A Simulation of a 6502 Based Computer System

Robert Moore, Candidate number: 1166, Reading School, Centre number: 51337

Table of Contents

[Analysis](#_Toc95849434)

[Introduction](#_Toc95849435)

[Computer Systems](#_Toc95849436)

[Motherboards](#_Toc95849437)

[Storage Devices](#_Toc95849438)

[Input / Output](#_Toc95849439)

[Processors](#_Toc95849440)

[The Fetch-Execute Cycle](#_Toc95849441)

[Programs](#_Toc95849442)

[System Architecture](#_Toc95849443)

[Similar Existing Systems](#_Toc95849444)

[AQA Assembly Language Simulator](#_Toc95849445)

[Other Peter Higginson Simulators](#_Toc95849446)

[Visual 6502](#_Toc95849447)

[Emulator 101](#_Toc95849448)

[Why the 6502?](#_Toc95849449)

[Technical Description](#_Toc95849450)

[The 65C02](#_Toc95849451)

[6502 vs. 65C02](#_Toc95849452)

[The Simulated System](#_Toc95849453)

[Arrangement](#_Toc95849454)

[Components](#_Toc95849455)

[Users](#_Toc95849456)

[Students](#_Toc95849457)

[Teachers](#_Toc95849458)

[Additional Requirements](#_Toc95849459)

[A Better Description of This Project](#_Toc95849460)

[Programming Language](#_Toc95849461)

[The Model](#_Toc95849462)

[Programs for the Simulator](#_Toc95849463)

[Level of Abstraction](#_Toc95849464)

[Object Oriented Model](#_Toc95849465)

[Acceptable Limitations](#_Toc95849466)

[Project Objectives](#_Toc95849467)

[1. Components](#_Toc95849468)

[2. Inter-Component](#_Toc95849469)

[3. Instruction Sets](#_Toc95849470)

[4. Hardware Components](#_Toc95849471)

[5. Simulation](#_Toc95849472)

[Design](#_Toc95849473)

[General Notes for Design](#_Toc95849474)

[Object Oriented Model](#_Toc95849475)

[Class Description Key](#_Toc95849476)

[Binary Electricity Model](#_Toc95849477)

[Inter-Component](#_Toc95849478)

[Node](#_Toc95849479)

[Connection](#_Toc95849480)

[Pin](#_Toc95849481)

[Wire](#_Toc95849482)

[Component](#_Toc95849483)

[Instruction Set](#_Toc95849484)

[Addressing Mode](#_Toc95849485)

[Operation](#_Toc95849486)

[Dynamic Instructions](#_Toc95849487)

[Instruction Set](#_Toc95849488)

[Assembler](#_Toc95849489)

[Simulated Hardware Components](#_Toc95849490)

[Processor](#_Toc95849491)

[Memory Devices](#_Toc95849492)

[Other Components](#_Toc95849493)

[Simulation](#_Toc95849494)

[Simulator](#_Toc95849495)

[User Interface](#_Toc95849496)

[Testing](#_Toc95849497)

[General](#_Toc95849498)

[Inter-Component](#_Toc95849499)

[Components](#_Toc95849500)

[Instruction Sets](#_Toc95849501)

[Component Subclasses](#_Toc95849502)

[Simulation](#_Toc95849503)

[Evaluation](#_Toc95849504)

[Table of References](#_Toc95849505)

[Code Dump](#_Toc95849506)

[File Arrangement](#_Toc95849507)

[general.py](#_Toc95849508)

[component.py](#_Toc95849509)

[instruction\_set.py](#_Toc95849510)

[assembler.py](#_Toc95849511)

[processor.py](#_Toc95849512)

[memory.py](#_Toc95849513)

[additional\_hardware.py](#_Toc95849514)

[instruction\_set\_65C02/addressing\_modes.py](#_Toc95849515)

[instruction\_set\_65C02/operations.py](#_Toc95849516)

[instruction\_set\_65C02/instructions.py](#_Toc95849517)

[instruction\_set\_65C02/opcode\_matrix.txt](#_Toc95849518)

[user\_interface.py](#_Toc95849519)

[simulator.py](#_Toc95849520)

[main.py](#_Toc95849521)

[testing.py](#_Toc95849522)

# Analysis

## Introduction

When learning the A level topic “Fundamentals of computer organisation and architecture” I struggled to visualise and understand how all of the components in a computer work as one. To understand how something works it is often useful to be able to interact and experiment with it. Diagrams and worked examples can show or explain different properties and features of computer systems but they are inflexible. In lessons we were also introduced to several assembly language simulators which, while good at helping with learning assembly language, didn’t show the physical processes very effectively.

The aim of this project is to create an interactive system for learning about computer systems with the main use being for learning about assembly language, machine code and CPU (central processing unit) architecture. The program simulates a 6502 based system and provides users with the ability to run or modify the system in order to demonstrate the purpose of each part of a system. The code itself can be read and edited to the same effect. As a result, the code and interface need to be clear and understandable. I will also be taking an object oriented approach for clarity and ease of editing. A user can easily swap out a part of the system for their own version as long as it complies to the same interface.

This project is for educational purposes aimed at A level computer science students. It is intended to be used for extending and consolidating understanding of computer systems. It could also be used in lessons to give students the opportunity to test themselves by putting what they have learned about computer systems, the operation of a CPU, and low-level programming into practice. As such it will conform to the model of computer systems that is taught at A level, specifically that of AQA since that is the model with which I am most familiar.

## Computer Systems

The following section describes the AQA model of computer systems. This section is here to provide an explanation of some parts of this model that I will refer to in other sections, so this section can be used to clarify the purpose and features of these.

### Motherboards

The components of a system are connected to the motherboard and data is transferred between them through the system bus. The system bus is the collective term for the data bus, address bus and control bus. Each bus is multiple wires or lines through which signals are transferred with the width of a bus referring to the number of lines it has.

The data bus is for the bidirectional transfer of actual data between the CPU and the other components. A 64-bit computer refers to a computer with a data bus width of 64 lines.

Address busses are used to communicate the memory location that the CPU is referring to, as such they are generally unidirectional. The width of an address bus determines the amount of physical memory that can be addressed.

Control signals are transferred between components through the control bus. These signals enable, facilitate, and control operations with each line being responsible for a different signal. Signals differ between system but tend to include: the system clock used for keeping the components synchronised; a read/write mode signal to determine whether the addressed location is being read or written to; interrupt request signals from I/O (Input/Output) devices that stop the CPU’s current operations such as a keyboard interrupt. Modern motherboards often have built in hardware for networking, sound, and video capabilities.

### Storage Devices

Computer systems need some form of storage for data. Small amounts of data can be stored within the processor (in registers and cache memory), but larger quantities of data are stored in separate components. Storage is separated into primary and secondary storage. Primary storage, sometimes called main memory, is devices, like ROM and RAM, which are connected to the CPU and occupy large sections of its addressable memory whereas secondary storage devices are addressed via I/O controllers (explained more later). For a processor to use data in secondary storage devices, it must first be copied to main memory.

#### Random Access Memory

Random access memory (RAM) is used to temporarily store instructions and data being processed by the CPU. RAM is volatile meaning that all data is lost when it turns off or loses power. Data pulled from secondary storage is usually stored, temporarily, in RAM. This data can be set or retrieved in any order.

#### Read Only Memory

Read only memory (ROM) is a common form of non-volatile primary storage. ROM usually contains start-up instructions that are executed immediately after the CPU receives power. These instructions are usually the BIOS (basic input/output system) or, more recently, UEFI (unified extensible firmware interface) includes details of connected hardware and the location of the operating system’s bootloader which would then be copied to RAM before starting up the computer. Most ROM is physically just as writable as RAM, however, the system in which it is used does not allow it to be written to either physically or logically.

#### Secondary Storage Devices

Secondary storage is non-volatile data storage that is connected to the system through an I/O controller. Types of secondary storage devices include hard disc drives, optical drives, solid state drives, and USB flash drives. Secondary storage devices are non-volatile meaning that data is retained while the device is off. Secondary storage is generally much slower than primary storage because it is not an immediate part of the system, but secondary storage devices can often store much larger amounts of data with some modern devices being able to store more than a terabyte. Due to the persistence of data stored in secondary storage, this is where applications and the operating system are generally stored.

### Input / Output

I/O (input/output) controllers provide an interface between the processor and other components within the system including storage devices, video displays and peripheral devices. I/O controllers consist of three main parts: an I/O port that connects the controller and the device; a set of registers to store data being exchanged through the controller; a physical interface connecting the controller to the system bus. The purpose of the I/O controller is to provide an interface to the processor so that the processor can address the device as if it were memory. This design avoids a processor needing to work with every conceivable peripheral and a peripheral needing to work with a variety of different processors.

### Processors

A processor is made up of billions of transistors which chain together to carry out instructions. A central processing unit (CPU) is the processor in a general-purpose computer. A CPU has a number of subcomponents namely a control unit (CU), an arithmetic and logic unit (ALU), and registers. Most modern processors also have onboard cache memory which stores recent and anticipated instructions that can be fetched faster than if they were stored elsewhere. Most processors and CPUs also have a separate, faster clock separate from the system clock which is usually achieved by multiplying the system clock’s frequency.

#### Registers

Registers are small units of memory, typically 1, 2, 4 or 8 bytes (depending on the processor). They store specific or general values required by the processor. A processor generally has the following registers:

##### Program Counter (PC)

Holds the memory address of the current instruction and is incremented once the instruction has been executed.

##### *Current Instruction Register (CIR)*

That stores the most recently fetched instruction whilst it is being executed.

##### Memory Address Register (MAR)

Which has the next address to be referenced by the processor. The MAR value is what is sent on the address bus.

##### Memory Buffer Register (MBR)

Sometimes called a memory data register, which stores the value received or to be sent on the data bus.

##### Status Register (SR)

Which is used to store information about the result of an instruction for example an overflow.

##### A set of general-purpose registers

That can store any type of data such as an accumulator register.

#### Control Unit

The CU is designed to decode the instructions retrieved from memory, translating them into a sequence of control signals to other components in the processor telling them to perform specific operations. The CU also contains the CPU clock.

