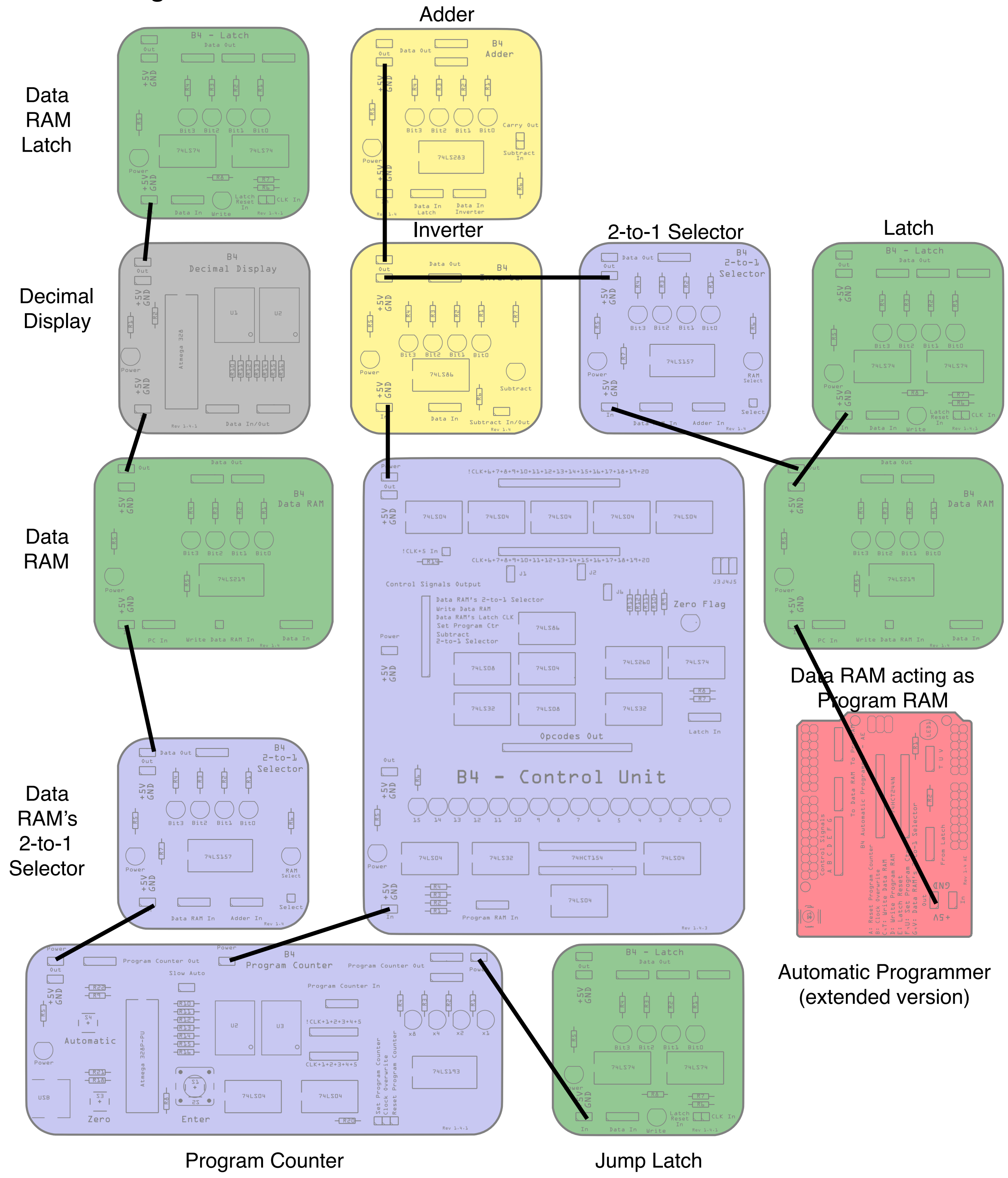
Also, we insert the Decimal Display into the setup so that we can more easily trace what's going on inside the Data RAM.

