Getting Started with Assemblox x86

Nicholas Stam

assemblox.nick-stam.com

Legal

Regarding Intel:

Assemblox x86 is a version of Assemblox designed to simulate the 8086 system developed by Intel Corporation. Assemblox is **not** affiliated with Intel Corporation. The Virtual 8086 system provided by Assemblox is intended to be a simplified version of the real system for educational purposes, and does not accurately represent the specifications of Intel devices and components. Assemblox is designed to be an educational tool to teach users how to work with the x86 instruction set. Continuing in this manual, x86 will be used as shorthand to refer to the instruction set of the Intel 8086. The author, Nicholas Stam, acknowledges that Intel/Intel Corporation, are registered trademarks of Intel Corporation. Any further references to the brand identity of Intel exist to contextualize the educational simulation and make no claims to affiliation with or ownership of the intellectual property of Intel Corporation.

Note

Author's Note:

Thank you for your interest in Assemblox! Please note that this software is still in the early stages of development. Please contact me at nicholas.stam@hope.edu if you observe any inaccuracies or encounter any bugs while using Assemblox. My goal is to keep the program simple, but as an educational software, I do not want to mislead users into believing these systems work in a way that is not accurate. For these reasons feedback is greatly appreciated.

Contents

1: The Assemblox GUI

Launching Assemblox for the first time can be overwhelming. Hopefully, this quick breakdown of the GUI helps you understand the segments of the GUI, as each individual piece is quite simple.

The complete GUI should look something like this:

Figure 1.1: The Assemblox GUI

There are three black ribbons designating the three major windows on the GUI. These are:

-The Computer Pane

-The Code Pane

and **-The System Pane**

At any time, you can left-click on these black ribbons to expand the selected pane.

Figure 1.2: The location of the Pane Ribbons

You can also expand and contract panes by clicking and dragging the delimiters between panes to customize your experience and focus on different parts of Assemblox at different times.

The Computer Pane

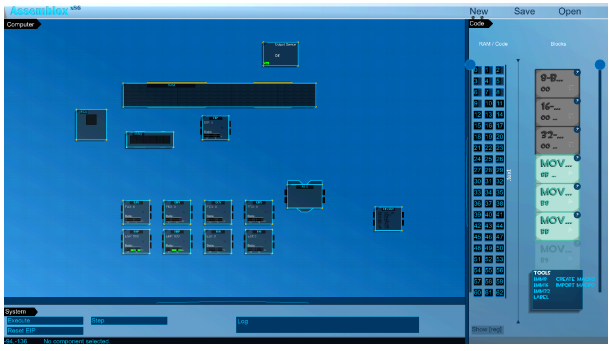


Figure 1.3: The computer pane

The most prominent of these panes is the computer pane. It expands across the entire screen but is unobstructed in the top left portion of the GUI. The **mouse wheel** can be used to zoom in and out of the computer pane. The computer pane is where you can access different **components.** The computer pane is most useful for observing or accessing the simulated hardware directly, and nothing you do in the computer pane will directly modify the contents of your program.

The Code Pane

The pane users will spend time on is the code pane. The code pane consists of two main sections, split into two columns. These columns can be resized by dragging the delimiter **(A)** between them either left or right.

Figure 1.4: The Code Pane marked A through H

The **left** column is the **RAM (B).** This is different than the RAM that we will see later in the computer pane, but they represent the same thing. The most important thing to know is that this is where you will actually change the contents of the RAM when you want to. **This is where all of the blocks that make up your program will be placed and shown.**

The **right** column is the **BLOCKS (C).** Think of these as the building blocks that you have to work with. You can click on any of these blocks to copy it. You will hear a sound and see the block now transparently hovering under your mouse. When you drag the block into RAM, you can see how many spots it takes. Then, left clicking will place that block into RAM. You can **right click** on a block in ram to erase it. We will go into more detail about this when we talk about creating programs.

There are two **scroll bars (D).** One is to the left of the RAM, and the other is to the right of the BLOCKS. These scroll bars help you scroll through the two columns.

You will be able to change the **shape** of your RAM using the selector at place **(E)**. **This feature is not yet implemented, so if your RAM disappears, make sure it is set to "GRID".**

The button labeled **show [reg] (F)** will be useful for watching how the computer reads your program. I recommend turning this on, although it can make the RAM section a bit more complicated. This shows where registers are "pointing" to when used for register-indirect addressing. If this doesn't make sense, don't worry - registers will be explained later.

The **toolbox (G)** contains several tools and frequently-used blocks to make programming the virtual 8086 faster. "Imm8", "imm16", and "imm32" (also called byte, word, and double-word) can also be found at the top of the blocks section. Labels and macros are also useful tools in the toolbox.

Finally, the **section label (H),** along with the color of the RAM, tells you more information about how the **Operating System** has segmented the memory. The five sections are: .text, .data, .bss, .stack, and the display buffer. For now, we only need to worry about the .text section, which is conveniently at the top. Most of the time, user-placed blocks will go into this section.

The System Pane

The third and final pane is the System Pane. Similar to how the Code pane gives us direct control over the RAM, the System Pane gives us control over the ECU (Execution Control Unit) and Processor State.

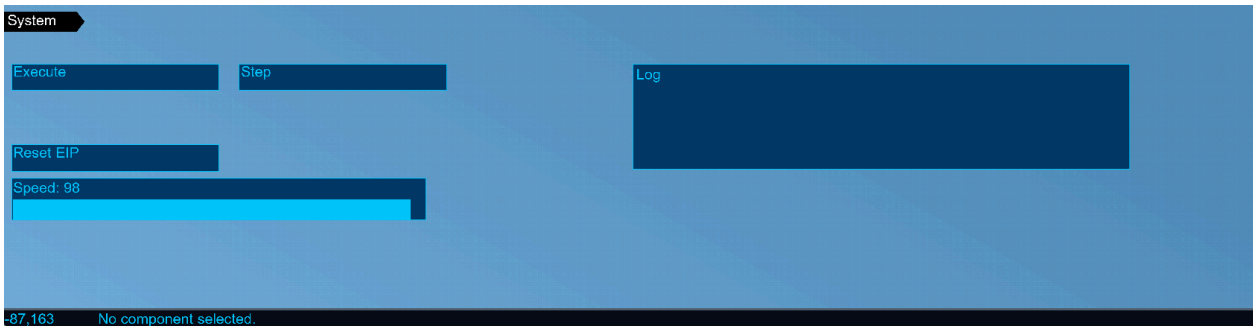


Figure 1.5: The System Pane

As of this version, the system pane is very simple. Here is what you can do from the system pane:

-Start the Execution of a Program, with the **Execute Button.** As soon as this button is pressed, the system will begin executing instructions. This button will toggle itself into a **STOP BUTTON,** so clicking this button twice will start executing instructions, then issue a processor halt and stop executing instructions.

-Execute only one instruction, with the **Step Button.** This button is similar to the execute button, except it starts the processor with a special hidden flag that causes the processor to issue a halt to itself after one instruction is executed. This is a great **debugging** tool.

-Reset the EIP register, with the **Reset EIP** button.. Although you can manually reset EIP (a register) on the computer pane, it is something that needs to happen frequently, so it is useful to have this button to quickly command the OS to reset EIP for you.

-Change the clock speed with the slider. Keep in mind that you might not see this slider unless the pane is expanded enough. You can set the clock speed to be slow, if you'd like to see each instruction one by one, or fast if you just want to test your program's correctness.

2: Introduction to the Virtual 8086

The heart of Assemblox is the virtual 8086. It is an incomplete and simplified representation of the Intel 8086, a real computer developed in 1978, and upgraded to 32-bit in 1985. Assemblox contains a virtual version of the 32-bit machine. Assemblox has cut out most of the complicated bits and only shows you the most **important components** of the actual 8086.

Components

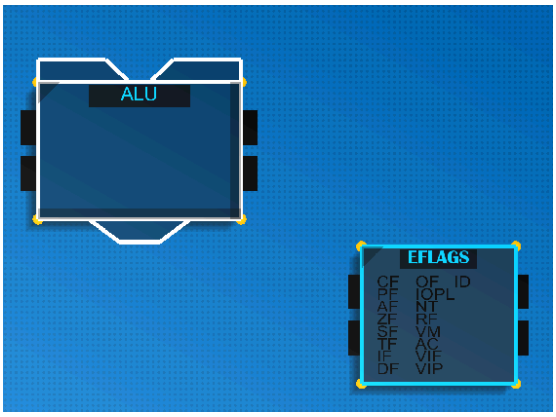
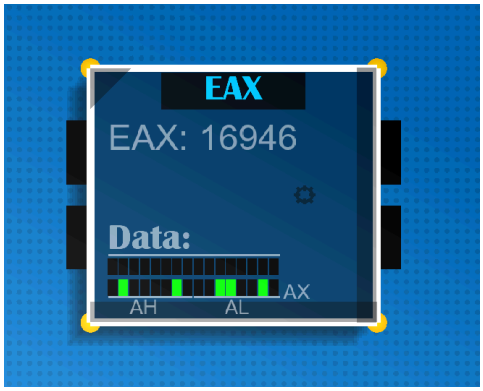


Figure 2.1: Components

The virtual computer is made of several **components**. These components are the pieces that appear in the **computer pane** and can be moved and resized. Components are visual representations of real pieces of computers. There are many types of components that all do different things, but by working together they form a complete processor. Shown to the right are two components: The ALU and the EFLAGS register. The ALU, or Arithmetic-Logic Unit is responsible for all the calculations that the computer performs. It interacts with various other components. When the system is live, these interactions are shown as flashing yellow lines. Further chapters will discuss some of the more complex components.