#### Arithmetic and Logic Unit

The ALU performs arithmetic (mathematical) and logical (Boolean) operations. These include addition, Boolean comparisons like AND, and bitwise shifts. When these operations return a result, it is usually stored on a general-purpose register called the accumulator.

### The Fetch-Execute Cycle

The main purpose of a CPU is the fetch-execute cycle. The fetch-decode-execute cycle consists of three main steps: fetch, decode and execute. These steps are repeated for every instruction. In addition, there is a fourth step for interrupt checking that may change the current program being run.

#### Fetch

The memory address on the PC, which points to the next instruction, is copied to the MAR then passed on to the address bus while a read signal is sent on the control bus. The relevant memory device will return the value stored at this address, potentially after multiple cycles depending on the speed of the device. The returned value is transferred from the data bus into the MBR and then copied to the CIR so that the current instruction to be executed is ready for decoding. The PC is then likely incremented to point to the next instruction however, the point at which the PC is incremented may vary between CPUs.

#### Decode

The instruction held in the CIR is decoded by the CU into a set of control signals for other components in the system such as the ALU or RAM. These signals are to be sent out in sequence to instruct a series of steps that make up the execution of the instruction.

#### Execute

The appropriate sub-component carries out the operation it has been instructed to by control signals from the CU. The relevant sub-component could be the ALU if it were a mathematical or logical instruction or the CU if the instruction is to retrieve or store data in memory. The SR is used to store information about the result of operations such as whether it has been completed or if there was an overflow during a mathematical operation.

#### Interrupts

An interrupt request (IRQ) is a signal sent to the processor via the control bus. This signal comes from an I/O controller (hardware interrupts), an application (software interrupts), or the processor itself (exception interrupts). Software interrupts are usually handled by the operating system. Every device that can send an IRQ signal has an interrupt request number to identify the source of an interrupt. Interrupt requests are designed to speed up processing by removing polling: checking every device and application for whether they need attention.

IRQs are stored in an interrupt register so at the end of a processors execute step, it will check the interrupt register for any interrupts. Interrupts have a priority hierarchy so that even if there are multiple interrupts, the processor instantly knows which interrupt to respond to first. Each IRQ is associated with a bit in the interrupt register with the position in the register defining the priority.

The response to an interrupt is an interrupt service routine (ISR) which is a small program. To find the relevant ISR, there is a reserved space in memory for the interrupt descriptor table (IDT) containing the interrupt vector or starting memory address for each IRQ. When executing an ISR, the current volatile state of the processor’s registers must be preserved in memory so that it can return to that state after the ISR is completed. The location in memory where this is data is stored is the system stack (more later). The process of saving and loading the state of a processor is known as context-switching.

### Programs

Programs are groups of instructions executed in order to perform a larger task. The most well-known programming is high level programming however, in order for a processor to execute the steps involved in a program, it has to be converted into machine code.

#### Instruction Sets

An instruction set is the set of bit patterns that define the machine operations that the processor can perform. A processor can only understand its instruction set; however multiple processors can be designed to use the same instruction sets.

#### Machine Code

Machine code is binary instructions. These instructions consist of an opcode (operation code) and operands with the opcode being a reference to a specific process that the processor should execute and the operands being any additional data the processor would need to execute this specific instance of that instruction.

#### Addressing Modes

There are multiple ways for the processor to receive the operands for instructions and these are called addressing modes. Addressing modes allow for a single operation, for example “AND”, to refer to a variety of operations where different data is “AND”-ed. Addressing modes vary between instruction sets but there are a few main types:

##### Immediate addressing

The instruction itself contains the operand.

##### Direct addressing

The instruction points to a location in memory where the operand can be retrieved from.

##### Indirect addressing

Indirect addressing is similar to direct addressing except that the location in memory is itself a pointer to another address.

##### Implied addressing

Retrieving the operand, if there is one, is the same process for any instance of this instruction.

#### Assembly Language

Machine code is difficult for humans to read and write. Most people would need to constantly be looking up the opcodes for each instruction they want to program. Assembly language uses mnemonics to represent instructions making it much easier to understand. They also allow for addressing modes to be implicit based on the operands given in the line of assembly. Processors cannot understand assembly language though, so an assembler is used to translate it back into machine code.

Assembly language also has symbols and labels. Symbols are similar to variables in higher level languages in the fact that they represent a value using a meaningful identifier. If a specific piece of data is stored at a given memory address, it can be easier to use a symbol to refer to that memory address than writing out that address every time it is used. Symbols can also make programs more understandable for that same reason. Labels are a specific type of symbol that are references to points in the program. Labels are especially useful for jumping to different subroutines or parts of programs since they remove the users need to calculate the address that a specific instruction would be stored at.

#### Subroutines and the Stack

As in high-level languages, low-level programs can have subroutines (functions and procedures). The machine code for subroutines is stored separately to the instructions to call it. To be able to return to the call address, it has to be stored somewhere along with the local variables that are not passed as arguments. This data is pushed to the stack. A stack is a data type that follows a last-in-first-out (LIFO) principal meaning that when you add, or push, something onto the stack, that item will be the first thing that you retrieve, or pop / pull, from it. The stack was mentioned previously when discussing Interrupts.

### System Architecture

The arrangement of chips in a computer system is called its architecture. Two of the most common architecture types are Von Neumann and Harvard which differ predominantly in their storage of instructions and data in memory.

#### Von Neumann

Von Neumann architecture is better known as it is used in general-purpose computers. A single, shared main memory is used to store both instructions and data. This provides a lot more flexibility through dynamically allocated memory. However, the processor can only access and instruction or data at any given time (because they are stored in the same memory) requiring at least two fetches for most instructions.

#### Harvard

Harvard architecture is typically found in embedded systems or digital signal processors. There is distinct memory for program instructions and data which are accessed by separate busses. The lack of flexibility is suitable since embedded systems tend to have fixed firmware. The separation allows for greater efficiency such as having a smaller data memory compared to instruction memory and changing the bus widths to fit that. The speed gained from accessing instructions and data simultaneously is beneficial in real-time processing.

## Similar Existing Systems

In my research of similar systems, I did not find any existing systems that attempted to solve this problem. However, I can still use parts of their design and implementation to inform and improve upon my own plans.

### AQA Assembly Language Simulator

The AQA assembly language simulator[[1]](#footnote-2) by Peter Higginson is a similar project to that which I am creating. It is designed to fit with what is taught in AQA A level computer science just as my simulator is. This simulator was used in some of my lessons to aid in teaching about assembly language however it lacks details that would allow it to be used for other parts of the computer organisation and architecture topic. Additionally, the computer in this simulator shares very little in common with mine.

### Other Peter Higginson Simulators

The Little Man Computer[[2]](#footnote-3) and ARMLite[[3]](#footnote-4) simulators that Peter Higginson has also created have a similar interface to that of the AQA assembly language simulator. The focus of all of these simulators is the instruction set rather than the physical or logical flow of data. These simulators have a text editor on left in which assembly language programs can be written or pasted. On the right there is a grid displaying the values stored in memory which can be edited. Between the two is space to display inputs, outputs, and register values. There are also options to change the number base of the values displayed.

In my simulation, the focus is on the processor. If I were to use these same sections, I would likely have to split them into separate tabs in order to display a larger memory space and more detailed information about the state of the processor. The number base option is a good idea that allows for easier reading and comparison.

### Visual 6502

The visual 6502[[4]](#footnote-5) shows the flow of data between parts of a 6502 processor. My simulator aims to show the same thing but at a different level of abstraction. The visualiser shows individual wires in the processor however my simulator will only represent the larger units that these can be grouped into. The visualiser lacks an assembler and editing the memory to write in machine code is slow and not very user friendly. Overall, that makes the visual 6502 very different from this project however I can still learn from it. For example, the wires and other nodes are selectable providing their name which often implies their function. I could use a similar feature to allow selection of parts of the system to display more information about it and its current state.

### Emulator 101

Emulator 101[[5]](#footnote-6) is an arcade game emulator that originally aimed to emulate Space Invaders (which uses the 8080 processor) but has since been expanded to, among other things, emulate the 6502. Although this has many similarities to my project, the thing that I find most useful is the way it has been written up. Before this, I had not seen any examples of how to approach writing about a simulation or emulation of computer hardware. I think more detail could have been provided so the documentation of this project is more in depth.

## Why the 6502?

This project will be simulating the 6502 based computer system created by Ben Eater. This system is well documented on his website[[6]](#footnote-7) which includes a series of videos showing and explaining how he built the system as well as datasheets for all of the components used. As such, users have access to additional resources that can assist in their understanding of this project, computer system, and of computer systems in general. However, there are other contributing factors that make a 6502 based system a suitable choice.

The 6502[[7]](#footnote-8) (“sixty-five-oh-two” or “six-five-oh-two”) is an 8-bit microprocessor launched by MOS Technology (later CSG) in 1975. What makes the 6502 significant is that it was the least expensive microprocessor available by a significant amount. As a result, there was a rapid decrease in the cost of computers helping to spark the home computer revolution. As with all microprocessors, the 6502 is not limited to use in home computers and was also used in video game consoles. The 6502 family of processors continues to be widely used especially in embedded systems.

All of this means that the 6502 is one of the most significant individual processors to date making it an ideal processor for students to learn about in more detail. Additionally, being 8-bit, it is much simpler to understand than modern multi-core, hyperthreaded CPUs with pipelining and doesn’t require much understanding beyond the A level course to understand how it works.

#### Uses of the 6502

* Atari 8-bit family of computers
* Apple I & II
* Nintendo Entertainment System (NES)
* Commodore 64
* Family Computer (Famicom)
* BBC Micro

### Technical Description

The 6502 has a 16-bit address and an 8-bit data bus (because it’s an 8-bit processor). It uses little endian byte order meaning that the least significant bit of a word is stored at the lowest memory address. This endianness is only relevant for addresses which are stored over 2 bytes (16 bits). The internal logic runs at the same speed as the external clock which is typically 1 or 2 MHz (meaning 1 or 2 million cycles per second). This is a relatively low clock speed, but the 6502 is able to complete similar instructions in significantly fewer clock cycles (sometimes half as many) allowing it to compete with faster CPUs. This is partly because it uses a simple state machine implemented by combinational logic (meaning it does not rely on the clock) which, although used in many other designs was used to a greater extent in the 6502. As with most 8-bit microprocessors, there is limited pipelining (overlapping of fetching, decoding, and execution). The low clock speed also allowed for better compatibility with affordable peripherals like memory that typically have slower access times. The chip only accesses memory during certain parts of the clock cycle allowing other components of the system to access memory in those times when the 6502 is not using it. This is particularly useful for graphical processing where the graphical data stored in memory can be accessed, processed, and displayed by the video hardware without the processor trying to access or modify addresses in memory at the same time.