Module Layout



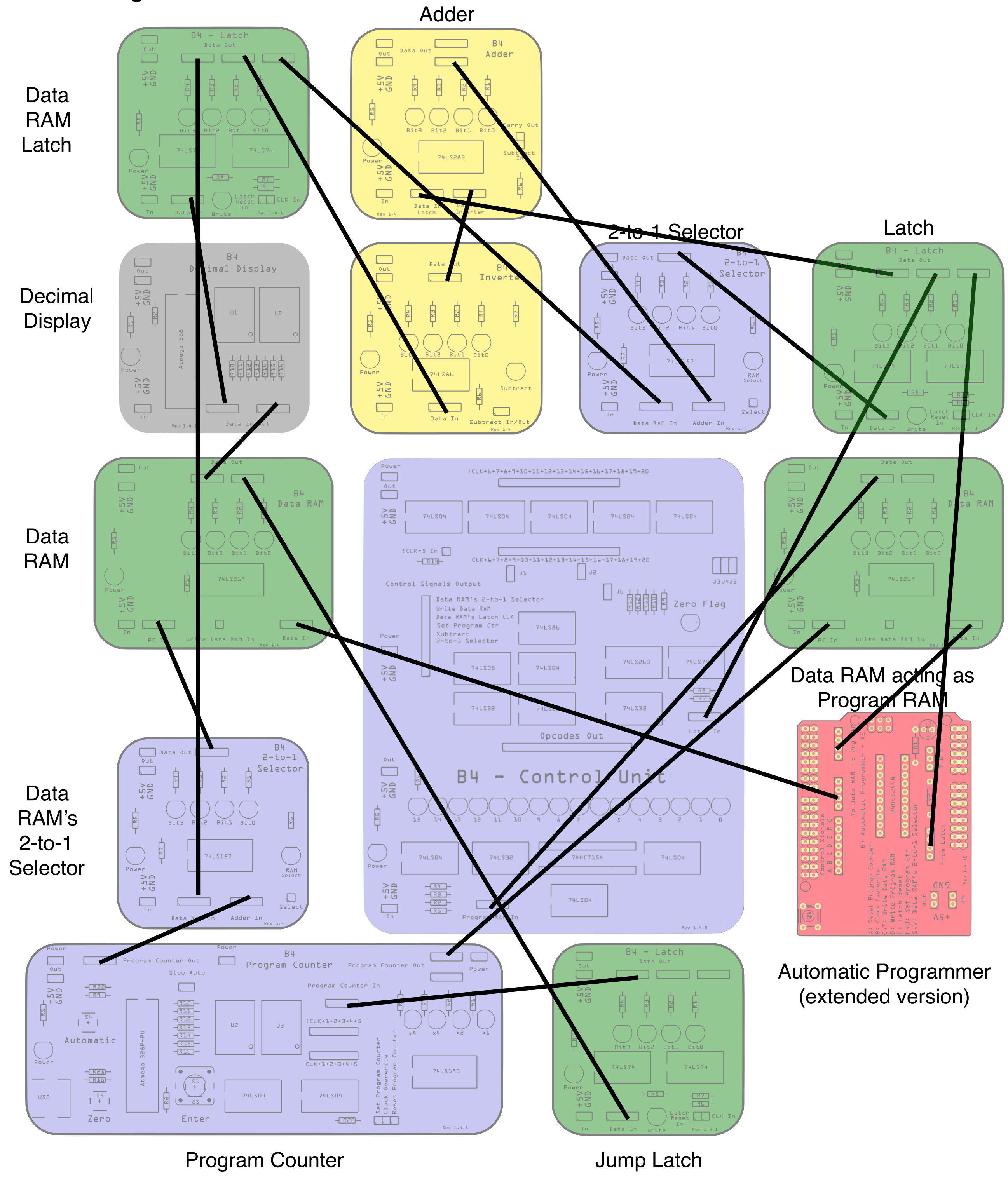
Module Layout for Experiment 8

Power Wiring



Setup of Experiment 8: Power Wiring only

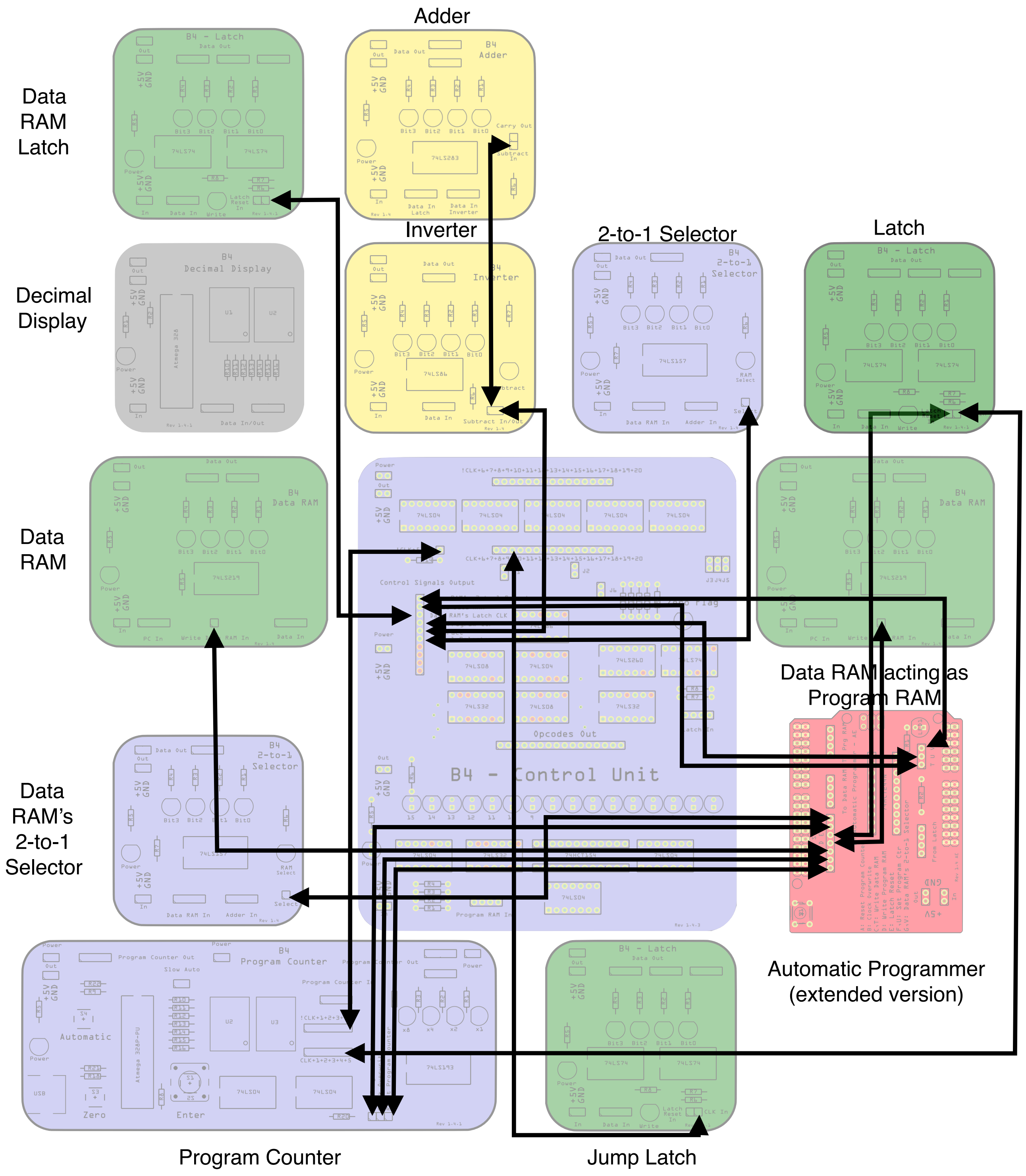
Data Wiring



Setup of Experiment 8: Data Wiring only

Control Wiring

In addition to the wiring shown below, apply the Control Unit wiring. **Please go to https://www.digital-technologies.institute/handbooks to download the Control Unit Calibration document, which contains the exact jumper settings for your Control Unit.**

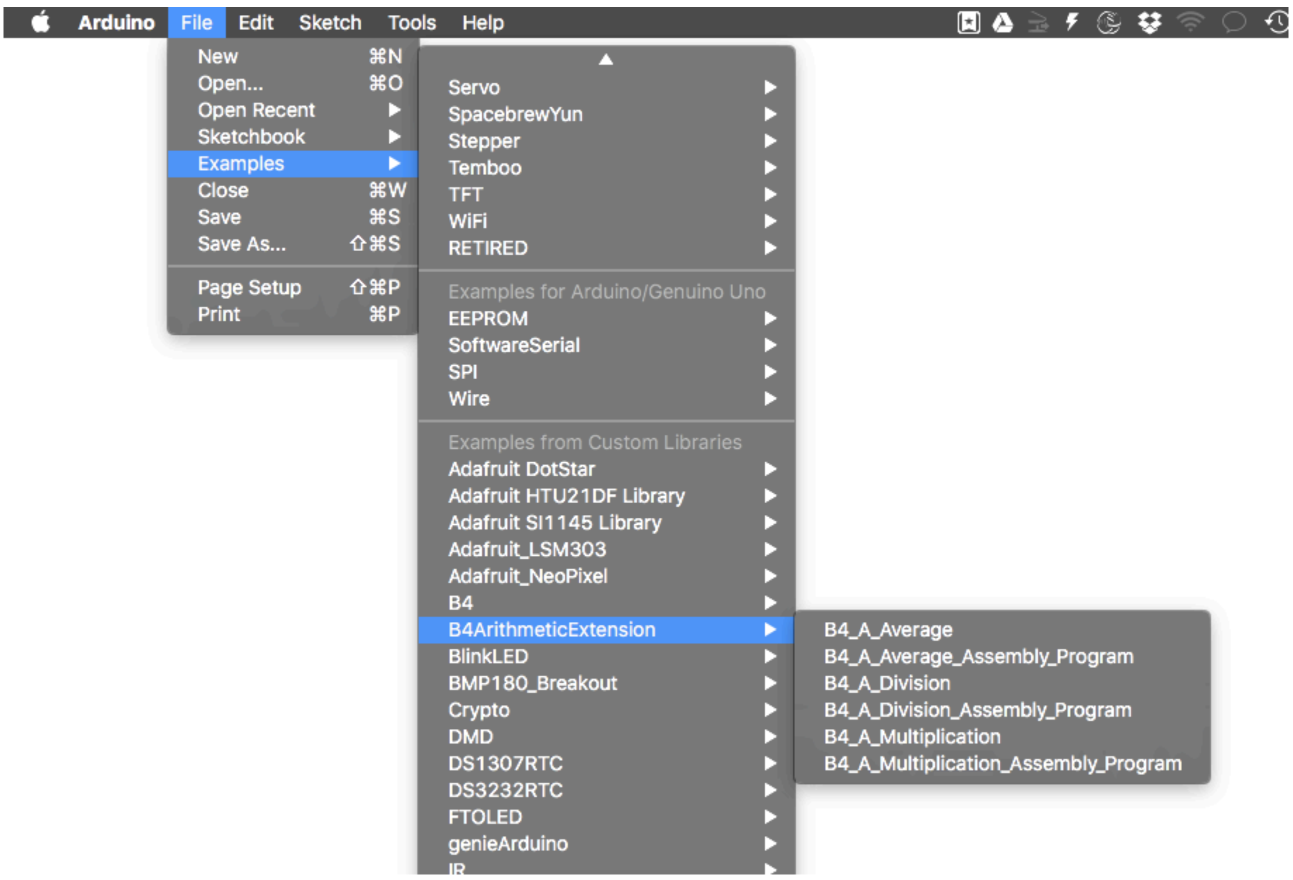


Setup of Experiment 8: Control Wiring only

When you are done, your B4/A's hardware is complete.Plug in a USB cable into the Automatic Programmer and connect the other end to a computer. Check that the green LEDs on all boards (except on the Automatic Programmer) are lit. If they are not, check the power wiring.

Experiment 9: Programming the B4/A

To Program the B4/A we first need to download and install the **B4ArithmeticExtension** library from http://www.digital-technologies.institute/downloads into the Libraries folder in which your Arduino Sketches reside. On Windows and Macintosh machines, the default name of the folder is "Arduino/libraries" and is located in your Documents folder. Then, re-start the Arduino IDE and go into the File menu. There, select Examples, and click on B4ArithmeticExtension. This will look something like in the following figure:



B4/A Library inside the Arduino IDE

The Library already contains a number of programs. For each program, you find two representations: One with the binary machine code arrays and a corresponding assembly program. Both are totally equivalent. You can choose with which representation you want to work with. In this chapter we use both representations: machine code and assembly language.

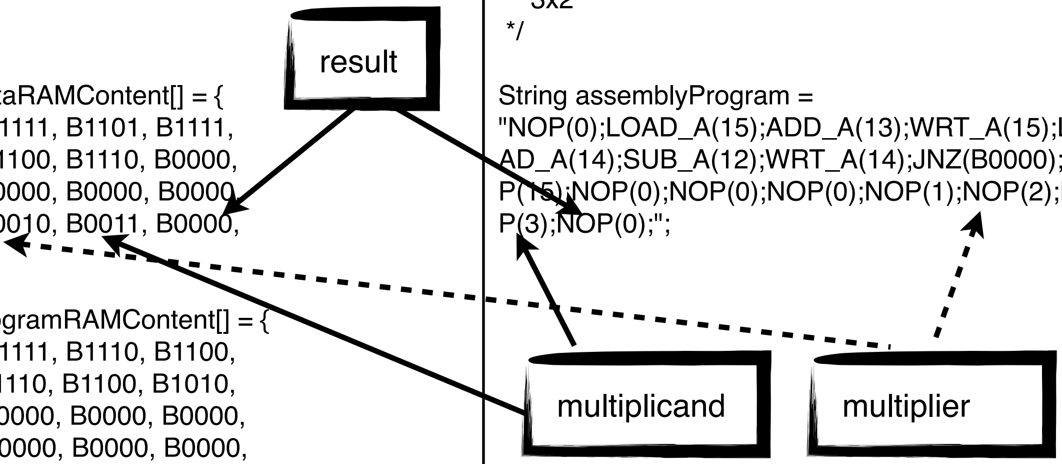
Computing Multiplication

Let's recap multiplication:

result=multiplicand x multiplier = series of additions of: multiplier+multiplier + ...