Registers



For the programmer, **registers** are perhaps the most important component. Put simply, registers store data. Each register holds one value, and this value can be up to 32 bits. Registers will display data as signed, so the maximum value displayed by a register is 2,147,483,647, and the minimum is -2,147,483,647. You can click the **bits** (the green or black rectangles) to modify the value in the register. Registers store values in **binary with two's compliment**. A green rectangle indicates a

one, or “on”, and the black rectangles indicate zeroes, or “off”. So, the register in Figure 2.2 depicts:

00000000 00000000 01000010 00110010

Which you may notice is segmented into four 8-bit chunks or “bytes”.

In total, there are **8 General Purpose Registers**, which each have their own number associated with them. These are the general purpose registers in order. It’s a **weird order**, it doesn’t go A, B, C, then D like you might expect.

Number	Binary	Register Name	Meaning
0	000	EAX	Accumulator
1	001	ECX	Counter
2	010	EDX	Data
3	011	EBX	Base
4	100	ESP	Stack Pointer
5	101	EBP	Base Pointer
6	110	ESI	Source Index
7	111	EDI	Destination Index

You won’t need to remember the order of these registers since Assemblox shows them by name, not number, but it’s important to be aware that it’s not so intuitive in order to prevent yourself from making a mistake. I know I still expect the second register to be EBX frequently. It’s also interesting to notice that with 8 registers, you can represent every register in just 3 bits. This explains why we don’t just add more registers to our computers - our instructions would need to be longer.

Registers are also very fast. They are close to the ALU and are used to provide information to the ALU. This makes registers the backbone of all calculations on the computer.

Imagine two friends are playing the card game “war”, but they do not know how to do math - at all. Instead, a third friend comes along who does know how to do math and decides to be the judge. The friends playing the cards are like registers: they know what card they have, and they are ready to quickly slam it down onto the table. The judge is like the ALU. It has no cards of its own, but provide it with two cards, and it will tell you who won.

Take a look at the EAX and EBX registers below. You may notice that they are covered in markings:

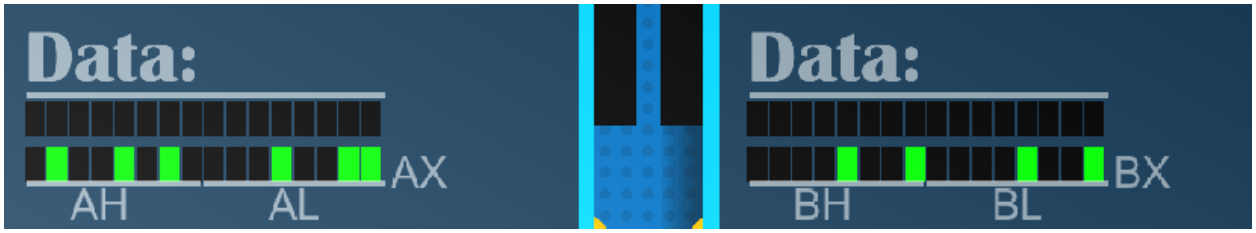


Figure 2.3: Markings under Registers

The ‘E’ in EAX and EBX stands for “Extended”. This is because, as I mentioned earlier, the 8086 was not always 32-bits. Instead, we had 16-bit registers: **AX, CX, DX, BX, SP, BP, SI, DI.** The fascinating part is that instructions involving these registers still exist and work just fine! In fact, they often use the exact same opcodes because the data is coming from the same 8 registers, just only the first 16 bits. However, this is not the case for 8-bit instructions. Each register contains two 8-bit sections, which are made up of the 16-bit section’s first 8 bits and last 8 bits. The first 8-bit is called the “High” section. In Assemblox, you can see “A-High” or AH and “B-High” or BH. The bits under that line are part of AH and BH. We also have “A-Low” or AL and “B-Low” or BL. Across the first 4 registers, we have **AL, CL, DL, BL, AH, CH, DH, and BH.** This is once again 8 “registers”, in quotes because AH and AL come from the same register, for example. Being able to represent registers with 3 bits, or 8 different values, is extremely important to how the x86 processor works.

But there are more registers that don’t need to be part of these 8 easily-addressable registers. These registers have special uses, and it’s usually the processor’s job to change them.

Because they aren't part of the 8 GPRs, we need special opcodes to change them. One of these special registers is called **EIP, or the Extended Instruction Pointer.** While we sometimes need to do math involving the value in EIP, it's much more frequently used by the Execution Control Unit. This is because **EIP POINTS TO THE NEXT INSTRUCTION IN RAM.** If the value of EIP is 55, then the instruction at ram location 55 will be loaded into the execution control unit. It is extremely important to keep EIP in-tact, otherwise, instructions may be executed at erroneous times and it could cause the end of the world. EIP is one of the registers that is under constant watch by the Operating System, as if the ECU tries to read from outside of the *.text* section the program must be killed.

Figure 2.4: EIP

Figure 2.5: EFLAGS

Another special register is called **EFLAGS**. EFlags stores information about the processor's state, which can tell us more about the results of mathematical operations from the ALU. For example, the Overflow Flag (OF) is ON (1) in Figure 2.5. This means that the result of the last ALU operation resulted in an Overflow. You can click individual flags to toggle them on or off.

In reality, EFLAGS is just a 32-bit value, just like every other register. Many of these bits have special meanings though, so instead of representing it as a number, Assemblox represents each flag as a few letters, either on or off. Some important flags to know are CF (carry flag), ZF (zero-flag), OF (overflow-flag), and (SF) sign-flag.

For now, all you need to know about registers is that they are:

1. **Extremely fast and accessible data storage**
2. **[GPRs are] addressable with just 8 values or 3 bits**
3. **The backbone of all calculations the CPU performs**
4. **EFLAGS works just like the other registers, we just read it a special way.**

RAM

Assemblox represents **RAM (Random-Access-Memory)** in two places. The first place shows the RAM **hardware** in the computer pane:

Figure 2.6: RAM Hardware

The RAM hardware component is not very interactive - it only exists to show when RAM is being accessed and by which other components. To interact with the RAM, visit the code pane section shown to the right. Interacting with RAM is very important, and will be how we provide instructions to the computer.

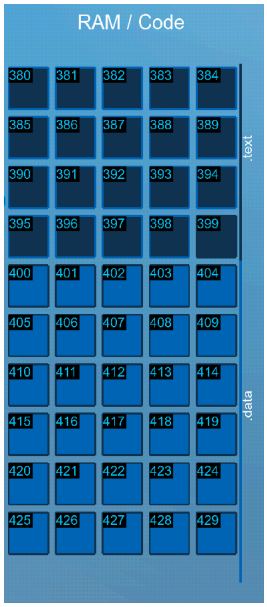


Figure 2.7 (Right)

What is RAM? RAM is fast, temporary, addressable data storage. The computer's memory is stored in bytes, where each byte is associated with an address. Other components, that are not currently displayed in Assemblox, are responsible for calculating memory addresses and retrieving values at specific addresses. By passing an address into these devices, we can retrieve saved information. RAM also requires power in order to retain values, so once the RAM is disconnected from power, all of its contents will be reset. To prevent data from being lost, use the **save** button in the top right to copy the contents of RAM to permanent storage.

Assemblox comes with 1.4 KB, or 1400 Bytes of RAM in the current free version. This isn't much at all, but it makes the memory easy to manipulate and understand at a glance. In reality, it takes a lot more than 1.4KB of your real computer's RAM to simulate 1.4KB of RAM on the virtual 8086.

As you scroll through the RAM, you will notice that the colors of the blocks change along with the labels on the right. This is because Assemblox OS has already divided your RAM into pre-prepared sections. The following is a brief description of each section and the intended purpose.

.text

.text is the section of RAM that holds **instruction opcodes**. When executing a program, EIP should begin at the first .text address. If EIP passes the last .text address, the Operating System will kill the process as a precautionary measure.