#### Registers

The 6502 has 6 main registers (see the Registers section in Computer Systems for more information) which is very few, especially by modern standards. Only one of these is a general-purpose register compared to 16 8-bit general purpose registers in the Z80 of the same era. To make up for this lack of registers, the 6502 has zero page addressing modes (the zero page being the first 256 addresses of memory). This requires only one byte to store this operand compared to two for a full address. This page will typically be part of RAM. These shorter instructions are also faster allowing these 256 addresses to be used like additional general-purpose registers.

##### Accumulator

The accumulator register (A) is an 8-bit general purpose register. It is the only general-purpose register.

##### Index registers

The 6502 has 3 index registers. The X and Y index registers are 8-bit registers which are used to modify operand addresses during execution (more later) hence why they are called index registers. The third index register is the stack pointer (S) which stores an 8-bit offset that is added to the start address of the stack (See Subroutines and the Stack, above) to find the top of the stack to push or pop/pull values. The stack’s addresses are hardwired to memory page $01 (meaning 1 in hex) which is the memory addresses from $0100 to $01FF (256 to 511).

##### Program counter

The program counter (PC) keeps track of the memory address of the next instruction or opcode to be processed. As such, the PC is 16-bits and is incremented after each instruction or opcode fetch.

##### Status Register

The processor status register (P) has 8 bits: NV-BDIZC. Most flags are changed based on the result of ALU (Arithmetic and Logic Unit) operations. The 8 flags are:

Negative (N) is set (made 1) if the result is negative based on two’s compliment.

Overflow (V) is a somewhat confusing flag[[8]](#footnote-9). The V flag is affected by addition and subtraction operations if they become too large in magnitude to store in 8-bits, the word length, using two’s compliment. This does not apply for the BIT instruction.

Bit 5 (-) is a reserved flag. This can be set and reset by the user as a custom status or mode bit.

The break flag (B) is used to force the BRK instruction to be executed next.

The decimal flag (D) is a mode select flag. When in decimal mode, the processor will interpret data as BCD (Binary Coded Decimal) numbers rather than two’s compliment signed integers.

Interrupt disable (I) is another mode select flag. When the I flag is reset, the processor will not respond to maskable interrupts.

The zero flag (Z) is, simply, whether the output of an arithmetic operation is zero.

Carry (C) is used to indicate whether an arithmetic operation needed to carry or borrow out of the most significant bit. This allows for multi-byte addition and subtraction.

#### Instruction Set

##### Addressing Modes:

###### Accumulator Addressing (A)

This form of addressing is represented with a one-byte instruction, implying an operation on the accumulator.

###### Immediate Addressing (#)

In immediate addressing, the operand is contained in the second byte of the instruction, with no further memory addressing required.

###### Absolute Addressing (a)

In absolute addressing, the second byte of the instruction specifies the eight low order bits of the effective address while the third byte specifies the eight high order bits. Thus, the absolute addressing mode allows access to the entire 65K bytes of addressable memory.

###### Zero Page Addressing (zp)

The zero page instructions allow for shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high address byte. Careful use of the zero page can result in significant increase in code efficiency.

###### Indexed Zero Page Addressing (zp,x)

###### Indexed Zero Page Addressing (zp,y)

(X, Y indexing) This form of addressing is used in conjunction with the index register and is referred to as "Zero Page, X" or "Zero Page, Y". The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally due to the "Zero Page" addressing nature of this mode, no carry is added to the high order 8 bits of memory and crossing of page boundaries does not occur.

###### Indexed Absolute Addressing (a,x)

###### Indexed Absolute Addressing (a,y)

(X, Y indexing) This form of addressing is used in conjunction with X and Y index register and is referred to as "Absolute, X", and "Absolute, Y". The effective address is formed by adding the contents of X or Y to the address contained in the second and third bytes of the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

###### Implied Addressing (i)

In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

###### Relative Addressing (r)

Relative addressing is used only with branch instructions and establishes a destination for the conditional branch. The second byte of the instruction becomes the operand which is an "Offset" added to the contents of the lower eight bits of the program counter when the counter is set at the next instruction. The range of the offset is -128 to +127 bytes from the next instruction.

###### Indexed Indirect Addressing ((zp,x))

In indexed indirect addressing (referred to as (Indirect,X)), the second byte of the instruction is added to the contents of the X index register, discarding the carry. The result of this addition points to a memory location on page zero whose contents is the low order eight bits of the effective address. The next memory location in page zero contains the high order eight bits of the effective address. Both memory locations specifying the high and low order bytes of the effective address must be in page zero.

###### Indirect Indexed Addressing ((zp),y)

In indirect indexed addressing (referred to as (Indirect),Y), the second byte of the instruction points to a memory location in page zero. The contents of this memory location is added to the contents of the Y index register, the result being the low order eight bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high order eight bits of the effective address.

###### Absolute Indirect Addressing ((a))

The second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits of that memory location is contained in the third byte of the instruction. The contents of the fully specified memory location is the low order byte of the effective address. The next memory location contains the high order byte of the effective address which is loaded into the sixteen bits of the program counter.

##### Operations

The 6502 has 56 operations, many of which can be used with multiple addressing modes. These can be seen below. (Ordered alphabetically)

|  |  |  |
| --- | --- | --- |
| ADC | **|** | Add Memory to Accumulator with Carry |
| AND | **|** | "AND" Memory with Accumulator |
| ASL | **|** | Shift left One Bit (Memory or Accumulator) |
| BCC | **|** | Branch on Carry Clear |
| BCS | **|** | Branch on Carry Set |
| BEQ | **|** | Branch on Result Zero |
| BIT | **|** | Test Bits in Memory with Accumulator |
| BMI | **|** | Branch on Result Minus |
| BNE | **|** | Branch on Result not Zero |
| BPL | **|** | Branch on Result Plus |
| BRK | **|** | Force Break |
| BVC | **|** | Branch on Overflow Clear |
| BVS | **|** | Branch on Overflow Set |
| CLC | **|** | Clear Carry Flag |
| CLD | **|** | Clear Decimal Mode |
| CLI | **|** | Clear Interrupt Disable Bit |
| CLV | **|** | Clear Overflow Flag |
| CMP | **|** | Compare Memory and Accumulator |
| CPX | **|** | Compare Memory and Index X |
| CPY | **|** | Compare Memory and Index Y |
| DEC | **|** | Decrement Memory by One |
| DEX | **|** | Decrement Index X by One |
| DEY | **|** | Decrement Index Y by One |
| EOR | **|** | "Exclusive-or" Memory with Accumulator |
| INC | **|** | Increment Memory by One |
| INX | **|** | Increment Index X by One |
| INY | **|** | Increment Index Y by One |
| JMP | **|** | Jump to New Location |
| JSR | **|** | Jump to New Location Saving Return Address |
| LDA | **|** | Load Accumulator with Memory |
| LDX | **|** | Load Index X with Memory |
| LDY | **|** | Load Index Y with Memory |
| LSR | **|** | Shift One Bit Right (Memory or Accumulator) |
| NOP | **|** | No Operation |
| ORA | **|** | "OR" Memory with Accumulator |

|  |  |  |
| --- | --- | --- |
| PHA | **|** | Push Accumulator on Stack |
| PHP | **|** | Push Processor Status on Stack |
| PLA | **|** | Pull Accumulator from Stack |
| PLP | **|** | Pull Processor Status from Stack |
| ROL | **|** | Rotate One Bit Left (Memory or Accumulator) |
| ROR | **|** | Rotate One Bit Right (Memory or Accumulator) |
| RTI | **|** | Return from Interrupt |
| RTS | **|** | Return from Subroutine |
| SBC | **|** | Subtract Memory from Accumulator with Borrow |
| SEC | **|** | Set Carry Flag |
| SED | **|** | Set Decimal Mode |
| SEI | **|** | Set Interrupt Disable Status |
| STA | **|** | Store Accumulator in Memory |
| STX | **|** | Store Index X in Memory |
| STY | **|** | Store Index Y in Memory |
| TAX | **|** | Transfer Accumulator to Index X |
| TAY | **|** | Transfer Accumulator to Index Y |
| TSX | **|** | Transfer Stack Pointer to Index X |
| TXA | **|** | Transfer Index X to Accumulator |
| TXS | **|** | Transfer Index X to Stack Pointer |
| TYA | **|** | Transfer Index Y to Accumulator |