Machine Code Representation	Assembly Code Representation

#include <B4ArithmeticExtension.h>
 	
B4 myB4;
/*
 * 3x2
 */
 	
  uint8_t DataRAMContent[] = {
    B0000, B1111, B1101, B1111,
    B1110, B1100, B1110, B0000,
    B1111, B0000, B0000, B0000,
    B1111, B0010, B0011, B0000,
  };
 	
  uint8_t ProgramRAMContent[] = {
    B0000, B1111, B1110, B1100,
    B1111, B1110, B1100, B1010,
    B1011, B0000, B0000, B0000,
    B0000, B0000, B0000, B0000,
  };
 	
void setup()
{
  myB4.loadDataAndProgram(DataRAMContent, ProgramRAMContent);
  myB4.programB4();
}
 	
void loop()
{
}
	#include <B4ArithmeticExtension.h>
	
	B4 myB4;
	/*
	 * 3x2
	 */
	
	String assemblyProgram = "NOP(0);LOAD_A(15);ADD_A(13);WRT_A(15);LOAD_A(14);SUB_A(12);WRT_A(14);JNZ(B0000);JMP(15);NOP(0);NOP(0);NOP(0);NOP(1);NOP(2);NOP(3);NOP(0);";
	
	void setup()
	{
	  Serial.begin(9600);
	  myB4.assembler(&assemblyProgram);
	  myB4.programB4();
	}
	
	void loop()
	{
	}



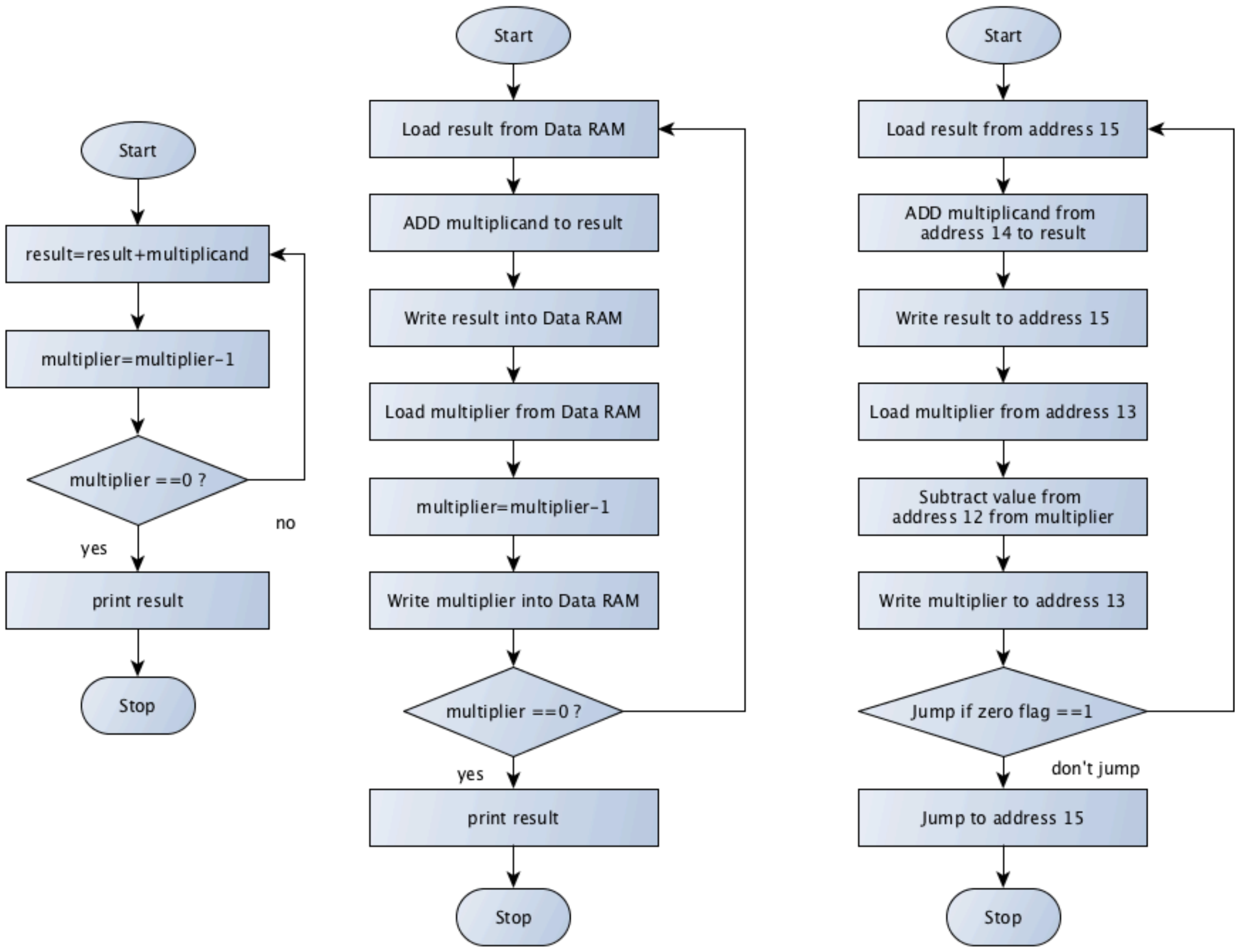
Arduino Multiplication Program to Compute 3 x 2

In the code above we have highlighted the parts that are in charge of the result, the multiplicand and the multiplier. In the following table, you can see the B4/A part of that program in our familiar table representation.

	Data RAM				Opcode	Description
address/Step #	**3**	**2**	**1**	**0**		
15	0	0	0	0	NOP	result
14	0	0	1	1	NOP	multiplier
13	0	0	1	0	NOP	multiplicand
12	0	0	0	1	NOP	constant
11					NOP	not used
10					NOP	not used
9					NOP	not used
8	1	1	1	1	JUMP	Jump to address 15 to display the result
7	0	0	0	0	JUMP if not zero	Jump to step 0 if multiplier not zero
6	1	1	1	0	WRITE to address	Store the multiplier at address 14
5	1	1	0	0	SUB from address	Subtract 1 (which is at address 12) from multiplier
4	1	1	1	0	LOAD from address	Load multiplier from address 14
3	1	1	1	1	WRITE to address	Store the result at address 15
2	1	1	0	1	ADD from address	Add multiplicand to result
1	1	1	1	1	LOAD from address	Load result from address 15
0	0	0	0	0	NOP	Landing pad for the JNZ instruction at step 7

3 x 2 Multiplication Program in Table Representation

We can also look at this program as a flowchart, which you can see in the next diagram. You have already seen the flowcharts on the left and in the middle in previous chapters. The flowchart on the right is more detailed and contains information about exactly which memory addresses the program needs to work with.



Start
result=result+multiplicand
multiplier=multiplier-1
multiplier ==0 ?
no
yes
print result
Stop
Start
Load result from Data RAM
ADD multiplicand to result
Write result into Data RAM
Load multiplier from Data RAM
multiplier=multiplier-1
Write multiplier into Data RAM
multiplier ==0 ?
yes
print result
Stop
Start
Load result from address 15
ADD multiplicand from address 14 to result
Write result to address 15
Load multiplier from address 13
Subtract value from address 12 from multiplier
Write multiplier to address 13
Jump if zero flag ==1
don't jump
Jump to address 15
Stop

3 x 2 Multiplication Program in Flowchart Representation

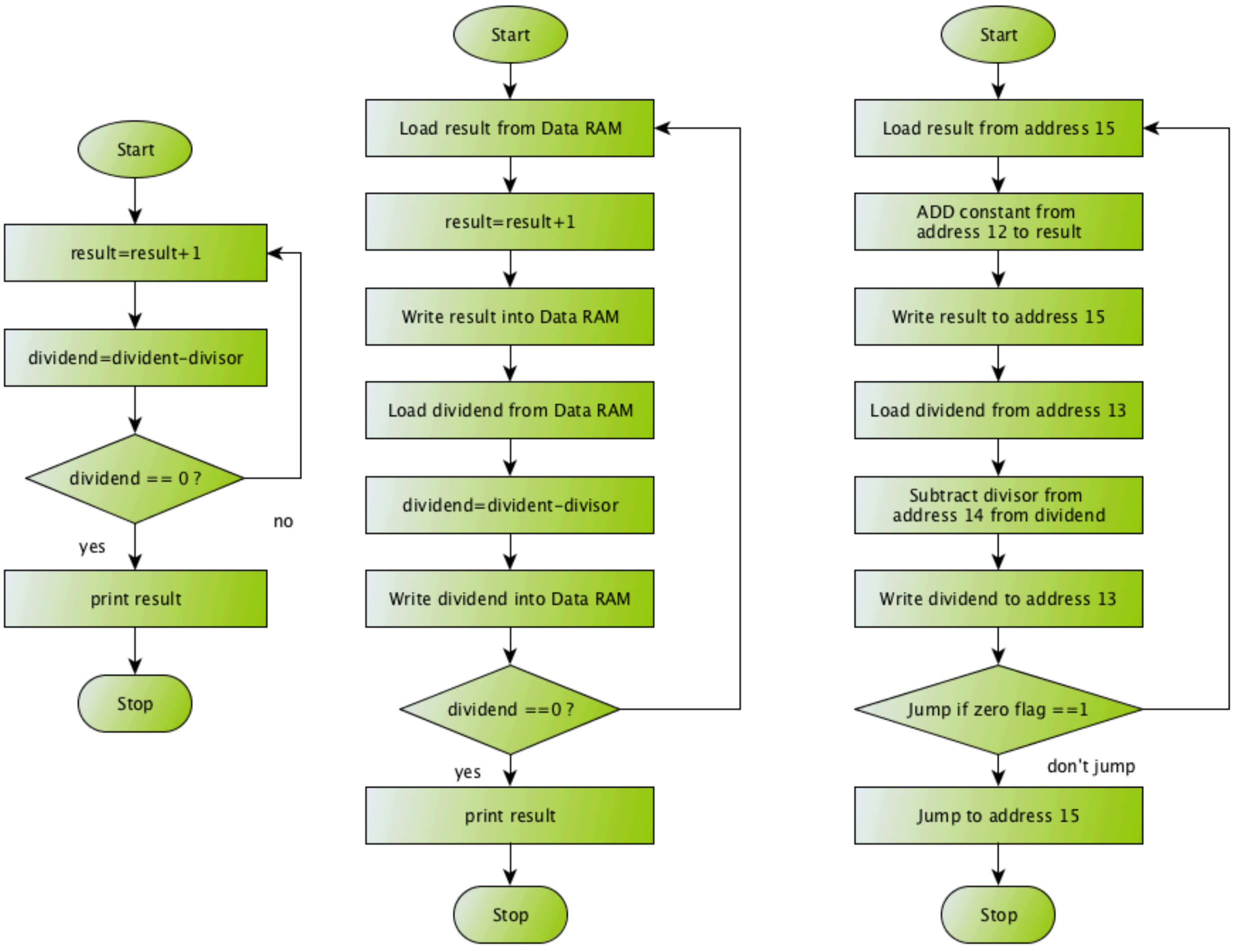
Question 9.1	Compute with your B4/A
?	1x1, 2x2, 3x3, 15x1
	0x0. What do you observe? Explain the limitations of this algorithm
	4x4 What do you observe? How can the result be explained?