.data

.data is the section that contains **global variables**. These addresses should not be executed as instructions, but can still be read and written. For example, if you want to use the value 123456 frequently in your program, you may be able to write shorter, faster programs if you save 123456 in the data section, and simply reference the .data address containing 123456 instead of repeatedly passing it as a immediate value at multiple locations in your program.

.bss

.bss stands for block starting symbol. .bss is similar to .data, except data in the .bss section is **ALWAYS** initialized to **zero**. You may write a program that writes and reads from .bss, preferably in that order, but keep in mind that saving or exporting a program with values in the .bss section will cause the program to drop all of the values in .bss. If you intend to have global variables that start at zero anyway, consider using a .bss address for this variable, in order to not waste space in the .data section.

.stack

.stack is a designated section of memory that allows for values to be "pushed" and "popped" from the location that ESP (the stack pointer register) is pointing to. Pushing is the operation that places a value onto the "top" of the stack, and popping is the operation that takes that value off of the top and stores what it is. One example is POP to EAX, which will move the value from the top of the stack to EAX. Stack memory is particularly useful for **function calls and returns**, and **local variables**. Subroutines with their own sets of parameters will be called by passing these parameters onto the stack and popping them off in the correct order. It is important to note that the Stack grows from a higher address to a lower address. In many cases, it's helpful to think of the stack as growing from the ceiling downwards like a stalactite instead of a stalagmite, but in assemblox, the memory is organized low-to-high from top to bottom already, so the stack grows like a stack. This can be confusing, but when we get to pushing and popping, you will be able to observe the stack growing and shrinking.

Display Buffer

The final section of memory is the Display Buffer. The display buffer is a designated section of RAM that **output devices** will read from. Not every output device operates the same way, so any explanation of what this portion of RAM actually does is subject to change.

Permanent Storage

Permanent Storage in Assemblox is a bit of a joke because there is no physical computer - the entire thing exists within your computer's actual RAM. However, this isn't a problem, because if you want to save programs or macros, you can just transfer them from Assemblox to your real computer's file system as either a .abxp (Assemblox Program) or .abxm (Assemblox Macro) file. This will be covered in more detail in "Programs and Macros".

Other Visual Components

There are a few components left to briefly discuss in order to become more familiar with the virtual 8088.

ISBQ:

The Instruction Stream Byte Queue, sometimes called the **prefetch queue**, simply shows which instructions are most likely to be executed next. The real 8086 does this as a technique to speed up the processor by loading them from RAM early into a faster data transfer system.

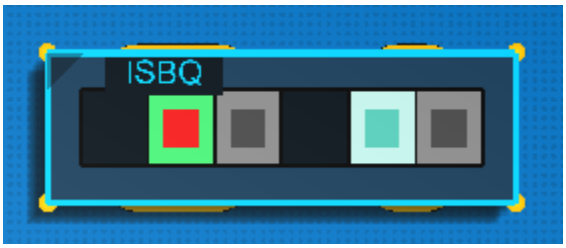


Figure 2.8: ISBQ

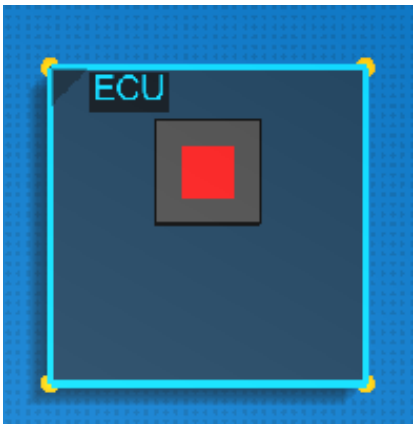


Figure 2.9: ECU

ECU:

The ECU or **Execution Control Unit** simply shows the last instruction executed. The instruction shown in the ECU has already been executed, and any yellow lines pulsing from the ECU throughout the system are a result of that instruction. In a real 8086, the ECU is the component that would contain **the Microprogram Store ROM**, which contains all the microinstructions for every Opcode.

3: The Intel x86 Instruction Set

As mentioned in the previous section, the Execution Control Unit contains **microcode**, which tells it exactly how to interact with the rest of the machine when it **decodes** an **instruction**. This is because the instruction itself is really just a sort of mapping to a location in ROM that contains the microinstructions. Even though technically it is possible to perform every task a processor is capable of through only a few instructions (a reduced instruction set computer, or RISC), Intel has provided us with *many* instructions that make life easier for the programmer, as long as we are familiar with these instructions. For example, LOOP is an instruction that decrements ECX and performs a jump. This instruction does not *need* to exist, since we can just use two instructions and achieve the same result, but it's also convenient for the programmer to use and only takes up one byte in our code. For these reasons, we call the x86 instruction set **CISC (complex instruction set computer)**.

Unfortunately, assemblox is not even close to having every possible x86 instruction. The current version, as this manual is being written, supports **75** of the 1,500+ instructions in the standard library, and there are many many more (I believe the total amount is close to **15,000**). However, if you're trying to use AVX512 to perform large matrix multiplication using Assemblox, yes I agree that would be pretty cool to watch, but I think there are better ways to do that.

In assemblox, the implemented x86 instructions are at your disposal, ready to be placed from the **blocks section of the code frame** into RAM.

Categories

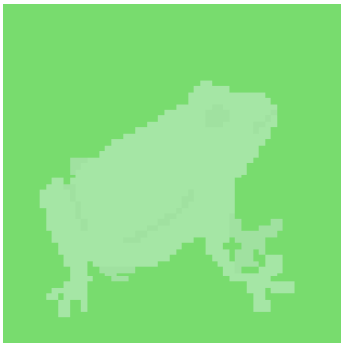
Assemblox has color-coded the x86 Instruction set for easy use. You can also look for the symbols on the instructions to learn more about the tasks they will perform.

Moving and Memory Management

This turquoise color with a stick figure person carrying a box designates that this instruction has something to do with the transfer of data **to or from memory**, or possibly from register to register.

Comparison or Question

The magenta block indicates that a comparison or "question" is being asked. This usually indicates that the processor will be setting **flags** for a future operation, and this instruction will be passed **without** modifying the contents of **RAM or registers**.



Jumps

The green block with a frog indicates that the processor will be assigning **EIP** a new value, rather than incrementing it. This either involves moving a value into EIP or performing a mathematical calculation on EIP **without** updating EFLAGS. Changing EIP effectively moves the processor to a new region of code. Jumps are essential to if-else logic and loops.

Arithmetic

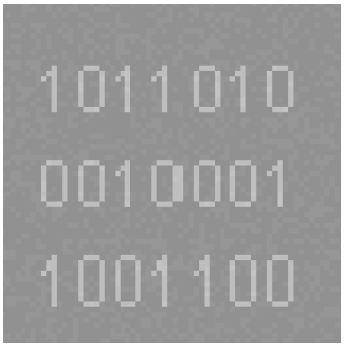
The red block containing mathematical operator symbols designates that this instruction invokes the **ALU** to perform a mathematical operation. This usually results in **EFLAGS** being updated, unless the operation specifically is designed to leave EFLAGS intact.

Functions and Subroutines

The lavender block with a lambda symbol designates that this instruction is involved in function **calls or returns**. These instructions implement the stack memory and also change the value of EIP, similar to a jump.

Direct Processor Control

The yellow block with the warning triangle indicates that this block is responsible for directly controlling the processor. For example, the HLT instruction. These blocks emulate the functionality of the system pane.



Stored and Immediate Data

Gray blocks are **data**. They are never intended to be interpreted as instructions, but rather as modifiers to instructions that provide more information, or just as numerical values.

Macros

White and Gray blocks indicate an unexpanded or **collapsed macro**. Before being decoded by the ECU, they must be **expanded** into the blocks they represent.

Some blocks belong to multiple categories. This is a result of Intel's architecture being CISC. The LOOP example from earlier is both arithmetic and a jump and therefore it is colored green and red.

Figure 3.9: The LOOP Block

For more information on what a block does, you can click the question mark in the top right of the template block. This will open a pop-up with a short description of how the instruction works. If an instruction requires you to follow it with immediate data, that too will be documented here.

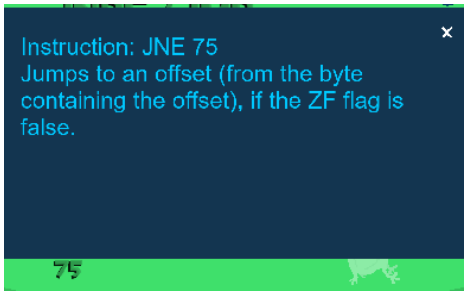


Figure 3.10: Info Box

Understanding Opcodes

An **opcode,** or **operation code** is a unique identifier for every possible instruction in assembly. Some instructions have the same name, but different opcodes. There are many ways to use the MOV instruction. You must consider: what type of data are you moving? Where are you moving it to? What operands are needed? If the answers to these questions change how the processor must act, then a different Opcode is required. The opcode is what gets decoded by the ECU into microinstructions.