##### Opcodes

The instructions, their addressing modes, and the opcode for each instruction addressing mode pair can be seen in the opcode matrix below. E.g. NOP has opcode $EA and uses the implied addressing mode.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |  |
| **0** | BRK  i | ORA  (zp,x) |  |  |  | ORA  zp | ASL  zp |  | PHP  i | ORA  # | ASL  A |  |  | ORA  a | ASL  a |  | **0** |
| **1** | BPL  r | ORA  (zp),y |  |  |  | ORA  zp,x | ASL  zp,x |  | CLC  i | ORA  a,y |  |  |  | ORA  a,x | ASL  a,x |  | **1** |
| **2** | JSR  a | AND  (zp,x) |  |  | BIT  zp | AND  zp | ROL  zp |  | PLP  i | AND  # | ROL  A |  | BIT  a | AND  a | ROL  a |  | **2** |
| **3** | BMI  r | AND  (zp),y |  |  |  | AND zp,x | ROL zp,x |  | SEC  i | AND  a,y |  |  |  | AND  a,x | ROL  a,x |  | **3** |
| **4** | RTI  i | EOR  (zp,x) |  |  |  | EOR  zp | LSR  zp |  | PHA  i | EOR  # | LSR  A |  | JMP  a | EOR  a | LSR  a |  | **4** |
| **5** | BVC  r | EOR  (zp),y |  |  |  | EOR zp,x | LSR zp,x |  | CLI  i | EOR  a,y |  |  |  | EOR  a,x | LSR  a,x |  | **5** |
| **6** | RTS  i | ADC  (zp,x) |  |  |  | ADC  zp | ROR  zp |  | PLA  i | ADC  # | ROR  A |  | JMP  (a) | ADC  a | ROR  a |  | **6** |
| **7** | BVS  r | ADC  (zp),y |  |  |  | ADC zp,x | ROR zp,x |  | SEI  i | ADC  a,y |  |  |  | ADC  a,x | ROR  a,x |  | **7** |
| **8** |  | STA  (zp,x) |  |  | STY  zp | STA  zp | STX  zp |  | DEY  i |  | TXA  i |  | STY  a | STA  a | STX  a |  | **8** |
| **9** | BCC  r | STA  (zp),y |  |  | STY  zp,x | STA zp,x | STX zp,y |  | TYA  i | STA  a,y | TXS  i |  |  | STA  a,x |  |  | **9** |
| **A** | LDY  # | LDA  (zp,x) | LDX  # |  | LDY  zp | LDA  zp | LDX  zp |  | TAY  i | LDA  # | TAX  i |  | LDY  a | LDA  a | LDX  a |  | **A** |
| **B** | BCS  r | LDA  (zp),y |  |  | LDY  zp,x | LDA zp,x | LDX zp,y |  | CLV  i | LDA  a,y | TSX  i |  | LDY  a,x | LDA  a,x | LDX  a,y |  | **B** |
| **C** | CPY  # | CMP  (zp,x) |  |  | CPY  zp | CMP  zp | DEC  zp |  | INY  i | CMP  # | DEX  i |  | CPY  a | CMP  a | DEC  a |  | **C** |
| **D** | BNE  r | CMP  (zp),y |  |  |  | CMP zp,x | DEC zp,x |  | CLD  i | CMP  a,y |  |  |  | CMP  a,x | DEC  a,x |  | **D** |
| **E** | CPX  # | SBC  (zp,x) |  |  | CPX  zp | SBC  zp | INC  zp |  | INX  i | SBC  # | NOP  i |  | CPX  a | SBC  a | INC  a |  | **E** |
| **F** | BEQ  r | SBC  (zp),y |  |  |  | SBC zp,x | INC zp,x |  | SED  i | SBC  a,y |  |  |  | SBC  a,x | INC  a,x |  | **F** |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |  |

### The 65C02

The computer system being modelled in this simulation is not a 6502 microprocessor but rather Western Design Centre’s (WDC) 65C02[[9]](#footnote-10) microprocessor launched in 1981. This is partly because the original 6502 had several bugs and quirks which would be difficult to model accurately. The 6502 is nMOS-based whereas the 65C02 is an enhanced CMOS version. The main differences are that the 65C02 has fewer problems, more instructions, and a reduced power usage (when run at the same speed). This reduced power usage means it is better suited to use in portable computers and microcontroller systems than the 6502. It has also been used in embedded systems and home computers. Many 65C02s can also be run significantly faster than the 6502 with some of them having a fully static core allowing them to be run slower as well.

#### Uses of the 65C02

* BBC Master
* Atari Lynx
* Apple IIe (enhanced) and IIc (portable)
* Many replicas of 6502 systems (e.g. Replica 1)
* Many dedicated systems such as chess computers

### 6502 vs. 65C02

As previously mentioned, the 65C02 is a low-powered 6502, with additional instructions and addressing modes, that has fixed several bugs.

#### Opcodes

Unchanged elements are shown in grey and italicised in this table.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |  |
| **0** | BRK  s | ORA  (zp,x) |  |  | TSB  zp | ORA  zp | ASL  zp | RMB0  zp | PHP  s | ORA  # | ASL  A |  | TSB  a | ORA  a | ASL  a | BBR0  zp, r | **0** |
| **1** | BPL  r | ORA  (zp),y | ORA  (zp) |  | TRB  zp | ORA  zp,x | ASL  zp,x | RMB1  zp | CLC  i | ORA  a,y | INC  A |  | TRB  a | ORA  a,x | ASL  a,x | BBR1  zp, r | **1** |
| **2** | JSR  a | AND  (zp,x) |  |  | BIT  zp | AND  zp | ROL  zp | RMB2  zp | PLP  s | AND  # | ROL  A |  | BIT  a | AND  a | ROL  a | BBR2  zp, r | **2** |
| **3** | BMI  r | AND  (zp),y | AND  (zp) |  | BIT  zp,x | AND zp,x | ROL zp,x | RMB3  zp | SEC  i | AND  a,y | DEC  A |  | BIT  a,x | AND  a,x | ROL  a,x | BBR3  zp, r | **3** |
| **4** | RTI  s | EOR  (zp,x) |  |  |  | EOR  zp | LSR  zp | RMB4  zp | PHA  s | EOR  # | LSR  A |  | JMP  a | EOR  a | LSR  a | BBR4  zp, r | **4** |
| **5** | BVC  r | EOR  (zp),y | EOR  (zp) |  |  | EOR zp,x | LSR zp,x | RMB5  zp | CLI  i | EOR  a,y | PHY  s |  |  | EOR  a,x | LSR  a,x | BBR5  zp, r | **5** |
| **6** | RTS  s | ADC  (zp,x) |  |  | STZ  zp | ADC  zp | ROR  zp | RMB6  zp | PLA  s | ADC  # | ROR  A |  | JMP  (a) | ADC  a | ROR  a | BBR6  zp, r | **6** |
| **7** | BVS  r | ADC  (zp),y | ADC  (zp) |  | STZ  zp,x | ADC zp,x | ROR zp,x | RMB7  zp | SEI  i | ADC  a,y | PLY  s |  | JMP  (a,x) | ADC  a,x | ROR  a,x | BBR7  zp, r | **7** |
| **8** | BRA  r | STA  (zp,x) |  |  | STY  zp | STA  zp | STX  zp | SMB0  zp | DEY  i | BIT  # | TXA  i |  | STY  a | STA  a | STX  a | BBS0  zp, r | **8** |
| **9** | BCC  r | STA  (zp),y | STA  (zp) |  | STY  zp,x | STA zp,x | STX zp,y | SMB1  zp | TYA  i | STA  a,y | TXS  i |  | STZ  a | STA  a,x | STZ  a,x | BBS1  zp, r | **9** |
| **A** | LDY  # | LDA  (zp,x) | LDX  # |  | LDY  zp | LDA  zp | LDX  zp | SMB2  zp | TAY  i | LDA  # | TAX  i |  | LDY  a | LDA  a | LDX  a | BBS2  zp, r | **A** |
| **B** | BCS  r | LDA  (zp),y | LDA  (zp) |  | LDY  zp,x | LDA zp,x | LDX zp,y | SMB3  zp | CLV  i | LDA  a,y | TSX  i |  | LDY  a,x | LDA  a,x | LDX  a,y | BBS3  zp, r | **B** |
| **C** | CPY  # | CMP  (zp,x) |  |  | CPY  zp | CMP  zp | DEC  zp | SMB4  zp | INY  i | CMP  # | DEX  i | WAI  i | CPY  a | CMP  a | DEC  a | BBS4  zp, r | **C** |
| **D** | BNE  r | CMP  (zp),y | CMP  (zp) |  |  | CMP zp,x | DEC zp,x | SMB5  zp | CLD  i | CMP  a,y | PHX  s | STP  i |  | CMP  a,x | DEC  a,x | BBS5  zp, r | **D** |
| **E** | CPX  # | SBC  (zp,x) |  |  | CPX  zp | SBC  zp | INC  zp | SMB6  zp | INX  i | SBC  # | NOP  i |  | CPX  a | SBC  a | INC  a | BBS6  zp, r | **E** |
| **F** | BEQ  r | SBC  (zp),y | SBC  (zp) |  |  | SBC zp,x | INC zp,x | SMB7  zp | SED  i | SBC  a,y | PLX  s |  |  | SBC  a,x | INC  a,x | BBS7  zp, r | **F** |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |  |

#### New Addressing Modes

##### Zero Page Indirect Addressing ((zp))

The 6502 has three indirect addressing modes: Indexed Indirect Addressing ((zp,x)), Indirect Indexed Addressing ((zp),y), and Absolute Indirect Addressing ((a)). For the zero page indirect addressing modes, there is no non-indexed option, so the 65C02 provided one.

##### Absolute Indexed Indirect Addressing ((a,x))

Absolute indexed indirect addressing adds the option for indexes on top of Absolute Indirect Addressing ((a)). This is particularly useful for branch tables.

##### Stack Addressing (s)

10 of the instructions with implied addressing modes have been changed to the new stack addressing mode. This is less of a new addressing mode but rather a more specific classification for a collection of instructions that use the stack (which is implied by the instructions). Instructions using the stack addressing mode are all of the pull and push (from the stack) instructions.

#### Modified Operations / New Instructions

##### Zero Page Indirect Addressing Mode

Zero Page Indirect Addressing ((zp)) has been added to the arithmetic instructions: ORA, AND, EOR, ADC, STA, LDA, CMP, and SBC.

##### Absolute Indexed Indirect Addressing Mode

JMP can be used with the new addressing mode, Absolute Indexed Indirect Addressing ((a,x)).

##### Stack Addressing Mode

As mentioned above, several instructions with implied (i) addressing mode have been changed to Stack Addressing (s). These instructions are: BRK, RTI, RTS, PHP, PLP, PHA, and PLA as well as 4 new instructions (see Modified below).

##### Accumulator Addressing Mode

INC and DEC can now be used with Accumulator Addressing (A).

##### Bit Test (BIT) Instruction

The BIT instruction has been expanded to now be able to be used with Indexed Zero Page Addressing (zp,x), Indexed Absolute Addressing (a,x), or Immediate Addressing (#).

#### New Operations

##### Branch Always (BRA)

BRA acts similarly to JMP but uses Relative Addressing (r) meaning that it uses one byte fewer to store. This also makes programs that uses BRA as opposed to JMP, relocatable.