Computing Division

Let's recap division:

result=dividend / divisor = series of subtractions of: dividend - divisor - divisor - ....

Just like multiplication is a series of additions, so is division a series of subtractions. We subtract the divisor from the dividend repeatedly until the dividend is zero. We count the number of subtractions in the result variable. Let's try 8 divided by 2. We compute 8-2-2-2-2=0. That's 4 subtractions, so the result of 8/2 is 4. Let's look at this algorithm in a flowchart. On the left, we have a high-level representation, which is becoming more detailed in the middle and on the right.


Start
result=result+1
dividend=divident-divisor
dividend == 0 ?
no
yes
print result
Stop
Start
Load result from Data RAM
result=result+1
Write result into Data RAM
Load dividend from Data RAM
dividend=divident-divisor
Write dividend into Data RAM
dividend ==0 ?
yes
print result
Stop
Start
Load result from address 15
ADD constant from address 12 to result
Write result to address 15
Load dividend from address 13
Subtract divisor from address 14 from dividend
Write dividend to address 13
Jump if zero flag ==1
don't jump
Jump to address 15
Stop

15/3 Division Program in Flowchart Representation

Machine Code Representation	Assembly Code Representation

Machine Code Representation

```
  #include <B4ArithmeticExtension.h>

B4 myB4;
/*
 *  15/3 division
 */

  uint8_t DataRAMContent[] = {
    B0000, B1111, B1100, B1111,
    B1101, B1110, B1101, B0000,
    B0000, B0000, B0000, B0000,
    B0001, B1111, B0011, B0000,
  };

  uint8_t ProgramRAMContent[] = {
    B0000, B1111, B1110, B1100,
    B1111, B1101, B1100, B1010,
    B0000, B0000, B0000, B0000,
    B0000, B0000, B0000, B0000,
  };

void setup()
{
myB4.loadDataAndProgram(DataRAMContent,
ProgramRAMContent);
  myB4.programB4();
}

void loop()
{
}
```

Assembly Code Representation

```
  #include <B4ArithmeticExtension.h>

B4 myB4;
/*
 *  15/3
 */

String assemblyProgram =
"NOP(0);LOAD_A(15);ADD_A(12);WRT_A(15);LO
AD_A(13);SUB_A(14);WRT_A(13);JNZ(B0000);JM
P(15);NOP(0);NOP(0);NOP(0);NOP(1);NOP(15);N
OP(3);NOP(0);";

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

void loop()
{
}
```

result

divisor

dividend

Arduino Division Program to Compute 15 / 3

In the code above we have highlighted the parts that are in charge of the result, the divisor and the dividend. In the following table, you can see the B4/A part of that program in our familiar table representation.

	Data RAM				Opcode	Description
address/Step #	**3**	**2**	**1**	**0**		
15	0	0	0	0	NOP	result
14	0	0	1	1	NOP	divisor
13	1	1	1	1	NOP	dividend
12	0	0	0	1	NOP	constant
11					NOP	not used
10					NOP	not used

	Data RAM				Opcode	Description
address/Step #	**3**	**2**	**1**	**0**		
9					NOP	not used
8	1	1	1	1	JUMP	Jump to address 15 to display the result
7	0	0	0	0	JUMP if not zero	Jump to step 0 if dividend not zero
6	1	1	1	0	WRITE to address	Store the dividend at address 13
5	1	1	0	0	SUB from address	Subtract divisor (which is at address 14) from dividend
4	1	1	1	0	LOAD from address	Load dividend from address 13
3	1	1	1	1	WRITE to address	Store the result at address 15
2	1	1	0	1	ADD from address	Add constant from address 12 to result
1	1	1	1	1	LOAD from address	Load result from address 15
0	0	0	0	0	NOP	Landing pad for the JNZ instruction at step 7

15/3 Division Program in Table Representation

We can also look at this program as a flowchart, which you can see in the next diagram. You have already seen the two flowcharts on the left and in the middle in previous chapters. The flowchart on the right is more detailed and contains information about exactly which memory addresses the program wants to work with.

Question 9.2	Compute with your B4/A
?	4/2, 4/4, 10/1
	4/3. What do you observe? Explain the limitations of this algorithm
	4/0 What do you observe? How can the result be explained?

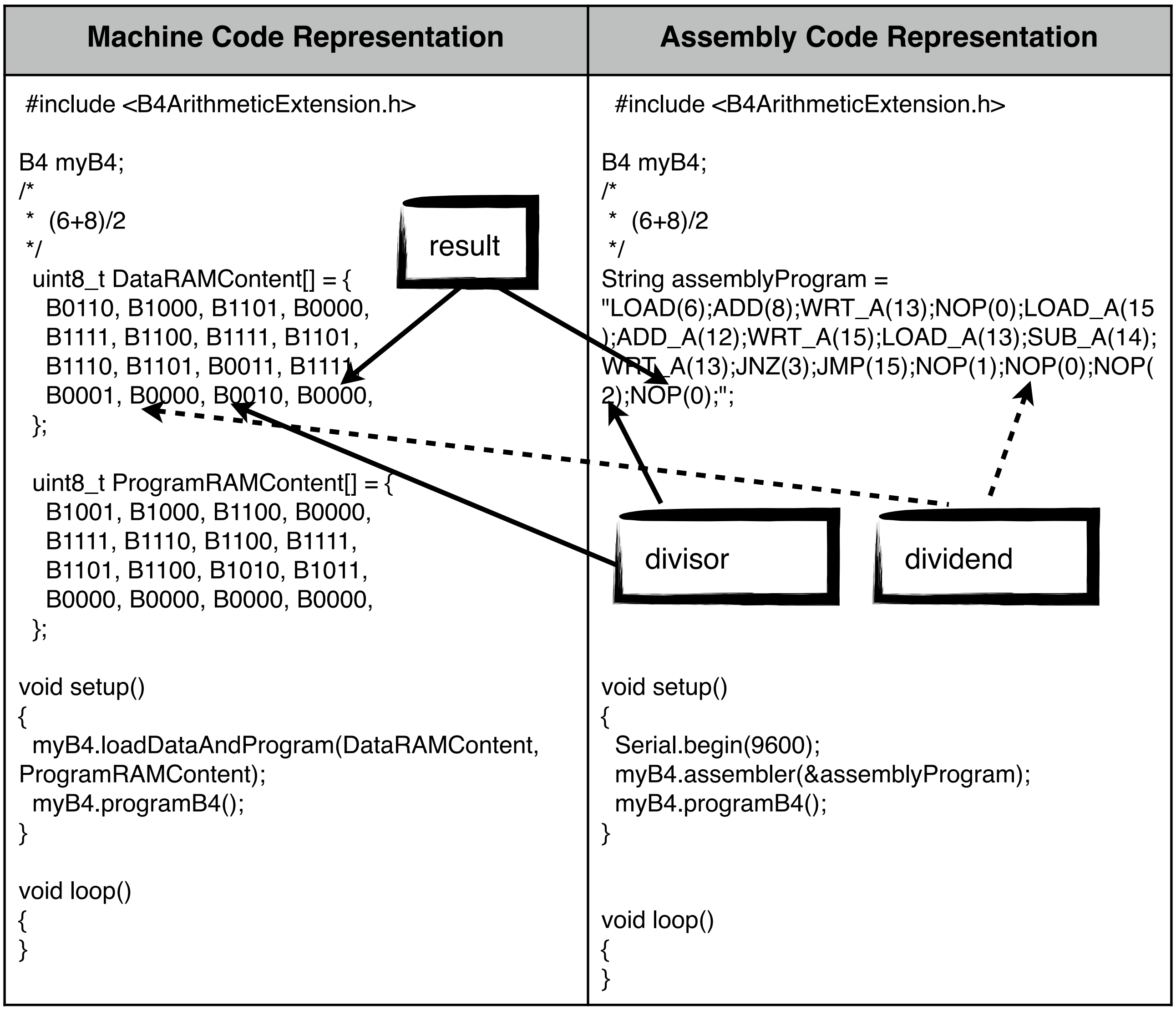
Computing Averages

To compute the average of two numbers a and *b*, we first add *a* and *b* and then divide the result by two: average = (a+b)/2.

This means we can re-use our familiar division program from the precious section to which we add three steps to the beginning of the program.