Most of the opcodes in Assemblox are one byte. You can see the opcode underneath the instruction template.

Figure 3.11: Instruction with opcode 2D

Assemblox makes frequent use of shortened phrases to explain what instructions do in a quick and easy way. It's not very helpful to see: MOV 8b /R SRC: REG DEST: R/M if you don't know what it means. However, if you can read this, it provides a lot of insight into how the instruction works. Here's an explanation of some common terms and abbreviations used in machine code documentation.

imm8, imm16, imm32:

Imm stands for IMMEDIATE, and the number is a size (8 bits, 16 bits, 32 bits). This means that a value will come immediately after the instruction, which will be used somewhat like a parameter. For example, MOV EAX IMM16/32 means we will move into EAX whatever value is in the next 2 OR 4 bytes. So if the next 4 bytes contain the value 12345, then this instruction would move 12345 into EAX. If you're familiar with x86 assembly, you'll know what instructions are written like this:

Instruction destination, source

This might seem strange at first, but it makes a lot of sense. The instruction AND the destination combine to make a single opcode, and the source is an immediate value that follows so it comes last.

byte, word, doubleword:

These are names given to different data sizes. A word is two bytes, and a double word is 4 bytes. In x64 architecture, there are also quadwords.

R/M:

R/M means "**Register or Memory**". When you see this, it's most likely involved in an instruction that will spawn a ModR/M Byte, which will be discussed in "The User, The Assembler".

Reg:

Reg is simply short for register. When you see this, the instruction requires a register to be specified as an operand.

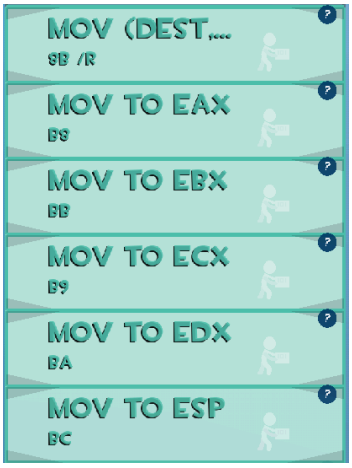
4: The User, The Assembler

The purpose of Assemblox is to bridge the knowledge gap between machine code and Assembly. Machine code is complicated and unintuitive, there's a reason Assembly was invented in the first place. But by receiving instant visual feedback Assemblox teaches users how a computer really works in a fun and exciting way.

The truth is, only one program ever *needed* to be written in machine code: the first **assembler**. An assembler is the code that translates the human-readable assembly language into opcodes. (The assembler couldn't be written in assembly, because assembly was useless without an assembler already existing!) An assembler is comparable to a compiler for assembly.

The assembler's job is not as simple as reading the instruction byte by byte and converting it to opcodes. It takes a very complicated algorithm to cross-reference operand types and prefix-bytes and instructions as various possible lengths of bytes - we can be thankful that the x86 and x64 assemblers have been written and thoroughly optimized by Intel.

The Fun Part



Assemblox has no built-in assembler. Instead, **the user's job is to assemble instructions into RAM**. You'll notice that there are many different blocks with similar names. In Assembly language, you can simply type the first portion of the name and the assembler will figure out the rest, but in Assemblox you have direct control over every operation. As you write

a program in Assemblox, think about how what you're doing is similar to what an assembler does. You may already know what you want your program to do, but choosing the correct opcodes and packing them efficiently into RAM is something that early computer designers had to spend a lot of time thinking about. But the most mindblowing part of all of this is **that this is all a computer program is.** If Assemblox wasn't as limited and was a bit faster (which in the future it might be), there would be nothing preventing you from assembling a C compiler, or Adobe Photoshop, or Microsoft Windows. In fact, if you were to look at the machine code for any of these programs on an x86 system, you would see some familiar instructions such as E8 - Call and EB - Jump. The assemblox language is real machine code, written in real machine code, interpreted on a virtual computer made out of machine code, running on an operating system made out of machine code, running on your computer.

Placing Blocks

Now that you understand what Assemblox is for, it's time to actually play around with the machine. Placing a block is simple:

Left-click the large, colorful block of your choice on the right of the blocks pane.

You will see the rectangle with the name of the block you selected appear below your mouse. In this example, it's the rectangle marked "ADD".

Figure 4.2: A selected Block

Move your mouse over to the RAM in the blocks pane. In this example, my mouse is over slot 10. As you can see, slots 11, 12, 13, and 14 have also been filled in solid red as well as outlined in red. This indicates that this instruction (Add to EAX) is 5 blocks, or 5 bytes wide. Why is this the case, if the instruction opcode was **05,** a value that can be represented in just one byte? This is because ADD to EAX requires an operand of type imm32, which is 4 bytes long. So one byte for the opcode plus four bytes for the operand creates an instruction with a width of 5. **Note: some instructions can take up less space than their width, and this is one of them.** Add to EAX can also take a 16-bit immediate value, or two bytes, because EAX and AX are the same physical register, and the Extended Registers are addressed equivalently to their non-extended counterparts.

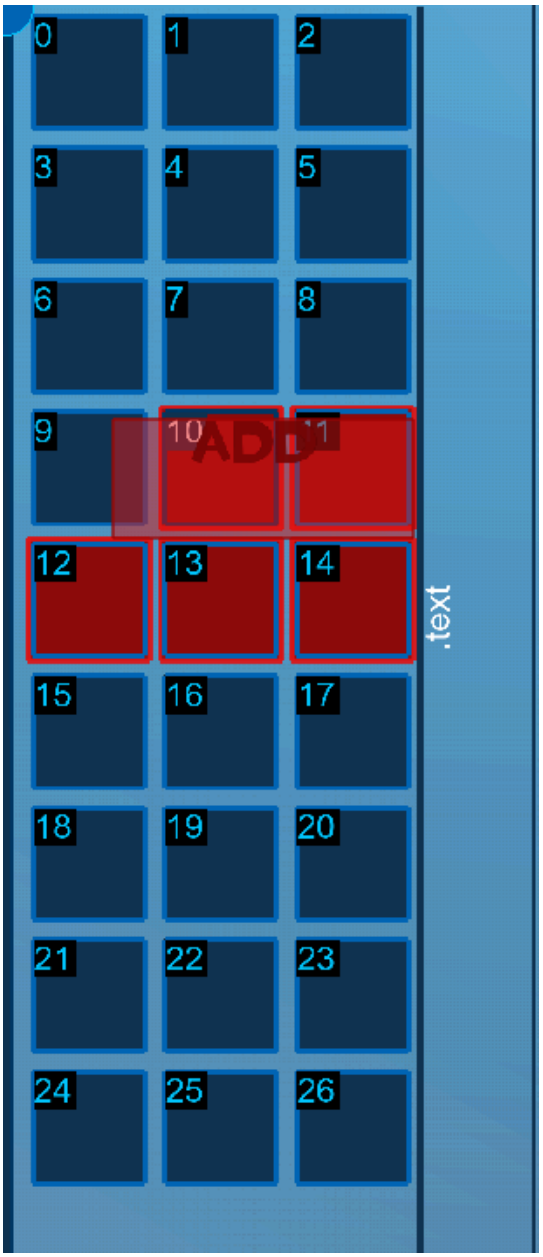


Figure 4.3: A block hovering over RAM

Left-click to place the block. You will notice that the operand spaces are no longer solid, but rather outlined. This indicates that you can place another block

Figure 4.4: A placed block

in these spaces, but they are still technically part of the ADD instruction. Typically, you only want to place immediate values in outlined spaces.

Complete the instruction with any missing operands. In the case of the ADD to EAX instruction, the instruction is missing operands. The virtual 8086 reads blank spaces as zeros - in reality, that's all they are, and it might be helpful to think of blank spaces as the exact same as gray spaces filled with zeroes. We can click and drag the imm32 block exactly the same way as the ADD to EAX block, place it into slot 11, and complete the instruction. The immediate value block has automatically opened an editor for us to customize our instruction. Because adding zero to EAX would be incredibly boring, we can type a more interesting value into this field. You don't need to click anything, we can just begin typing. I'll do the least random random number, 37. Hitting enter will close the immediate value editor, and update the block. If you want to do negative values, press the minus (-) key after typing the number. Now we have a complete instruction that will add 37 to the EAX register.