##### Store Zero (STZ)

Rather than setting a registers value to zero and storing that in memory, a zero can be immediately stored. This makes the task of storing a zero faster and it takes up less space in memory.

##### Push and Pull Index Registers (PHY, PLY, PHX, PLX)

These four Stack Addressing (s) instructions were added to allow the X and Y index registers to be pushed to and pulled from the stack directly in the same way that the accumulator register was in the 6502. With PH being the push instructions and PL being the pull instructions.

##### Processor State Instructions

Stop processor (STP)

The processor is effectively shut down until a hardware reset occurs. This reduces power usage.

Wait for interrupt (WAI)

The processor is in a low power state until it receives a hardware interrupt (IRQ, NMI, or RESET). Once in this state, it can respond without any delay.

##### Bit Manipulation Instructions

SMB & RMB (0 – 7)

These bit manipulation instructions allow specific bits of a value in zero page location to be set (SMB) or reset (RMB). Therefore, it should be no surprise that these instructions are set memory bit and reset memory bit, respectively. The bit that is being affected is part of the instruction itself rather than being a second operand as may have been expected in a different instruction set. E.g. SMB3 $12 would make bit 3 a 1 (set) so if address $0012 originally stored 000000002 it would now be 000010002.

TSB & TRB

“Test and set bits” (TSB) and “test and reset bits” (TRB) can use either Absolute Addressing (a) or Zero Page Addressing (zp). They perform a BIT operation on this address then affect its bits based on the contents of the accumulator. The affect is such that if a bit is set (1) in the accumulator the same bit at the specified address will be set or reset (based on which instruction is being executed) while bits where the accumulator bit is reset remain unaffected. This is logically the same as for TSB or for TRB where **M** is the byte at the addressed memory location and **A** is the value stored in the accumulator register.

BBS & BBR (0 – 7)

Branch bit set / reset takes two operands, with an implicit third being the bit (0 – 7) stated within the instruction. The first operand is a zero page address, and the second is a relative address. If the bit specified within the instruction of the value at the zero page memory address is set or reset (instruction dependant) then the processor will branch to the relative address as given by the second operand.

## The Simulated System

As was previously stated, the computer system being simulated in this project is a well-documented system created by Ben Eater. However, the full / final documented system will not be simulated. The system simulation is given mainly as a starting point upon which the user can build. As a result, only part of the system will be simulated. This section contains the processor, ROM, RAM, and clock as its main components. This allows the user to create and run programs that only use memory devices and their addresses without including the greater complexity of I/O devices and their addressing.

The ROM (Read Only Memory), RAM (Random Access Memory), and clock components all have very similar logical functionality as their respective generalised versions described in Computer Systems (above). This is primarily due to the limited range of functionality possible for a device to still be considered to be of these specific types.

The system must also include NAND gates, a power supply, button, and resistors and capacitors. Excluding the NAND gates, these components have much simpler functions and few inputs and outputs. Resultantly, these have not been given as detailed descriptions as other components.

### Arrangement

A picture containing diagram

Description automatically generated This is an adaptation of Ben Eater’s system schematic that shows only the components of the system that are to be simulated and not the rest of the components which predominantly pertain to I/O.

### Components

#### Clock

The clock[[10]](#footnote-11) (a 1 MHz crystal oscillator can) has four I/O pins: N/C, GND, Output, VCC.

Power and ground pins (labelled VCC and GND respectively) are fundamental to the operation of any chip. As such, some version of power and ground pins are found in all logic components.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| VCC | –– | 14 | 1 | –– | N/C |
|  |  |  |  |  |  |
| Output | –– | 8 | 7 | –– | GND |

The N/C pin is a no connect pin meaning that is should not be connected. No connect pins generally exist for structural purposes like symmetry.

The output pin is the clock signal created within this component. This is a 1 MHz oscillator so the output pin should change between high and low 2 million times per second (with high to low and low to high transitions both being counted).

#### Processor

The W65C02S 40 pin PDIP is the specific processor model used in this system.

A0 – A15 address bus

BE bus enable

D0 – D7 data bus

IRQB interrupt request (bar)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| VPB | –– | 1 | 40 | –– | RESB |
| RDY | –– | 2 | 39 | –– | PHI2O |
| PHI1O | –– | 3 | 38 | –– | SOB |
| IRQB | –– | 4 | 37 | –– | PHI2 |
| MLB | –– | 5 | 36 | –– | BE |
| NMIB | –– | 6 | 35 | –– | NC |
| SYNC | –– | 7 | 34 | –– | RWB |
| VDD | –– | 8 | 33 | –– | D0 |
| A0 | –– | 9 | 32 | –– | D1 |
| A1 | –– | 10 | 31 | –– | D2 |
| A2 | –– | 11 | 30 | –– | D3 |
| A3 | –– | 12 | 29 | –– | D4 |
| A4 | –– | 13 | 28 | –– | D5 |
| A5 | –– | 14 | 27 | –– | D6 |
| A6 | –– | 15 | 26 | –– | D7 |
| A7 | –– | 16 | 25 | –– | A15 |
| A8 | –– | 17 | 24 | –– | A14 |
| A9 | –– | 18 | 23 | –– | A13 |
| A10 | –– | 19 | 22 | –– | A12 |
| A11 | –– | 20 | 21 | –– | VSS |

MLB memory lock (bar)

NC no connection

NMIB non-maskable interrupt (bar)

PHI1O phase 1 out Clock

PHI2 phase 2 in clock

PHI2O phase 2 out Clock

RDY ready

RESB reset (bar)

RWB read/write (bar)

SOB set overflow (bar)

SYNC synchronise

VDD positive power supply

VPB vector pull (bar)

VSS internal logic ground

#### NAND Gate

Ben Eater used a quad 2-input NAND gate[[11]](#footnote-12) to process some of the control signals. The pins of the NAND gate are divided into its 4 (hence quad) 2-input sub-gates numbered 1 to 4.

VCC and GND are the power and ground pins respectively.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A1 | –– | 1 | 14 | –– | VCC |
| B1 | –– | 2 | 13 | –– | B4 |
| Y1 | –– | 3 | 12 | –– | A4 |
| A2 | –– | 4 | 11 | –– | Y4 |
| B2 | –– | 5 | 10 | –– | B3 |
| Y2 | –– | 6 | 9 | –– | A3 |
| GND | –– | 7 | 8 | –– | Y3 |

A and B pins numbered 1 – 4, make up the inputs to the sub-gates of the same number.

Y pins are the outputs of the sub-gates.

#### ROM and RAM

The ROM (28C256 256K Parallel EEPROM[[12]](#footnote-13)) and RAM (62256 256K SRAM[[13]](#footnote-14)) both have similar pinouts. Their 28 pins as follows:

A0 – A14 are address input pins

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A14 | –– | 1 | 28 | –– | VCC or VCC |
| A12 | –– | 2 | 27 | –– |  |
| A7 | –– | 3 | 26 | –– | A13 |
| A6 | –– | 4 | 25 | –– | A8 |
| A5 | –– | 5 | 24 | –– | A9 |
| A4 | –– | 6 | 23 | –– | A11 |
| A3 | –– | 7 | 22 | –– |  |
| A2 | –– | 8 | 21 | –– | A10 |
| A1 | –– | 9 | 20 | –– | or |
| A0 | –– | 10 | 19 | –– | I/O7 |
| I/O0 | –– | 11 | 18 | –– | I/O6 |
| I/O1 | –– | 12 | 17 | –– | I/O5 |
| I/O2 | –– | 13 | 16 | –– | I/O4 |
| GND or VSS | –– | 14 | 15 | –– | I/O3 |

I/O0 – I/O7 are data input/output pins

\* (chip enable for ROM) or (chip select for RAM) determines if the chip is active or in a standby state.

\* Note: the bar above CE and CS means that the chip is enabled when the pin is not set.

(write enable) determines whether the chip is being read from or written to.

(output enable) disables reading and affects writing.

Power supply and ground are called VCC and GND for ROM or VCC and VSS for RAM.

## Users

When creating a program, the end users must be considered. By analysing the wants and needs of the end users, firm objectives can be created that define the requirements of the system. I have identified two key user-groups for this project: students and teachers.

### Students

The simulator is designed for educational purposes relating to an A level computer science topic so the primary users will be A level computer science students. Students will be able to use this system to further their understanding of computer systems by use of a more interactive simulation (more interactive than the Similar Existing Systems). As such, everything should be programmed in a way that is easily understandable and, as a result, more editable. To achieve this, a highly modular, object oriented approach will be taken with meaningful identifiers used. Additionally, the simulation should make it easy to import and export machine code or assembly language programs to make them more easily shareable between students. All of this must be done in combination with a simulation that shows the flow of data through a computer system which is primary functionality of this project.

Students would be wanting to use this system both in school and at home. As a result, not only do the programs created within the system need to be sharable, but the entire project must itself be portable. It must be possible for students to use this system on different devices meaning that it cannot be dependent on specific characteristics of the device it is developed on, such as the operating system. Thankfully, this is unlikely to become a problem since there are interpreters for many high-level languages available to install onto any modern computer allowing the code of the project to be edited and run on a huge variety of devices.

#### Student Survey

I asked several computer science students to answer these questions.

##### Questions

Q1: What programming languages do you use? (Or know how to use)

Q2: Specify "other" (If selected)

Q3: How much experience do you have with programming?

Q4: What did you find easy/hard in the computer organisation and architecture topic?

Q5: Do you think a simulation of a computer system would help you to understand computer systems better?