1. Load the first number,
2. add the second number
3. store the result of the addition into memory (address 13) where the dividend is.

We then update the jump address of JNZ from 0 to 3, because we added three instructions to the beginning of the program



Machine Code Representation

```
 #include <B4ArithmeticExtension.h>

B4 myB4;
/*
 *  (6+8)/2
 */
  uint8_t DataRAMContent[] = {
    B0110, B1000, B1101, B0000,
    B1111, B1100, B1111, B1101,
    B1110, B1101, B0011, B1111,
    B0001, B0000, B0010, B0000,
  };

  uint8_t ProgramRAMContent[] = {
    B1001, B1000, B1100, B0000,
    B1111, B1110, B1100, B1111,
    B1101, B1100, B1010, B1011,
    B0000, B0000, B0000, B0000,
  };

void setup()
{
  myB4.loadDataAndProgram(DataRAMContent,
ProgramRAMContent);
  myB4.programB4();
}

void loop()
{
}
```

Assembly Code Representation

```
 #include <B4ArithmeticExtension.h>

B4 myB4;
/*
 *  (6+8)/2
 */
String assemblyProgram =
"LOAD(6);ADD(8);WRT_A(13);NOP(0);LOAD_A(15
);ADD_A(12);WRT_A(15);LOAD_A(13);SUB_A(14);
WRT_A(13);JNZ(3);JMP(15);NOP(1);NOP(0);NOP(
2);NOP(0);";

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


void loop()
{
}
```

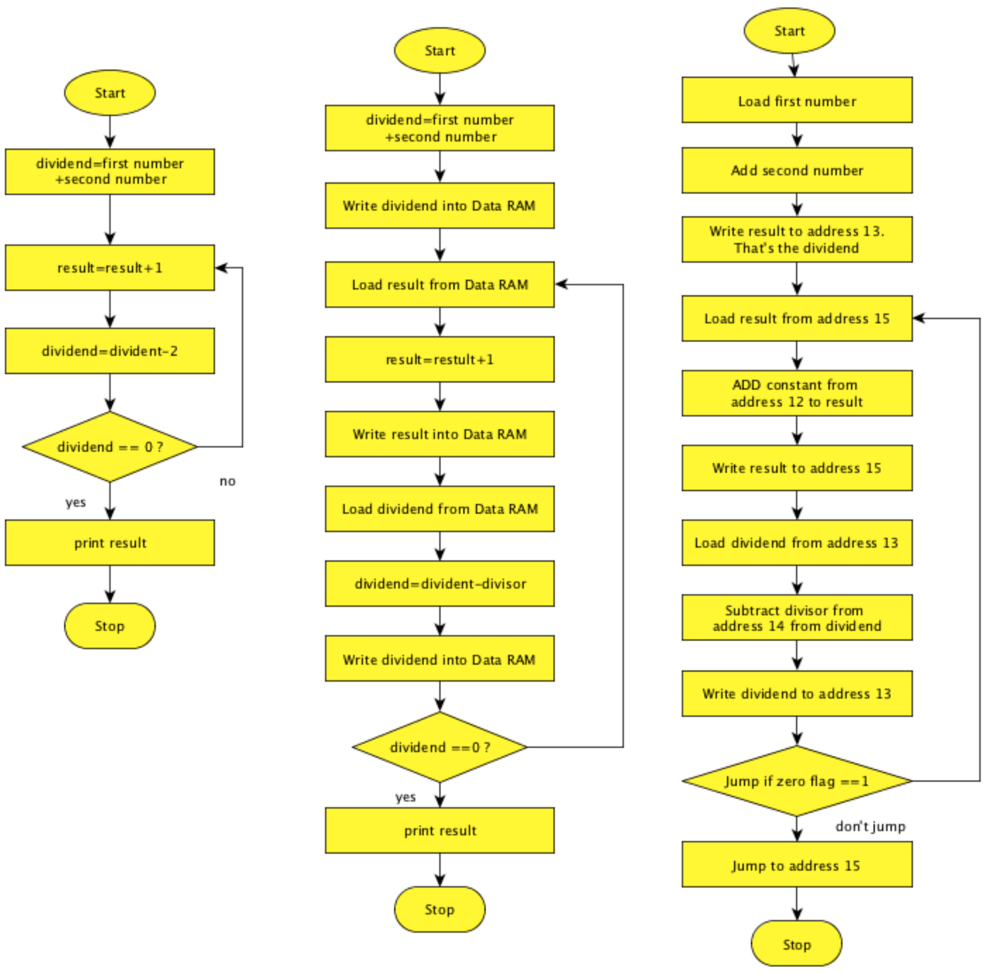
Arduino Division Program to Compute (6+8) / 2

In the code above we have highlighted the parts that are in charge of the result, the divisor and the dividend. This looks a lot like our familiar division program, which has been extended by LOAD(6);ADD(8);WRT_A(13). This computes the dividend (14), that we then divide by 2. In the following table, you can see the B4/A part of that program in our familiar table representation.

	Data RAM				Opcode	Description
address/Step #	**3**	**2**	**1**	**0**		
15	0	0	0	0	NOP	result
14	0	0	1	0	NOP	divisor
13	0	0	0	0	NOP	dividend
12	0	0	0	1	NOP	constant
11	1	1	1	1	JMP	Jump to address 15 to display the result
10	0	0	1	1	JNZ	Jump to step 3 if dividend not zero
9	1	1	1	0	WRT_A	Store the dividend at address 13
8	1	1	0	0	SUB _A	Subtract divisor (which is at address 14) from dividend
7	1	1	1	0	LOAD_A	Load dividend from address 13
6	1	1	1	1	WRT_A	Store the result at address 15
5	1	1	0	1	ADD_A	Add constant from address 12 to result
4	1	1	1	1	LOAD_A	Load result from address 15
3	0	0	0	0	NOP	Landing pad for the JNZ instruction at step 10
2	1	1	1	0	WRT_A	Store the dividend at address 13
1	1	0	0	0	ADD	Add the second number
0	0	1	1	0	LOAD	Load the first number

(6 + 8) / 2 Average Program in Table Representation

We can also look at this program as a flowchart, which you can see in the next diagram. You have already seen the two flowcharts on the left and in the middle in previous chapters. The flowchart on the right is more detailed and contains information about exactly which memory addresses the program wants to work with.


Start
dividend=first number +second number
result=result+1
dividend=divident-2
dividend == 0 ?
no
yes
print result
Stop
Start
dividend=first number +second number
Write dividend into Data RAM
Load result from Data RAM
result=restult+1
Write result into Data RAM
Load dividend from Data RAM
dividend=divident-divisor
Write dividend into Data RAM
dividend ==0 ?
yes
print result
Stop
Start
Load first number
Add second number
Write result to address 13. That's the dividend
Load result from address 15
ADD constant from address 12 to result
Write result to address 15
Load dividend from address 13
Subtract divisor from address 14 from dividend
Write dividend to address 13
Jump if zero flag ==1
don't jump
Jump to address 15
Stop

(6 + 8) / 2 Average Program in Flowchart Representation

Question 9.3	Compute with your B4/A
?	(7+5)/2, (4+2)/2
	(5+2)/2 What do you observe? Explain the limitations of this algorithm
	(8+8)/2 What do you observe? How can the result be explained?

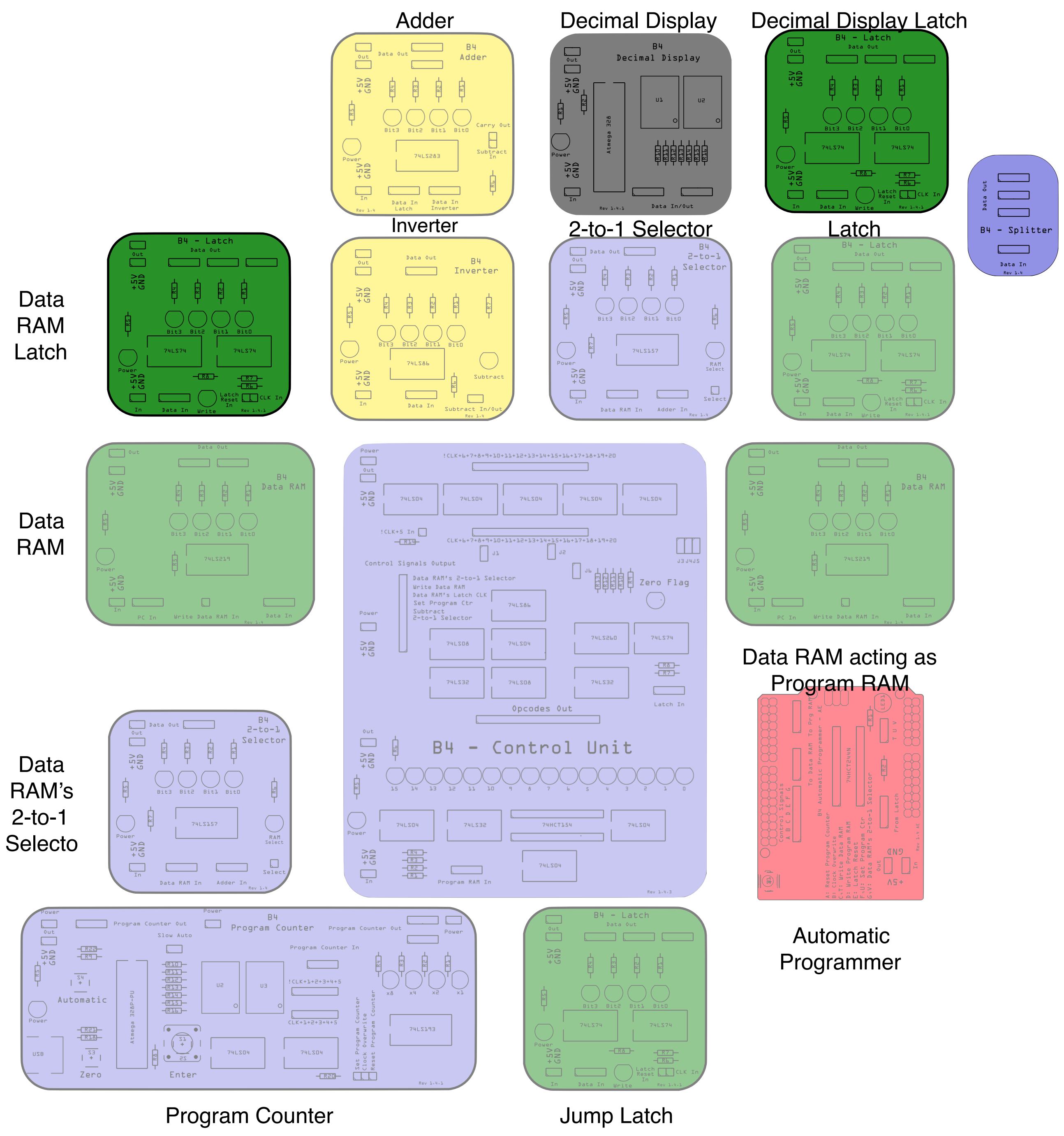
Experiment 10: Print()