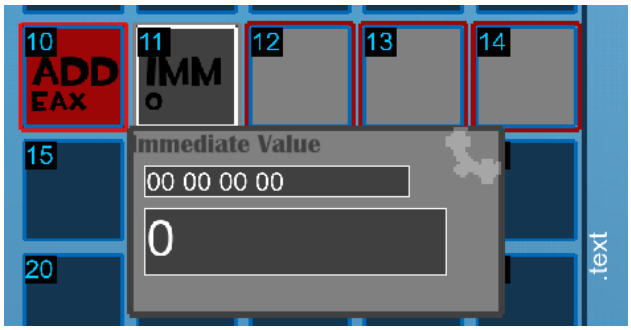


Figure 4.5: A complete instruction

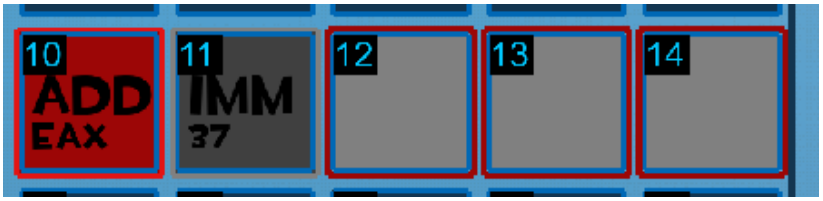


Figure 4.6: Add 37 to EAX instruction

Executing Instructions

Our simple program that adds 37 to the EAX register is complete, now it's time to test it and watch it run. Bring your attention to the System pane when you are ready to execute a program.

Figure 4.7: The System Pane

Make sure you can see the computer while it runs, then follow these simple steps.

1. **If EIP is not 0, press Reset EIP or reset it manually.** This makes sure that our program will actually encounter our new instruction.
2. **Press Execute.** And watch the magic! Pay close attention to the EAX register.

3. **You should see something like this:**

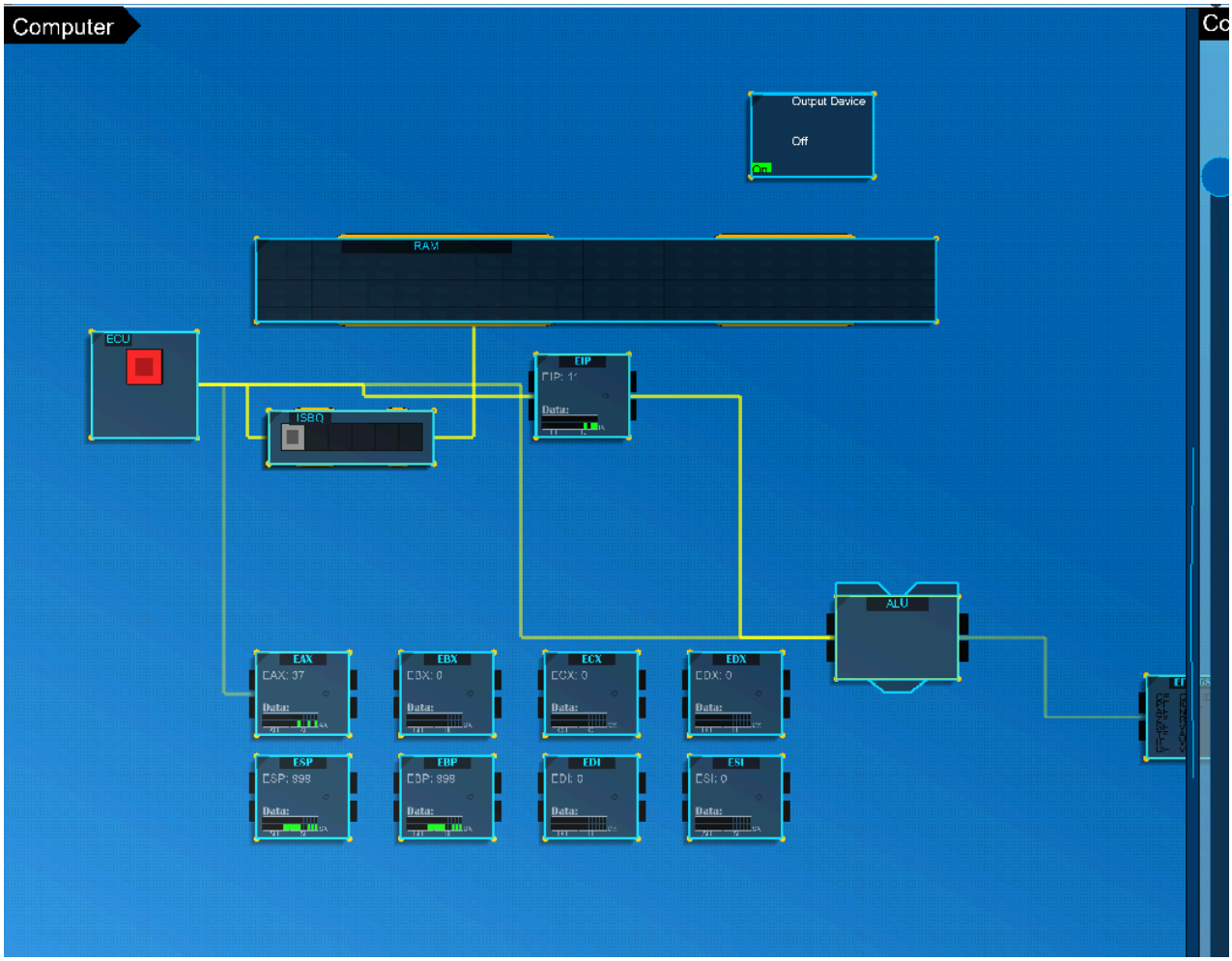
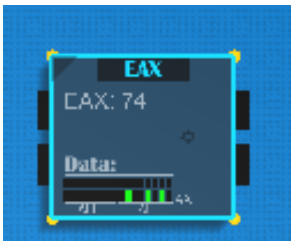


Figure 4.8: ADD to EAX, 37 being executed

Notice the lines indicating the flow of electricity through the machine. A lot of parts are involved even in such a simple instruction! The ECU read from the ISBQ, decoded the instruction, and sent a value from RAM (the 37 we placed into RAM) to the ALU along with EAX, the ALU updated EAX, Incremented EIP, and updated EFLAGS. The ISBQ then read from RAM to update itself, although the instruction it found was blank. When executing a program, you'll notice that the ALU, ECU, and EIP are almost constantly lit up.

Even with all these steps, we can see that EAX indeed had 37 added to its original value, which was zero. If we repeat the steps, we will see that EAX becomes 74 as expected.



Interpreted Assembly

The concept of interpreted Assembly/Machine Code is ironic. Although Assemblox uses real Intel Opcodes to represent instructions, behind the scenes, each block simply tells the machine where to read more machine code from, and this code contains a lot of extra information. The reason Assemblox works this way is:

1. **Each instruction secretly does a lot more than an actual 8086 instruction.** The interpreter reads the opcode and lights up specific components, draws the yellow lines, places colorful blocks, etc.
2. **Assemblox programs don't need to be compiled.** As soon as you are happy with your program, you can run it and test it.
3. **You can make live changes as the machine is running.** You can change registers, operands, and even delete bytes of machine code as the machine is running live. It's not a real computer, so you can break it however you want.

The cons of this method are that assemblox is much much much much much slower than if you were to write a real assembled assembly program and run it on real hardware. (However, if you want Assemblox to run as fast as possible, you can press the undocumented Ctrl+O keyboard shortcut to activate **overclock mode** and disable visual feedback. Make sure your program has no infinite loops because this might crash Assemblox). Use this in conjunction with Ctrl+D for **desktop mode**, which increases the FPS limit.

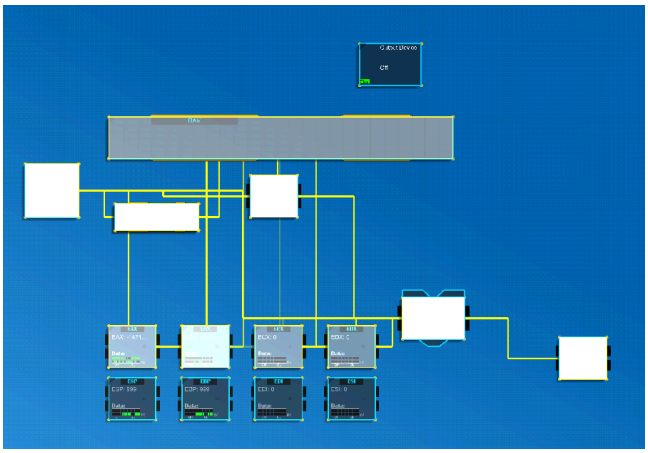


Figure 4.9: Overclock Mode

The Show [reg] Button

Once you've built your program, it can be fun and helpful to watch how the registers interact with the RAM. Registers can be treated as pointers, most notably EIP, the extended instruction **pointer**. Pressing this button shows you the RAM where each register is pointing to. As the machine is running, you can watch how these pointers move around.