##### Responses

|  |  |  |  |
| --- | --- | --- | --- |
| **Q1** | **Q3** | **Q4** | **Q5** |
| Python | Many years, few projects | Addressing modes | Yes |
| Python, C#, Java, C++, JavaScript, SQL, Lua | Many years, many projects | I am a bit confused about how images are displayed using stored bits; I assume that it is specific to the implementation | Yes |
| Python, C#, Java, C++ | Few years,  few projects | I found it difficult to understand how the different parts of computer systems communicate in order to work together | Yes |
| Python, JavaScript | Few years, many projects | Everything was pretty interesting so I spent quite some time understanding the concepts behind them - I'd say the topic had quite a few hard areas to grasp the fine details at first but nothing was extremely difficult. Understanding exactly how the FDE cycle works was extremely interesting and it'd be great if that was shown in detail in your project. Seeing pieces of data (or just pointers/memory references) passed between registers and the whole processor would be pretty cool and a great way to explain the process to others. Something that was a bit complicated was the exact impacts of changing components on processor performance. So for example, before the topic I knew reducing the word length would make the processor slower but understanding the specific reason behind the slow performance was enlightening (i.e., because the number of operations required to do one task would be higher / less data could be processed at once). Learning the same with clock speed and number of cores was awesome too. Having the ability to perhaps alter either of these and seeing what exactly changes in a simulation would be perfect, or even just somewhere it saying that the processor is slower now because of a specific reason. I know you didn't ask for such a big answer but hey, hope it helps. If you want more detail on what I said let me know. | Yes |
| C# | Many years, many projects | I found the external device section pretty easy. Learning about the buses and internal components was harder for me. Also, some of the specifics of Von Neumann vs Harvard was complicated. | Yes |
| Python | Few years,  few projects | Fetch decode execute cycle | Yes |
| Python, C#, JavaScript, C | Many years, many projects | Easy: Assembly operations; factors affecting performance; input/output devices Hard: Interrupts, addressing modes, fetch execute cycle details | Yes |
| Python, C#, Java, C++, JavaScript, HTML, CSS, React, JSX, VB.NET, Node.JS, assembly language\*, C, Bash, jQuery | Many years, many projects | Everything was easy | No |

\*Presumably AQA assembly language

##### Conclusion

These responses show that many students found parts of the computer organisation and architecture topic challenging especially the fetch-execute cycle, addressing modes, and possibly what assembly languages actually are. Additionally, most students believed that a more interactive/responsive simulation of a computer system would be helpful in learning about and understanding computer systems.

In terms of programming languages, Python was known by almost all students asked. This is likely because Python is very flexible and, as a result, is often taught in lower years of secondary school. As can be expected, students who have been programming for longer have used more programming languages.

### Teachers

While students will be the main users of this system, their teachers could also find use for it in running demonstrations for the class or planning a set of tasks for their students to complete using this system. To assist in this secondary usage, I will include functionality to store and share systems more effectively than just having to share entire, potentially multi-file, programs. This will allow for the distribution of specifically designed exemplar systems to students for use in lessons.

## Additional Requirements

### A Better Description of This Project

Despite describing this project as being, simply, a simulation of a computer system, it is intended to be much more than that. As I have said, part of the purpose of this system is that parts of it will be modified, appended, or replaced by its users. As such, the most important parts of this project are the things that are least likely to be changed. This includes the interfaces of objects within the object oriented model, and the method by which data is exchanged between components of the simulated system.

As a result, the final product will be more of a framework in which “any” computer system can be implemented. The system will, almost certainly, have limits to what can be implemented without needing to change the fundamental parts of the system listed above, however it should have as broad a range of compatible computer systems as possible to make it as usable as possible.

Nevertheless, when the final program is run, the user can expect to be greeted by The Simulated System (detailed above) created within this framework.

### Programming Language

The final program is intended to be easily understandable and editable, so it is important that it is in a suitable programming language to create an effective and useful project. I have experience in four high level programming languages (Python, Java, C#, and C++) so using one of these would make the final program more understandable to other users of that language. However, it is more important that the language is one that a large number of potential users are familiar with. Based on 2020 surveys[[14]](#footnote-15), these four languages are very common amongst developers with Python as the most popular of these. Being commonly used by developers means that they are likely used by students in similar proportions. JavaScript, a programming language that I am aware of but have never used, was also ranked highly as a used and wanted language so I will consider it as a fifth option.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Language** | **Experience**  ***(1 = most experience)*** | **Online Surveys**  ***(1 = most used)*** | **Student Survey**  ***(1 = most used)*** | **Speed of Execution****[[15]](#footnote-16)**  **(1 = fastest\*)** | **Compiled / Interpreted** | **Typing** |
| Python | 1 | 1 (1, 2) | 1 (7) | 5 (4, 4) | Interpreted | Dynamic |
| C# | 2 | 3 (2, 4) | 2 (5) | 2 | Compiled | Strong (explicit) |
| C++ | 3 | 5 (5, 5) | 4 (3) | 1 (1, 1) | Compiled | Strong (explicit) |
| Java | 4 | 4 (4, 3) | 4 (3) | 3 (3, 2) | Compiled | Strong (explicit) |
| JavaScript | 5 | 2 (3, 1) | 3 (4) | 3 (2, 3) | Interpreted | Dynamic |

\*Speed is a comparison of average relative speeds over many similar algorithms. As such, the speed ranking may not be accurate for all algorithms. However, it is still a useful measure of probable execution speeds.

As can be expected, each programming language has different strengths and weaknesses. As a result, compromise is necessary. I consider user familiarity to be the most important factor, so I have chosen to use Python. While Python is the slowest language amongst these options, it is the faster interpreted language. Interpreted languages are slower to run but do not require time to compile before execution therefore I would consider an interpreted language to be better suited to a project where the code is intended to be edited frequently. The typing is mostly for understandability. Python and JavaScript are dynamically typed meaning that a variable can change type and the interpreter will determine a variable type. However, I know that python allows type hints which show the expected type of a parameter. Type hints do not affect the execution but provide clarity similar to that of the function definitions of more explicitly typed languages.

## The Model

### Programs for the Simulator

For users to be able to run programs on their systems, it would be useful to have some form of assembler within the project. This is not a necessary part of this project as for whatever processor is used, its assembler would be slightly different requiring the user to put in significant work to implement an assembler while there are likely multiple assemblers for that processor available on the internet. As such, an assembler implemented for this project will be simplistic and very modular. The simplicity is to aid in the modularity and reusability in many other user-defined assemblers. An assembler included within the project is there more for convenience than functionality.

### Level of Abstraction

A suitable level of abstraction should be used to match the purpose of the project. There is no need to simulate every transistor in the computer but instead all that is needed is the set of logical processes and interactions within and between components. Here, the physical, electrical processes have been abstracted leaving clearer more understandable processes. This is well suited to the purposes of this simulator.

### Object Oriented Model

An object oriented approach is to be taken to provide more meaningful/understandable code since one of the goals of this project is to have the user be able to modify and expand upon the system. Only some of the objects / classes are described here in very little detail.

#### Classes

##### Component

One-to-one representations on physical hardware components and their logical functionality can be created in classes. For example, a RAM (Random Access Memory) class could be created that, when initialised, stores an array (or any ordered collection such as a list) of binary data where each index is an address and allows them to be set or retrieved. These are the logical processes of a RAM chip. Similar abstraction can be applied to other hardware allowing for a collection of objects to represent all of the chips or devices in a computer system.

All hardware components would require an inter-component interface to exchange data during the simulation. This would need to be done by storing the state of their I/O pins or the wires connecting them and allowing this state to be read by all components that are connected to that wire or pin. This results in a many-to-many bidirectional relationship that would need to somehow be resolved down into many one-to-many one-directional relationships. Additionally, to aid in simulations, components could include some mechanism to save and load their state.

##### Classes in the Instruction Set

Classes should also be used to modularise instruction sets. These would consist of a class for operations, for addressing modes, and a class to contain the operation and addressing mode instances (and the relationships between them) that make up an instruction set. This instruction set would have methods for executing instruction-addressing-mode pairs (as given by an opcode). The execution methods would need to make calls to methods of the addressing mode in order to fetch the operands either from registers or from memory and call the instruction using those operands to produce the result. The instruction set will also either contain or be closely related to an assembler object or static class depending on whether or not it used attributes that can change during runtime.

An object oriented approach to implementing an instruction set would make it much easier to implement many instruction sets into the simulator. This makes it much easier for users to implement and use the relative instruction set for a processor that they are using. This is because the interface has already been defined and some of the methods of other instruction sets may be reusable for that new instruction set.

#### Abstract Classes

Abstract classes cannot be instantiated. Instead, they provide an interface, attributes, and methods for their subclasses to build upon.

##### Component

As mentioned above, simulated hardware components would all have similar interfaces. As such, they can inherit this shared interface and many attributes or methods from an abstract super class. This includes the pin interface and the save/load state. However, the state methods would likely need to be abstract meaning that any subclasses must define the process for these methods before they can be instantiated. This is because many components would have different attributes that contribute to their state. For example, the Processor has a set of registers that each have a value that can change during simulation.

##### Abstract Subclasses

Groups of components, like memory devices, could also be grouped together under an abstract subclass. This would allow the storage and manipulation of data in memory to be defined only once in a superclass.

### Acceptable Limitations

#### Instruction Set

The 65C02 instruction set has more than 200 instructions. It is acceptable for only some of these to be implemented. The implementation must still allow all of these instructions to be created. This is not a challenge since the functions for the execution of these instructions are very simple. The decision to only have some instructions implemented is a matter of time management.

#### Assembler

There are many 6502 assemblers available on the internet. The purpose of the assembler within the project is to have an easily accessible assembler. The assembler does not need to be the fastest, the most efficient, or the best available. The assembler will have limited functionality outside of converting assembly language mnemonic-operands pairs into machine code.

#### Electricity

Realistic electricity does not need to be simulated. For most computer systems, the electrical signals will simply be interpreted as high or low. There is little benefit from implementing realistic electricity in a system that will only interpret it as binary values.

#### Invalid Computer Systems

Not all computer systems are expected to be possible to simulate within the framework created. These especially include systems that cannot have their processes broken down into steps for the step-based simulation. Additionally, computer systems that would not function in the real world are not required to be non-functional in the same way within the simulator.

#### Hardware Error Simulation

There are many ways that hardware cannot function as intended. These do not need to be simulated correctly. Errors related to the states of the components should not execute in a way that ignores these errors, but it is not necessary to simulate responses to invalid situations.

#### Simulation Variety

While a very large variety of computer systems containing a large variety of components could be simulated, it seems only necessary to demonstrate the full functionality of the simulator with one system. This is a choice made with time in mind as researching, implementing, and testing multiple computer systems would be extremely time consuming. The diversity in the components of the single simulated system is sufficiently large to demonstrate the versatility of the simulator.