To this point, we have been using programs that produced a single result, such as a multiplication or a division. The Decimal Display showed anything that happened in the Data RAM module. That was fine for as long as we knew the program well and knew at which step the final result was shown. In our programs that was at program step 15.

But what if we want to perform a sequence of calculations, such as Fibonacci numbers? We will talk about Fibonacci in the next section, but what you need to know is that a new Fibonacci number is computed every time a loop executes. So we need a way to print just the Fibonacci number to the Decimal Display once per loop. This will make the operation of our computer much more user friendly. All the user needs to do is to press the Enter button repeatedly (or use the Automatic mode) and watch the display.

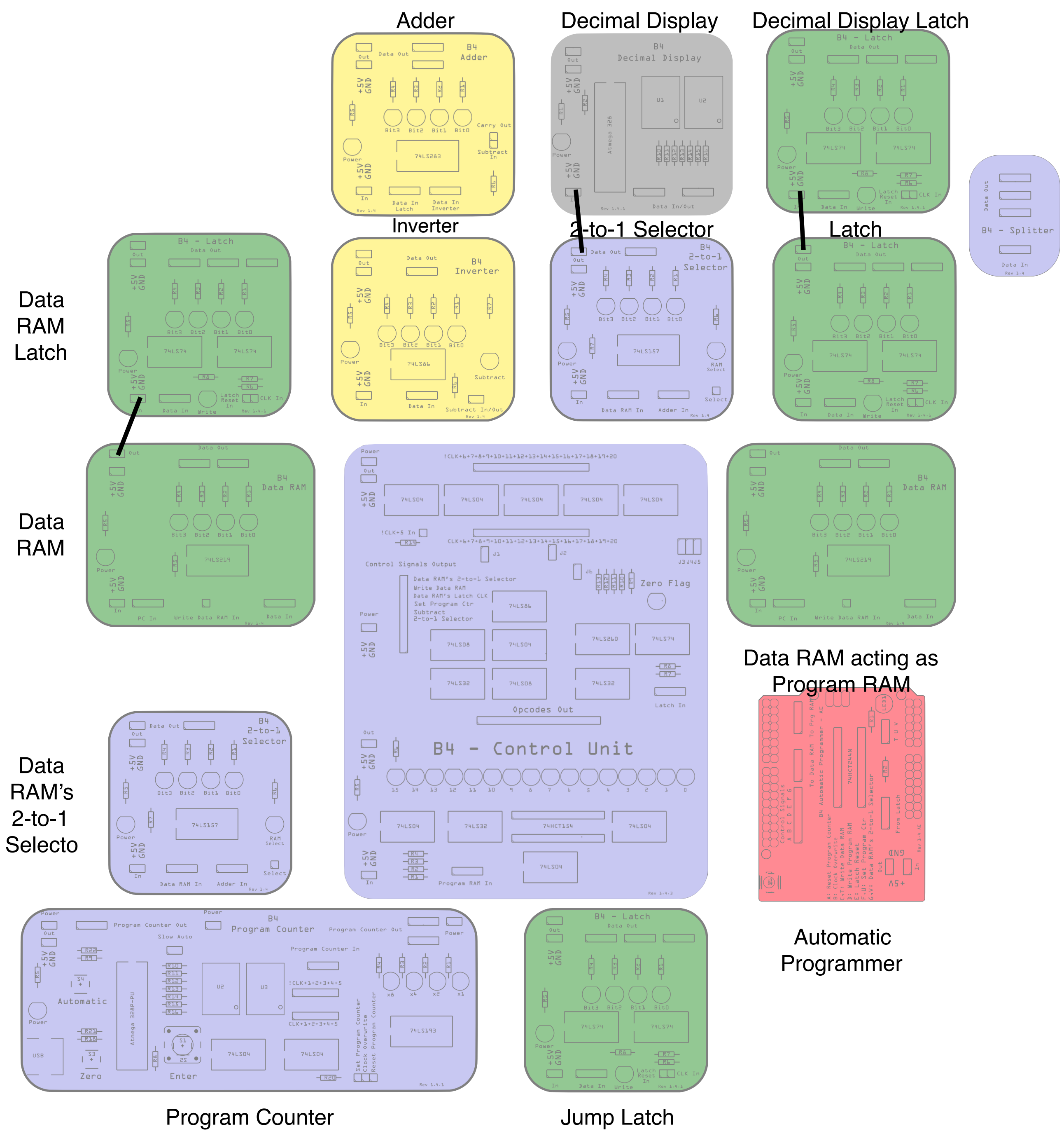
For this, we need to slightly re-design the B4/A. We re-use the design from the previous experiment. **We relocate the Decimal Display and add a Decimal Display Latch. The Data RAM Latch moves where the Decimal Display was previously. We also add the Splitter module.**

The following Figures illustrate the changes to the setup from the previous experiment.