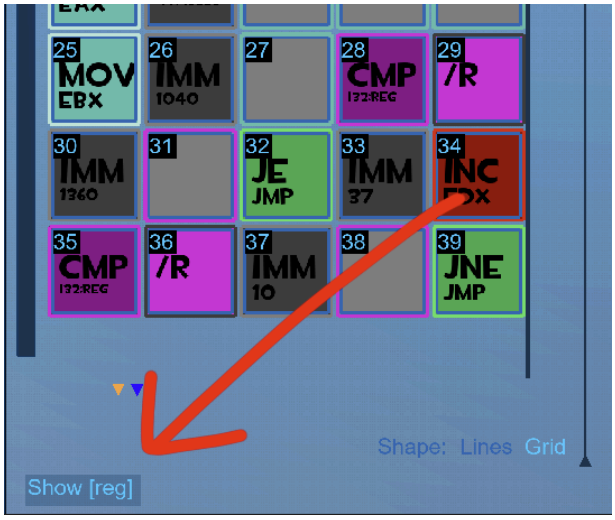


Figure 4.10: Show [reg] button

The blue labels above some bytes indicate that a register has that byte's address as its value. In the example to the right, ESI, EDI, and ECX are all zero. EDX holds the value of 4, and EIP holds the value of 35. This means that the CMP block at position 35 will be the next instruction executed. Reading a register as a pointer to a different value is called indirect addressing, and it is one of many **addressing modes** used by the 8086.

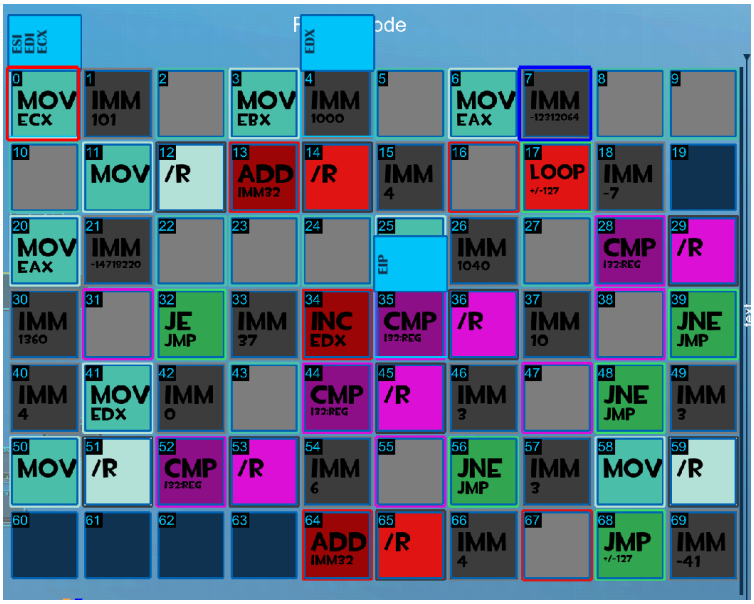


Figure 4.11: Show [reg] labels

Addressing Modes

Addressing modes are techniques used by the machine to reference data. The square brackets on the show [reg] button indicate **indirect addressing.** Another addressing mode is **register addressing**, which is the method of describing reach register as a number between 0 and 7, inclusive. The numbers given to each register in Chapter 2: Registers are register addressing.

Indirect addressing often comes with an offset, which is useful for creating **data structures** and custom data types. For example, a RGBA color may be 4 bytes or a double word, but we know that the blue portion will always be at the address of the color, plus two bytes.

The ModR/M Byte

Take a look at the following instruction and its opcode.

Figure 4.12: MOV (Dest, Src) 8B /R Instruction

This is a move instruction capable of moving data between registers and memory. It requires no immediate operands, yet is still able to move register values to hundreds of possible locations. How is this possible? It makes use of a special byte in the opcode (/R) or the **ModR/M byte.** When this instruction is decoded, the ECU reads opcode 8B, which has microinstructions indicating to continue reading the second byte and to break it down into three main pieces. This second byte is the ModR/M byte, and when we place this instruction in Assemblox, it is automatically created for us. The ModR/M byte is very fascinating. We can use just 8 bits to learn a lot of extra information about how to process this instruction. Here's how it works:

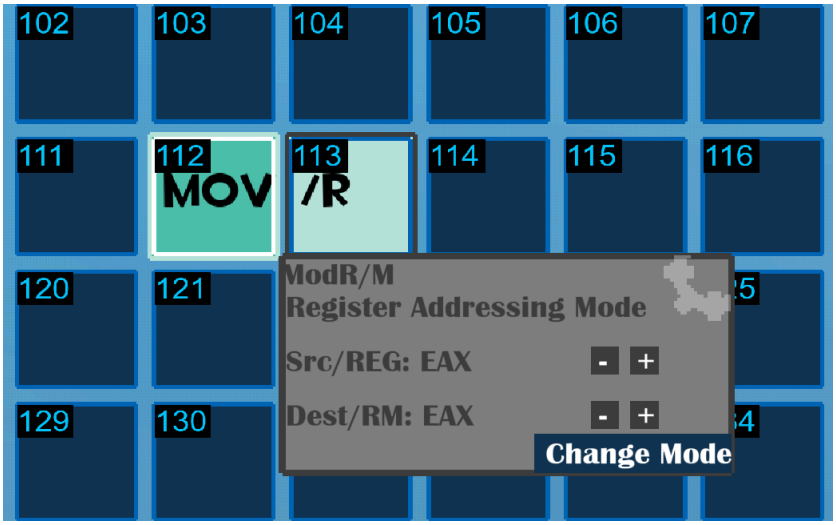


Figure 4.13: The ModR/M block

1. **The first two bits specify an addressing mode.**
2. **The next three bits specify a register**
3. **The last three bits specify a register or memory location.**

Here is a breakdown of what a real ModR/M byte would look like:

Addressing Mode		**Register**			**Register or Memory**		
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Because we have only 8 registers and 4 addressing modes, we can specify two registers and an addressing mode in just 8 bits, giving us short and fast instructions!

If we want to Move the value in register EAX into register EBX, this is the instruction we would use:

Instruction **ModR/M Byte**

MOV instruction (Hex 8B)								**Mode**		**Register**			**Register or Memory**		
								Register-Addressing		EAX (0)			EBX (3)		
1	0	0	0	1	0	1	1	1	1	0	0	0	0	1	1

And there we have it: An instruction that moves the value from EAX to EBX.

If we want to move the value from EAX to the Memory Location pointed to by EBX, written as [EBX], then we only have to change the addressing mode section to register-indirect, or mode 00.

Instruction **ModR/M Byte**

MOV instruction (Hex 8B)								**Mode**		**Register**			**Register or Memory**		
								Register-Indirect		EAX (0)			**[EBX]**		
1	0	0	0	1	0	1	1	0	0	0	0	0	0	1	1

It's unintuitive to calculate the opcode you need for the ModR/M byte by hand, so Assemblox has made it easy. Assemblox provides buttons to change the Mode, Reg, and R/M fields separately. Here's what the two previous instructions look like in assemblox:

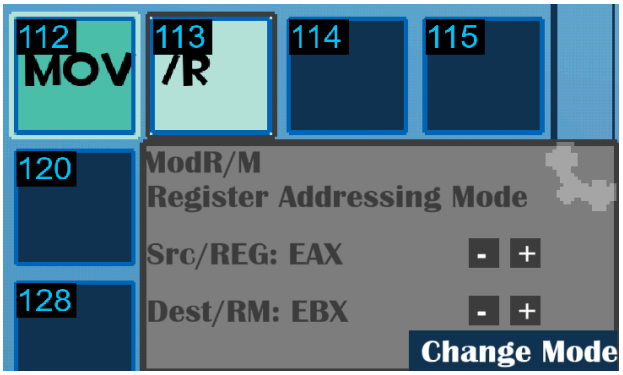


Figure 4.14

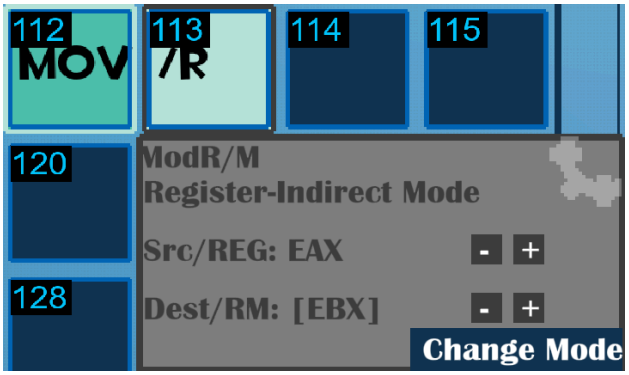


Figure 4.15

There are many different instructions that make use of ModR/M bytes, and the way the ModR/M byte is interpreted depends on the opcodes preceding it. For example, some instructions only read from the destination, while the source/reg field is treated as an opcode extension. Instructions that make use of ModR/M bytes will have more information on how they are interpreted in the instruction information box. As the assembler, the user is responsible for setting these Mod R/M bytes correctly.