#### Simulation accuracy

The simulation of a 65C02 based system should be accurate when in typical conditions. If, for example, the computer system is changed during the simulation, there is an acceptable potential for errors or inaccuracies to occur.

#### User Interface

The user interface must allow the user to run their simulation and give them access to information about its state as it is run. However, the range of computer systems that can be simulated makes it impossible to generalise a complex user interface. As such, it is likely that users will create their own interfaces. Therefore, the user interface provided in the solution is a generalised base interface that will work with any simulated system at the cost of aesthetics and a small reduction in usability.

#### Interrupts

The system being simulated does not have any components that can cause interrupts. As such, interrupt simulation is not required for the system to function correctly and so does not need to be simulated.

## Project Objectives

### 1. Components

All simulated hardware components must have a common interface as defined by the abstract Component class. This interface predominantly consists of the pin interface. Pins are a representation of the physical I/O pins of chips. Pins are separate objects in the model and have separate objectives.

**1.1. Pin Addressing**

Most components have multiple pins, so, when referring to a specific pin, there must be ways to uniquely identify that pin.

**1.1.1. Identifier**

A pin must be addressable by its identifier which is specified when initialising the Pin object. As such, pin identifiers must be unique for a component.

**1.1.2. Index**

A pin must also be addressable by an index. The indexing starts at 1 as is the typical numbering of pins on datasheets.

**1.2. Pin Methods**

In addition to pin addressing methods, components must have several additional methods some of which will have to use the addressing methods. As a general rule: a Pin method will have an equivalent in Component that calls it.

**1.2.1. Get Pin**

There must be a method or methods to return the electrical state of a pin. These methods will use the addressing method.

**1.2.2. Set Pin**

There must be a method or methods to set the electrical state of a pin. Addressing will be used to specify the pin of which the value is being affected.

**1.2.3. Pin Select**

There must be a method that returns the relevant pin object after addressing it. This is necessary for addressing since the identifier or index must be used to get the pin object so that it can be affected.

**1.2.4. Pin Count**

There must be a method that gives the number of pins that a component has. This assists in conversion between identifiers and addresses.

**1.2.5. Pin Identifiers**

There must be a method that returns a list or tuple or by other means gives the identifiers of all of the pins of a component. This allows for easy conversion between pin identifiers and pin addresses.

**1.3. Multi-Pin Addressing**

Often multiple pins need to be affected. To make this simpler, by having the iteration within the Component’s methods, pins can be addressed collectively. The order of the pins being addressed makes a difference especially when there is another argument or a return value that relates to individual pins.

**1.3.1. Identifiers**

A component’s pins must be addressable by a collection of identifiers. The constraints of each identifier are the same as in single pin addressing.

**1.3.2. Indexes**

Pins of a component must be addressable by use of a collection of indexes. Each index must be valid meaning an integer between 1 and the number of pins (inclusive).

**1.3.3. Mixed Collections**

Pin addressing must allow for the pin addresses collections to be a mixture of indexes and identifiers. This is not expected to be used frequently however it is very beneficial when it is used.

**1.3.4. Slice**

The indexes used in addressing must be able to be given as a slice object which can be used to construct a collection of indexes.

**1.4. Multi-Pin Methods**

Since multiple pins can be addressed simultaneously, there must also be methods that support the addressing of multiple pins.

**1.4.1. Get Pins**

The electrical state of several pins must be accessible by a single function call. The return value is expected to be a collection of electrical states that are returned from the single pin equivalent.

**1.4.2. Set Pins**

The electrical state of several pins must also be able to be set by a single function call. It must be possible to (a) set the pins addressed to the same state or (b) each to a separately given state. These two setting types are expected to be in separate methods, but each require only a call to that method to have the states set.

**1.5. Component State**

A component’s state is a representation of the current values of all of the attributes that relate to the internal logic of a component. For all components this will include the electrical state of all of their pins, so the pins section of state functionality will be in this base class. For many specific components there will be other parts of the state.

**1.5.1 Save**

There must be a method that fetches / calculates, formats, and returns the state of a component. This can be considered to be the equivalent of saving the state for use after the components state has potentially changed. For this objective, the pins’ states must be able to be saved.

**1.5.2. Load**

There must be a method that allows a state to be “loaded”. Loading of a state will set the values of the relevant attributes to those given. For this objective, the pins’ states must be loadable.

**1.5.3. Default**

There must be a method that loads a default state. The default state is the same for all components of the same type. The default state of pins must be all logical low.

**1.5.4. Failed Load**

If a load fails, for example if an invalid state is given, the previous state must be loaded. The previous state meaning the state when the load method is called.

**1.6. Internal Logic**

There will be a method or several methods that simulate the internal logic of each simulated hardware component. The interface of components enforces that there should be a single method that can be called to prompt the component to respond to the states of its pins.

**1.6.1. Power**

The Internal logic must be dependent on the power provided to a component. This does not apply to the power source or components that do not use an external power supply.

### 2. Inter-Component

To connect hardware components, wires are used to link their I/O pins. To represent this, there must be relationships between components as part of an inter-component network.

**2.1. Structure**

The network of wires that relate components can be effectively modelled as a graph. Most will be trees, however there is the potential for cyclical wire sections, so it is more complete to model it as a graph. In this model, the edges are representative of the wires and the nodes are the point at which multiple wires are joined. The nodes and edges are implemented as objects. There is no single adjacency matrix or list for this graph, instead the adjacency data is stored within each node object.

**2.1.1. Retrieve State**

A node must be able to retrieve its electrical state from the other nodes that it is connected to. Pin nodes are a specific type of node that makes state retrieval possible and also has the greatest use for state retrieval.

**2.1.2. Dynamic Edges**

The edges of the inter-component graph must be able to be changed after the graph is initialised. This includes both the adding (a) and removing (b) of edges.

**2.1.3. Dynamic Nodes**

Similarly, it must be possible to add (a) and remove (b) nodes after initialisation of the wire network representation. This will involve frequently adding and removing edges.

**2.2. Connection**

The edges of this graph are implemented as objects. This is unconventional but it well suited in this situation. This is because pin nodes (more later) must have a specific interface due to their relation to components which is significantly different to the interface of wire nodes. The edge objects are called connections.

**2.2.1. Node Interface**

The Connection objects must allow any node type to be accessed through the same interface. This is necessary due to the previously mentioned difference in interface. To create this interface, the node must be stored.

**2.2.2. Direction**

Wires are bidirectional in nature therefore their representations must also be bidirectional. Bidirectional meaning that the nodes at either end of an edge are able to interact with that edge in the same way, regardless of which end they are on.

**2.3. Pin Node**

A pin node is a specific type of node that represents a physical I/O pin. Pins can only be connected to one other node which means that they are always at the ends of graphs.

**2.3.1. Component**

Pin nodes are dependent on a component to which they belong. As such, pins must store their related component and no pin may be allowed to exist without a component.

**2.3.2. Electric State**

Unlike other nodes, pins must store an electric state. Other nodes can only retrieve their states from by use of their connections which will eventually traverse to pin nodes.

**2.3.3. Connection**

Pins have a single connection rather than many connections. This connection must be retrievable (a), able to be disconnected (b), and able to be replaced (c).

**2.4. Wire Node**

Wire nodes are a simple node type. They are a meeting point for multiple connections but have no attributes of their own. The only data associated with a wire node is its edges.

**2.4.1. Connections**

The connections from a wire node must be stored within it. It must be possible to retrieve the Connection objects (a), clear the connections of a wire completely disconnecting it from the graph (b), replace the connections which is most likely to be used to “move” a node (c).

**2.4.2. Connection**

Each connection associated with a node should also be able to be referenced through the node’s object. The connection-wise methods must include:

a. Retrieving a connection by use of a unique identifier such as the connected node.

b. Removing connections once again by use of an identifier.

c. Adding or creating new edges/connections.

### 3. Instruction Sets

The instructions that a processor can execute make up its instruction set. As such, simulated processors must have instruction sets. Since instruction sets are large in terms of data, they are stored in a separate object to remove the need for duplication of this sizeable data when two processors have the same instruction set.

Each instruction is made up of an operation and an addressing mode. Each of these are represented by modules (static classes) meaning they are simply subroutines and values that can be referenced by a single identifier. The objectives of addressing modes and operands must be met by the specific instructions implemented since AddressingMode and Operation base classes are abstract.

Instruction sets also need an assembler. Assemblers can be created that work with multiple instruction sets, so instruction sets, and assemblers must be separate but still associated.

**3.1. Addressing Modes**

The addressing mode of an instruction defines the process for retrieving its operand(s). The addressing mode is also relevant in the assembler because it is determined implicitly by the operands given and is used to convert those operands into machine code.

**3.1.1. Fetch Operands**

An addressing mode must be able to retrieve the operands needed to execute an instruction (a). Operands fetching should be simulated at least somewhat accurately to the real-world equivalent. This may take multiple cycles and so the simulation must require the correct number of clock cycles to execute an instruction (b). The instruction times can be found on page 20 of the W65C02S datasheet.

**3.1.2. Assemble Operands**

An addressing mode must be able to assemble valid operands (a). If operands are given, that do not match the addressing mode, an appropriate exception should be raised. This can be used to determine whether operands are of a given addressing mode. The addressing mode of assembly operands must be possible to determine by use of addressing mode objects (b).

**3.1.3. Assemble Labels**

Certain addressing modes will assemble the addresses given by labels differently, therefore an addressing mode must define how labels are to be interpreted and assembled. A default method of assembling them as a full address (16-bits for a 6502) can be used as this is the most common label interpretation.

**3.2. Operations**

The operation of an instruction is the action that it should perform. This includes loading from or saving to memory addresses, adding or subtracting values, and branching to different sections of code.

**3.2.1. Execute**

Every operation must be able to correctly perform its function on a valid processor (a). This will involve using the addressing mode’s operand fetch on certain clock cycles. Executing an instruction must take the correct number of clock cycles (b) however most of this is determined by the addressing mode so the operation must simply not undermine this.