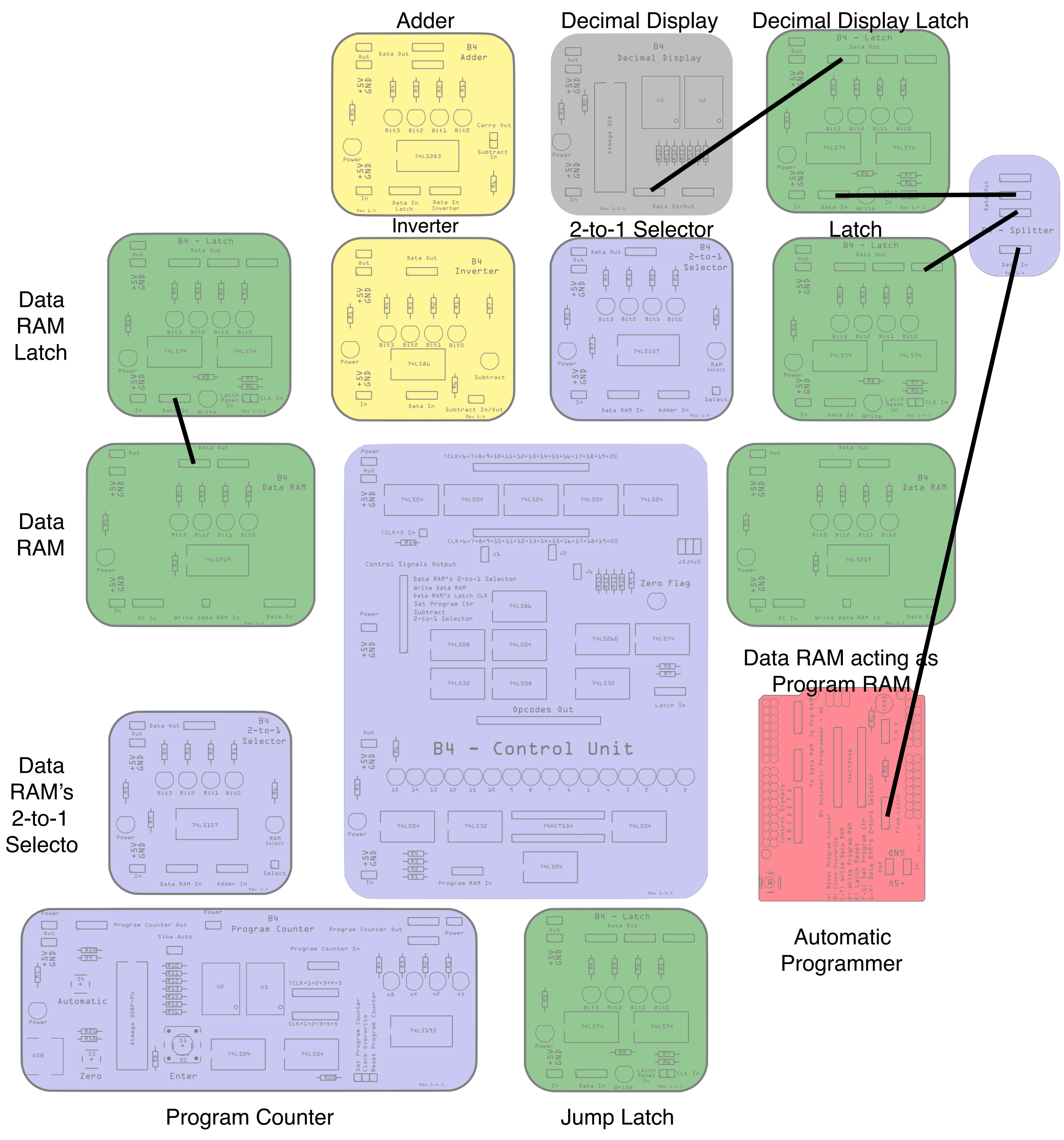


Setup of Experiment 9: Modified Module Arrangement

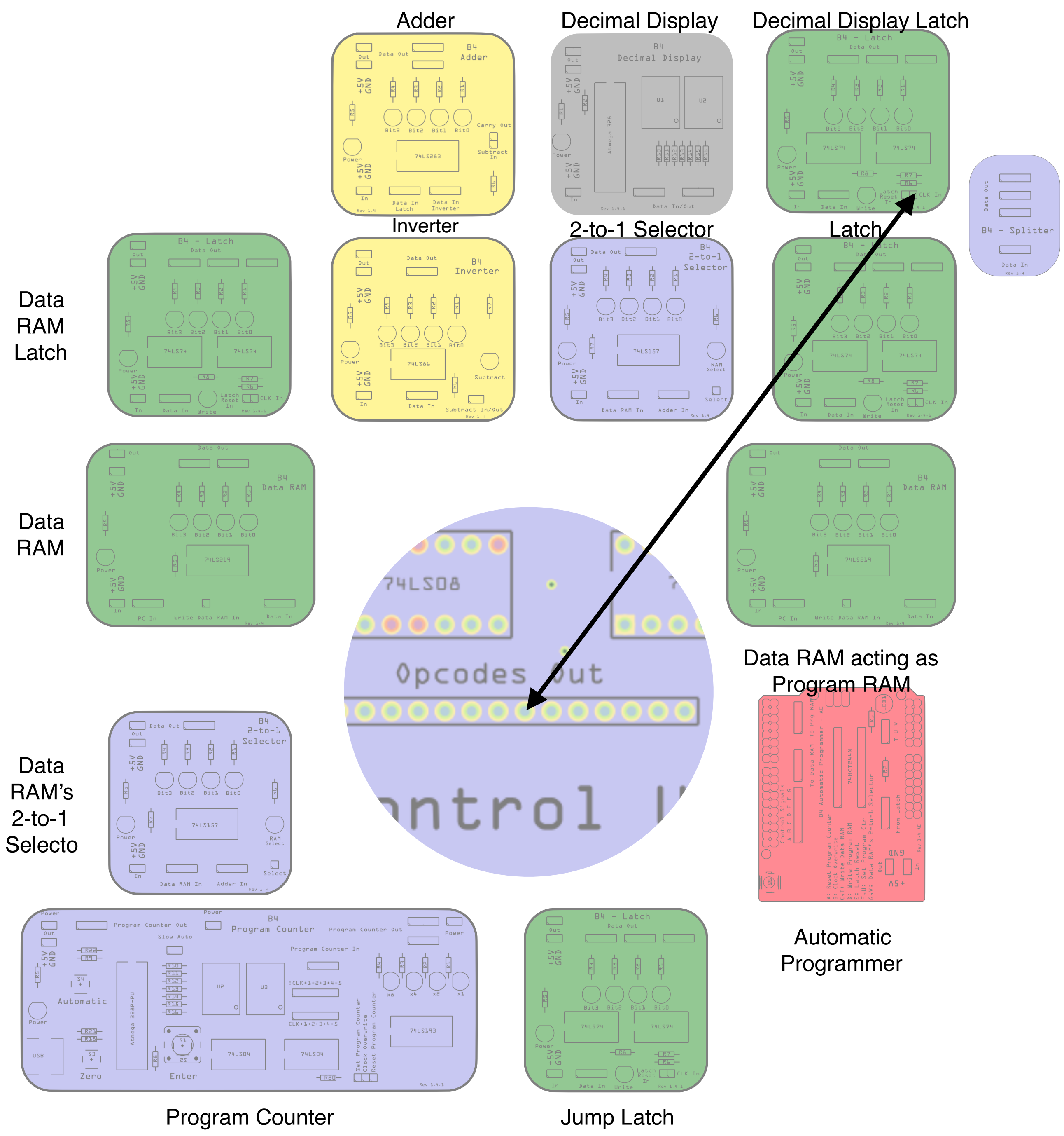
The Splitter module is required because the Latch does only have 3 outputs. However, for this experiment we need four outputs.



Setup of Experiment 9: Modified Power Wiring only



Setup of Experiment 9: Modified Data Wiring only



Setup of Experiment 9: Modified Control Wiring only

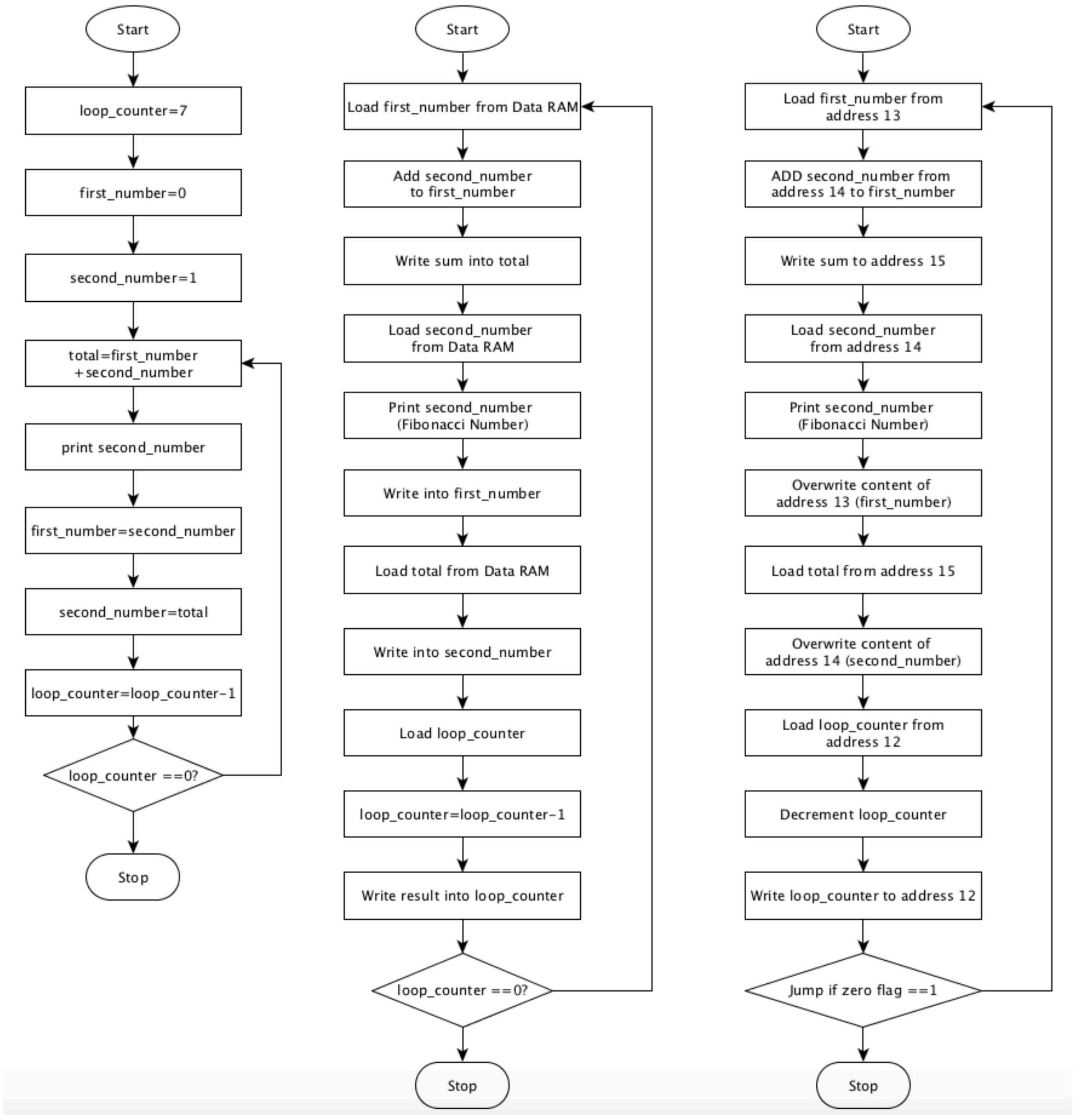
Note that a single control wire goes from the Decimal Display latch to the 7th opcode pin from the right. The opcode for the Print command is therefore B0110.

Computing Fibonacci Numbers

The Fibonacci sequence is a series of numbers where a number is found by adding up the two numbers before it. Starting with 0 and 1, the sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... They appear unexpectedly often in mathematics and in nature, from sunflowers to hurricanes to galaxies. Sunflowers seeds are arranged in a Fibonacci spiral, keeping the seeds uniformly distributed no matter how large the seed head may be. So Fibonacci numbers represent some applied mathematics.

Calculating Fibonacci numbers is rather straightforward with our B4/A. All we have to do is to start with 0 and 1 and the next Fibonacci number is the sum of its two predecessors. So, 1+2=3, 2+3=5, 3+5=8 and so forth. With our 4 bit architecture, we should be able to compute the first eight Fibonacci numbers from 0 to 13.