5: Programs and Macros

Placing blocks one at a time can be tedious, so Assemblox has two primary ways to save sequences of blocks to speed this process up. These two methods are called **programs** and **macros**.

Programs

In Assemblox, a program is a file that can be saved and loaded that contains information for all blocks within the .**text** and .**data** sections. When a program is loaded, the RAM will be reset and the new program will be loaded in. The **new**, **save**, and **open** buttons in the top right of Assemblox all deal with programs. Programs are saved in your user folder's AppData section under the folder called /Assemblox/myprograms/. Program files end with .abxp which stands for "Assemblox Program". These files can be transferred between devices, as they contain all the information for how to place the blocks into RAM. If you press open, you will see that Assemblox already comes pre-packaged with a few example programs to test out.

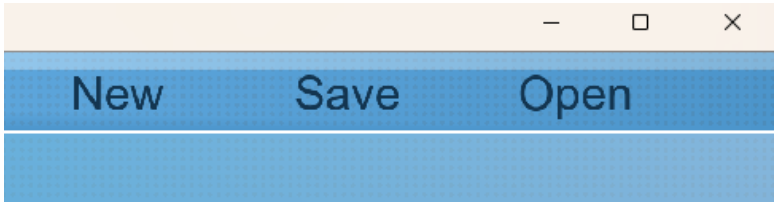


Figure 5.1: Program Buttons

Unlike macros, only one program can be open at a time. Because of this, programs should be thought of as individual project files. When you want to start a new project, press the **new button** to create a new program file.

Macros

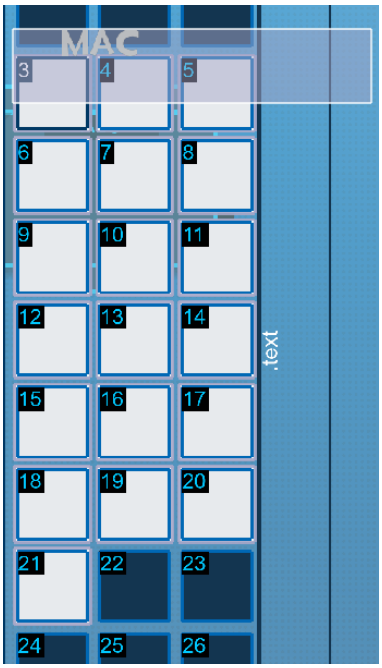
Similarly to programs, macros are files that contain information on how to place blocks. The difference is that Macros are treated like blocks themselves. Macros are useful for when you want to create custom patterns that are likely to be frequently repeated in your code, and instead

of placing each block one by one, you can simply place the macro. To create a macro, first, build a program that contains all the blocks you want to be part of your macro, then in the toolbox, press **create macro**. Input a name, a starting address, and a final address for your macro. This will generate a new file that ends with **.abxm**, or "Assemblox Macro". Then, in any program you want, press the import macro button in the toolbox. This will import your macro as a block template, underneath all of the other blocks.

Figure 5.2: Toolbox

Here I have a macro block with the name "CLEARDB". As you can see, the opcode says "Combo", because there are many different blocks contained within this macro. If we click this block, we can pick up the macro and place it into memory as if it were any other block. The white block is called a **collapsed** macro. Before it can be decoded by the ECU, it must be **expanded** into several real instructions. This happens automatically when the instruction is loaded into the execution control unit.

Figure 5.3 / 5.4 : Macro Block



Assemblox Macro files can also be transferred between devices as if they were programs. You will find them in the AppData folder, contained in /Assemblox/mymacros. Using macros frequently will make writing Assemblox code much faster. There is no limit on the amount of macros you can import into Assemblox at any time.

6: Output Devices

Assemblox comes with virtual output devices that attach themselves to the computer's RAM. You can find these output devices in the computer pane.

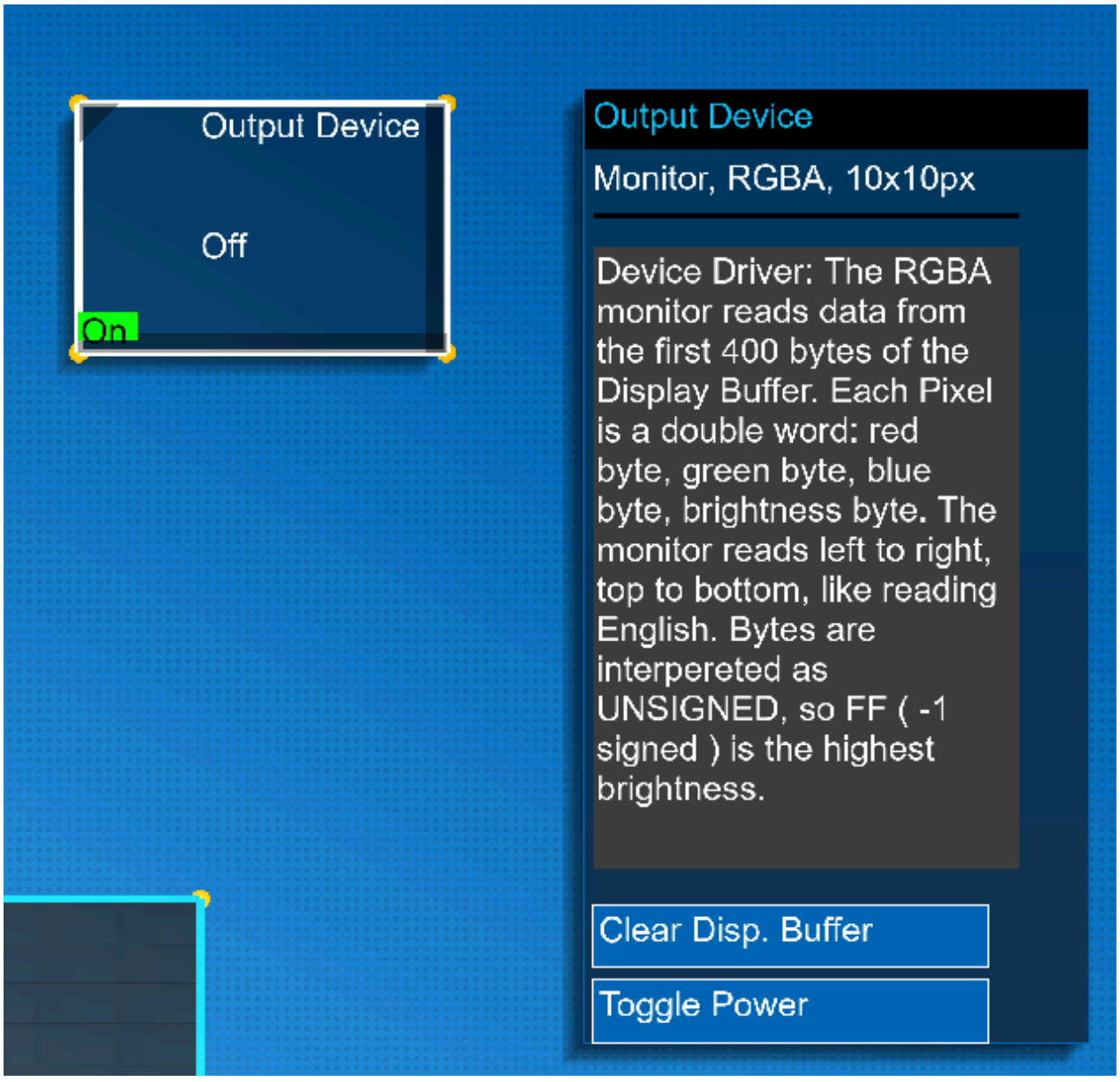


Figure 6.1: Output devices component and inspector

The Display Buffer

Output devices contain their own memory, which they use to display data. This device memory is modified by reading the **display buffer** section of the virtual 8086's RAM. The program that tells the output device how to read and interpret this data is called the **device driver**. The display buffer is designated memory for use from output devices, so to produce an output, the display buffer must be written to. The display buffer is the last section of memory and is composed of gray blocks.

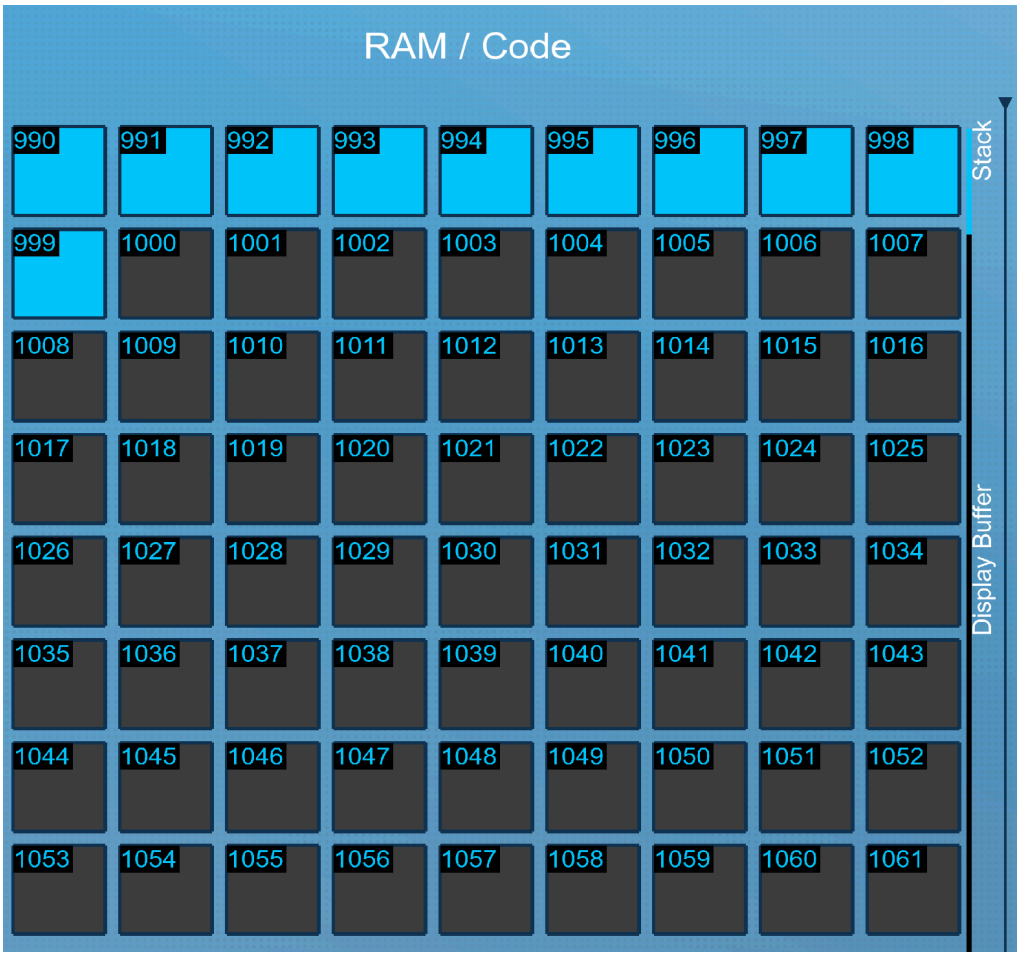


Figure 6.2: The display buffer

You can edit the display buffer by placing blocks into it, or with code.

The RGBA 10x10 Monitor

The first output device in Assemblox, pictured at the top of this chapter, is the RGBA 10x10 monitor. This device starts OFF, but when turned on you will begin to see a flashing yellow line between the monitor and the RAM. The flash indicates that the monitor is reading from RAM, and the frequency of this flash is the **refresh rate** of the monitor. The refresh rate of the monitor is independent of the clock speed of the CPU, and the monitor has its own power source.

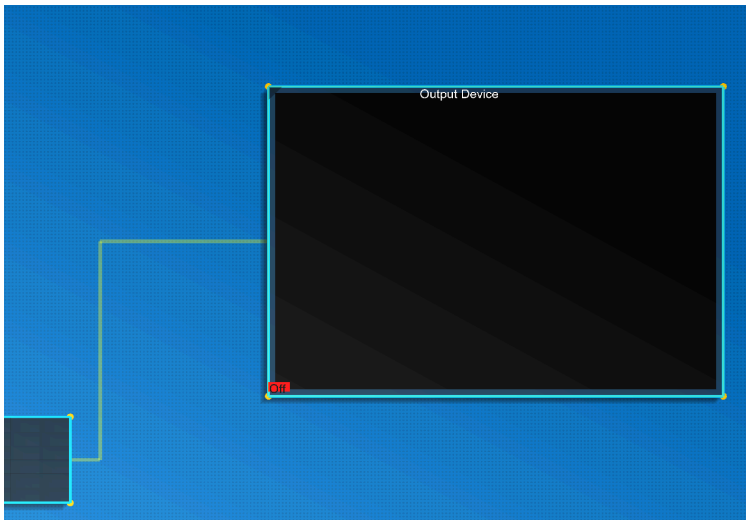


Figure 6.3: Monitor refreshing

Every time the monitor refreshes, it copies the contents of the display buffer into its own private memory. Therefore, even if the display buffer is changed, the visual content on the monitor will not update until it has copied that data.

The monitor driver, which can be seen by clicking on the monitor, provides information about how the monitor interprets its own memory. In the case of the RGBA Monitor, it reads 400 Bytes from the display buffer as 100 double-words, each 4 bytes long. Each double-word provides pixel data in the following format:

Byte 1	**Byte 2**	**Byte 3**	**Byte 4**
Red	Green	Blue	Brightness

This does not mean that all the blocks in the display buffer need to be 32-bit double-word blocks. You can also place 8-bit blocks, just remember that each empty space is interpreted as a zero.

So, if we want to modify the output device so that the top left pixel is completely green, we need to change the first 4 bytes to **00 FF 00 FF**. Because as signed integers, FF is -1, we can place -1 blocks into RAM in positions **1,001** and **1,003 (0x3E9 and 0x3EB)**. We can also place 255 and achieve the same result.

This is the result:

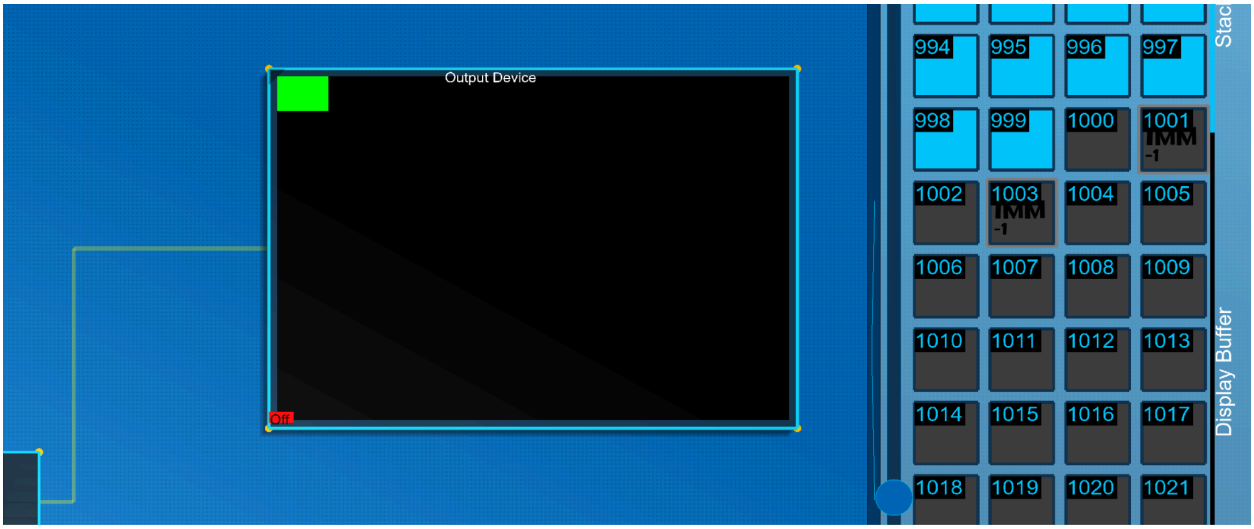


Figure 6.4: The Monitor with a pixel set to Green

Keep in mind that values are stored Little-Endian on the 8086, or **least significant byte first**. Values are also signed by default in Assemblox, so make sure to calculate the correct doubleword for each color you want before placing it into RAM, or by storing doublewords that represent colors in your code as 4 individual byte blocks for simplicity.

7: The Operating System

The virtual 8086 in assemblox comes with a pseudo-operating system called Assemblox Virtual OS. This operating system is very simple, its only job is to prevent the user from crashing the system.

The operating system is responsible for memory segmentation and is constantly watching specific registers, such as EIP, ESP, and EBP to ensure that no **segmentation faults** or **stack overflow** and **stack underflow** errors occur. When the OS detects that an invalid section of memory is attempted to be accessed, the OS automatically issues a **HALT** to the processor, ceasing all operations. For your convenience, the OS also writes a string to the System Pane explaining why the OS issued the HALT.

In future versions of Assemblox, you will be able to modify OS variables for increased customization.