Let's look at this algorithm in a Flowchart:



Start
loop_counter=7
first_number=0
second_number=1
total=first_number +second_number
print second_number
first_number=second_number
second_number=total
loop_counter=loop_counter-1
loop_counter ==0?
Stop
Start
Load first_number from Data RAM
Add second_number to first_number
Write sum into total
Load second_number from Data RAM
Print second_number (Fibonacci Number)
Write into first_number
Load total from Data RAM
Write into second_number
Load loop_counter
loop_counter=loop_counter-1
Write result into loop_counter
loop_counter ==0?
Stop
Start
Load first_number from address 13
ADD second_number from address 14 to first_number
Write sum to address 15
Load second_number from address 14
Print second_number (Fibonacci Number)
Overwrite content of address 13 (first_number)
Load total from address 15
Overwrite content of address 14 (second_number)
Load loop_counter from address 12
Decrement loop_counter
Write loop_counter to address 12
Jump if zero flag ==1
Stop

Fibonacci Program in Flowchart Representation

Let's start with the left hand side, which is a more pseudo-code style flowchart. We begin by setting the loop_counter variable to 7 (because we want to compute the first eight Fibonacci numbers). We then initialise the two variables first_number to 0 and second_number to 1. We then print first_number. **As you will see, at the beginning of each loop, second_number will always contain the Fibonacci_number**. We then compute the total of first_number+second_number. first_number then remembers the value of second_number and second_number remembers the total. Finally, we decrement the loop_counter and check if we need to continue looping. That's all.

Sounds simple. Let's desk-check this code with a table:

	Command	**first_number**	**second_number**	**total**
First Iteration	total=first_number+second_number	0	1	1
	print second_number	0	1	1
	first_number=second_number	1	1	1
	second_number=total	1	1	1
Second Iteration	total=first_number+second_number	1	1	2
	print second_number	1	1	2
	first_number=second_number	1	1	2
	second_number=total	1	2	2
Third Iteration	total=first_number+second_number	1	2	3
	print second_number	1	2	3
	first_number=second_number	2	2	3
	second_number=total	2	3	3
Fourth Iteration	total=first_number+second_number	2	3	5
	print second_number	2	3	5
	first_number=second_number	3	3	5
	second_number=total	3	5	5
Fifth	total=first_number+second_number	3	5	8
	print second_number	3	5	8

	Command	first_number	second_number	total
	first_number=second_number	5	5	8
	second_number=total	5	8	8
Sixth Iteration	total=first_number+second_number	5	8	13
	print second_number	5	8	13
	first_number=second_number	8	8	13
	second_number=total	8	13	8
Seventh Iteration	total=first_number+second_number	8	13	21(5 in the B4)
	print second_number	8	13	8
	first_number=second_number	13	13	5
	second_number=total	13	5	5

You see the grey fields contain the Fibonacci number. In the seventh iteration, our B4/A reaches the limits of its 4 bit architecture, but we still get the final Fibonacci number (13).

Assembly Code Representation

```
#include <B4ArithmeticExtension.h>

B4 myB4;
/*
 *  Fibonacci
 */
String assemblyProgram =
"LOAD_A(13);ADD_A(14);WRT_A(15);LOAD_A(14);PRINT();WRT_A(13);LOAD_A(15);WRT_A(14);LOA
D_A(12);SUB(1);WRT_A(12);JNZ(15);NOP(7);NOP(0);NOP(1);NOP(0);";

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

void loop()
{
}
```

Arduino Fibonacci Program

In the following table, you can see the B4/A part of that program in our familiar representation.

	Data RAM				Opcode	Description
address/Step #	**3**	**2**	**1**	**0**		
15	0	0	0	0	NOP	total
14	0	0	0	1	NOP	second_number
13	0	0	0	0	NOP	first_number (Fibonacci Number)
12	0	1	1	1	NOP	loop_counter
11	1	1	1	1	JNZ	Jump if loop_counter not zero
10	1	1	0	0	WRT_A	Store the loop_counter at address 12
9	0	0	0	1	SUB	Decrement loop_counter by 1
8	1	1	0	0	LOAD_A	Load loop_counter from address 12
7	1	1	1	0	WRT_A	Store second_number at address 14
6	1	1	1	1	LOAD_A	Load total from address 15
5	1	1	0	1	WRT_A	Store first_number at address 13
4	0	0	0	0	PRINT	Print the Fibonacci Number on the Decimal Display
3	1	1	1	0	LOAD_A	Add second_number from address 14
2	1	1	1	1	WRT_A	Store the total at address 15
1	1	1	1	0	ADD_A	Add second_number from address 14
0	1	1	0	1	LOAD_A	Load first_number from address 13

(6 + 8) / 2 Average Program in Table Representation

Loading and Running the Program

Load the program into the B4/A and run it. You will see that the Decimal Display will show the sequence 1,1,2,3,5,8,18. Because we can’t automatically reset the Decimal Display Latch it is possible that the initial value on the Display is not 0. In this case, you can manually reset the Decimal Display Latch with a 1-pin wire.

Summary and Conclusions

In this course we have learned how computers apply algorithms to compute any maths that is not addition or subtractions. We have seen how loops allow for repeat addition leading to multiplication. Repeat subtraction is division. By combining these concepts we can teach computers to compute averages and even Fibonacci numbers.

We have also learned that we cannot just mutate (change) code to make the computer do these things, we also had to significantly upgrade its hardware to physically enable it to meaningfully execute the new code.

You now know more about the inner workings of computer processors than most people on this planet. Keep up the good work and keep pushing forward.

Challenge

Just like multiplication is a series of additions, so is power a series of multiplications. And we already know that the B4/A can perform multiplication. All we need to to is to repeat multiplications. Let's look at two examples:

5^3 = 5x5x5 = (5+5+5+5+5)*5 = (5+5+5+5+5) + (5+5+5+5+5) + (5+5+5+5+5) + (5+5+5+5+5) + (5+5+5+5+5)

In the above example, we compute 5x5 and add the result to itself 4 times.

2^3 = 2x2x2 = (2+2)x2 = (2+2) + (2+2)

Here, we compute 2x2 and add the result to itself 1 time.

Can you design and implement an algorithm that computes the the power of two numbers? For example 2^3?

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/
Fibonacci Numbers: https://en.wikipedia.org/wiki/Fibonacci_number

Appendix A: Programming Table Template.

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

	Data RAM				Program RAM					Description
Step #	3	2	1	0	3	2	1	0	Opcode	
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: Solutions

Question 2.1		Solution
?	What was the reason for leaving the 1 pin wire from the Program RAM's port D to the Program Counter's Set Program Counter disconnected during the programming phase?	If we left it connected during the programming phase, then, the Program Counter would be influenced by the Program RAM, which would lead to an incorrect storage of program instructions.
	What would have happened if we had not disconnected the wire. Explain your thinking and conduct an experiment to verify it.	

Question 9.1	Compute with your B4/A	Solution
?	1x1, 2x2, 3x3, 15x1	
	0x0. What do you observe? Explain the limitations of this algorithm	Our algorithm first decrements the dividend and then checks if it is zero. In the case of a zero dividend, the subtraction by 1 would lead to 15 and the loop would run another 15 times before it eventually reaches 0
	4x4 What do you observe? How can the result be explained?	4x4 = 16, which is B10000, or B0000 in a 4-bit architecture

Question 9.2	Compute with your B4/A	Solution
?	4/2, 4/4, 10/1	
	4/3. What do you observe? Explain the limitations of this algorithm	4 cannot be divided by 3 without rest. Our algorithm relies on a 0 to end the loop. But 4-3 is 1 and 1-3 is -2, which is 14 in a 4-bit architecture. Our algorithm will produce the wrong result.
	4/0 What do you observe? How can the result be explained?	Our algorithm first decrements the dividend and then checks if it is zero. In the case of a zero dividend, the subtraction by 1 would lead to 15 and the loop would run another 15 times before it eventually reaches 0

Question 9.3	Compute with your B4/A	Solution
?	(7+5)/2, (4+2)/2	6, 3.
	(5+2)/2 What do you observe? Explain the limitations of this algorithm	7 cannot be divided by 2 without rest. Our algorithm relies on a 0 to end the loop. But 7-2 is 5, then 5-2=3 and 2-2=1. Finally, 1-2 is -1, which is 15 in a 4-bit architecture. Our algorithm will produce the wrong result.
	(8+8)/2 What do you observe? How can the result be explained?	8+8 is 16, which is B0000 in a 4-bit architecture. Our algorithm will try to compute 0/2, which is, surprisingly 8. This is because of the nature of our algorithm, which first computes dividend-divisor. So, the first operation will be 0-2, which is 14 in a 4 bit architecture. The loop runs 8 times.

Appendix C: Quick Reference Guide

Name	Mnemonics	Machine Code	Set 2-to-1 Selector	Activate Inverter	Set 2-to-1 Selector of Data RAM (PC values)	Set Program Counter	Output Latch
Load from address	LOAD_A	1111	1		1		
Add from address	ADD_A	1110			1		
Subtract from address	SUB_A	1101		1	1		
Store at address	WRT_A	1100			1		
Jump to address	JMP	1011				1	
Jump if not zero to address	JNZ	1010				1	
Load (Absolute)	LOAD	1001	1				
Add (Absolute)	ADD	1000					
Subtract (Absolute)	SUB	0111		1			
Print on Decimal Display	PRINT	0110					1
Do Nothing	NOP	0000					