**3.2.2. Mnemonic**

An operation must be associated with a mnemonic. These mnemonics are used when writing assembly language programs and, therefore, interpreting them.

**3.3. Instruction Set Objects**

Instruction set objects encapsulate the collection of operation and addressing mode pairs that make up the instruction set of a processor or family of processors.

**3.3.1. Instructions**

The primary purpose of instruction set objects is to store the instructions. Instruction sets must store the instructions associated with them (a), assigning each instruction an opcode either explicitly or implicitly (b), and have those instructions be publicly retrievable (c).

**3.3.2. Execute**

The instruction set must have a method that can execute a step of an instruction by use of a valid processor and opcode. This is more efficient for the user than having to find the instruction (operation and addressing mode) first then execute it in a separate method call.

**3.4. Assembler**

An assembler object or module contains methods that work together to convert assembly language programs into machine code.

**3.4.1. Identify Instruction**

An assembler must be able to identify the operation (a), addressing mode (b), and therefore opcode (c) of an assembly language instruction. This is expected to be done by use of the operation’s mnemonic and the addressing modes assemble operands method. The opcodes are defined within the instruction set object.

**3.4.2. Assemble Instruction**

An assembler must be able to use an identified instruction to return its machine code equivalent. This is done using the opcode, and the addressing mode’s operand assembling method.

**3.4.3. Assemble Programs**

An assembler must be able to assemble multiple lines of assembly code in a single method call. This is because assembling singular instructions is very rare in practical terms, most times whole programs are being assembled.

**3.4.4. Process Symbols**

An assembler must be able to interpret the use of symbols. Symbols allow data, for example a given set of operands, to be referred to, multiple times under an identifier rather than re-writing them each time.

To manage symbols, an assembler can simply store the identifiers with their relative data and use that to replace any occurrences of an identifier with the data. In the in-project assembler, a symbol must be defined before its usage and can be redefined later without affecting previous uses. In this respect, symbols act as dynamic variables.

**3.4.5. Process Labels**

Labels are a specific type of symbol that require special attention as they are references to addresses of instructions within the program being assembled. The address of a label may not be possible to determine before assembling the rest of the program potentially leaving placeholders for where assembled labels must be. This is why addressing modes having an assemble label method was necessary.

By whatever means, assemblers must be able to correctly assemble programs in which labels have been used.

### 4. Hardware Components

The subclasses of Component that are used to model physical hardware devices. As such, they will have differences. The biggest of these being the internal logic.

**4.1. Processor**

Processor objects are representations of 65C02 microprocessors. To be an accurate model, they must have certain features.

**4.1.1. Instruction Set**

The instruction set of the processor must be stored and accessible (a). The processor must use the instruction set to execute its instructions (b). The execution will involve using the instruction set to identify the operation-addressing mode pair, calculate any operands, and affect the relevant registers or memory locations.

**4.1.2. Registers**

The processor must have values stored (a) for each of its registers and these values must be accessible to retrieve or change (b). When testing this, unless different registers are implemented differently, these objectives can be shown to have been met simply if they function for multiple registers since it is very unlikely to have errors for only some register.

For reference, the registers of the 65C02 as it is being modelled are as follows: program counter (PC), instruction register (IR), processor status register (P), the register of the timing control unit (TCU), stack pointer (S), accumulator register (A), x index register (X), y index register (Y).

**4.1.3. Register Addressing**

Like pins, registers must be addressable by indexing (a); a register identifier (b); multiple indexes (c), identifiers (d), or a mix of both (e); a slice (f).

**4.1.4. State**

The processor state must also include saving (a), loading (b), and a default state (c) for the values of its registers.

**4.1.5. Internal Logic**

a. A processor must be able to execute instructions (by use of its instruction set).

b. A processor must only execute a step of an instruction once per clock cycle.

c. A processor must be able to read from and write to memory locations.

d. A processor must output the correct control signals (i.e. high when reading).

**4.2. Memory Devices**

Memory devices, such as random access memory and read only memory, function similarly. As a result, they are being modelled under a Memory class. All memory devices must do the following.

**4.2.1. Memory Data**

Memory devices must store data. This data will typically need to be a fixed length, however it would be possible to model tape for a Turing machine (as an example) which can be infinitely long.

**4.2.2. Memory Access**

Memory must be accessible by:

a. Retrieving all of the data stored.

b. Setting all of the data stored.

c. Clearing the stored data.

d. Reading from an address.

e. Writing to an address.

f. Reading from multiple addresses at once.

g. Writing to multiple addresses at once.

**4.2.3. Addressing**

A memory location must be addressable by an integer (a) or a binary number (b). Normalising and validating a memory address should be in its own method.

**4.2.4. State**

Memory device states must save (a), load (b), and clear if defaulted (c) their memory data when the relevant state methods are called.

**4.2.5. Internal Logic**

When given the relevant control signals, memory devices must read (a) and write (b). This involves reading from the address pins and reading from or writing to the data pins. Additionally, random access memory must be cleared if it loses power (c).

**4.3. Power Supply**

To provide power the entire system, there is a power supply component. It has two pins: power and ground.

**4.3.1. Power**

The power supply must be able to be turned on or off. As such, the power supply must have a power value stored (a) that defines whether the power pin should be high or low meaning powered and unpowered respectively. This value must be publicly retrievable (b) and it must be possible to set its value (c) or toggle its value (d) meaning low becomes high and high becomes low.

**4.3.2. State**

The power value of a power supply must be retrievable (a), changeable (b), and defaulted to false (c) as part of the state.

**4.3.3. Internal Logic**

a. The ground pin must be made low.

b. The power pin must be set to the power value.

**4.4. Button**

The button component being modelled in this system has four pins with two on each side of the button / push switch. As such, the two pins of each side will act as if they are connected by a wire. The button’s state works similarly to the power supply since it is boolean.

**4.4.1. Pressed**

The button component must store whether or not it is pressed (a). This value must be publicly retrievable (b), able to be set (c), and possible to toggle (d).

**4.4.2. State**

Whether or not a button is pressed is part of its state. Therefore, the state methods must save (a), load (b), and set to false as a default value (c) the pressed value respectively.

**4.4.3. Internal Logic**

The electrical state of pins on each side of the button must be combined as if they were connected by a wire (a). If the button is pressed, these two combined states must be combined (b) meaning that all four pins will be in the same state.

**4.5. Clock**

**4.5.1. Output**

The output value of the clock has the same requirements as Button’s pressed value and Power Supply’s power value. It is stored (a), retrievable (b), settable (c), togglable (d).

**4.5.2. State**

The output of a clock must be included in its state in saving (a), loading (b), and defaulting (c). The default output of a clock is low.

**4.5.3. Internal Logic**

The clock’s output pin must be set to the value in its VCC or GND pin. Which pin is dependent on whether the clock is high or low (its output value).

**4.6. Simple Components**

“Simple Components” is used here to mean components that have no additional attributes beyond the Component base class. This is not a comment on the complexity of their internal logic. The only thing that distinguishes simple components from each other and the superclass is their internal logic. As such, that is their only objective.

**4.6.1. NAND Gates**

For each gate, the Y pin must be set to the VCC or GND pin’s electrical state based on the logical NAND of its A and B pins. Meaning, if A and B are both logical high then Y is attempted to be set low by use of the ground value otherwise Y is set to the power state in an attempt to make it high. In reality, the states of A and B would have been compared to the power and ground pins to determine their logical states before NANDing them, however this yields the same logical result as comparing afterwards in this simulation.

**4.6.2. Resistor**

The internal logic of a resistor must attempt to simulate the logical affect of a resistor within the model of electricity used.

### 5. Simulation

There are some objectives that apply to the simulation as a whole rather than individual elements of it.

**5.1. Electricity Modelling**

Electricity is used to send data between components so it must be modelled in some way in this project. In whatever way it is modelled it must have certain features.

**5.1.1. Logical States**

The model of electricity must allow for an electrical state to be interpreted as high or low.

**5.1.2. Flow of Electricity**

The model of electricity must attempt to simulate the flow of electricity. This means that unconnected devices should not be able to exchange electrical signals.

**5.2. Simulated Hardware**

Having component subclasses is all well and good, but the simulation needs to run a computer system which will require it to deal with instances of these classes.

**5.2.1. Components**

The simulator must have a collection of components that are being simulated within it (a). This collection must be able to be retrieved (b), added to (c), and subtracted from (d).

**5.2.2. Component Addressing**

Each component in this collection must be addressable by an identifier (a) or index (b).

**5.2.3. State**

The state of a given component must be accessible, meaning it can be got (a) or set (b), within the simulator’s user interface.

**5.2.4. Step**

There must be some method that defines how the simulation will step. This is because this project aims to deliver a simulator that uses step-by-step execution. The step function may be different depending on the components being simulated. A typical step would involve cycling the clock and running the internal logic of each component.

**5.2.5. Multiple Steps**

The simulator must be able to run multiple steps sequentially. If a user knows it will take a certain number of steps to perform a task, they may not want to interact with the simulation until after this task. If there was no multi-step option, the user would have to instruct the simulator to step every time which could potentially take a very long time.

**5.3. Assemblers**

Similarly, an individual simulation is associated with assemblers that can be used by the user to assemble programs.

**5.3.1. Assemblers**

The simulator must have a collection of assemblers that can be used to assemble programs (a). This collection must be able to be retrieved (b), added to (c), and subtracted from (d).

**5.3.2. Assembler Addressing**

Each assembler in this collection must be addressable by an identifier (a) or index (b).

**5.3.3. Assembly Input**

The user must be able to either load an assembly program from a text file (a) or write an assembly program in the user interface (b). A loaded program must also be able to be edited within the interface a if it had been written in it (c).

**5.3.4. Save Assembly to File**

The user must be able to save an open assembly program to a text file. There is a directory named saved\_assembly within the project files where programs could be stored, but any valid file will be accepted.

**5.3.5. Assemble**

Obviously, the user must be able to assemble an assembly program that has been input into the simulator (by file loading, writing, or editing of a loaded file) using a selected/addressed assembler.

**5.3.6. Save Machine Code to File**

The machine code produced by the assembler must be able to be saved to a binary file of the user’s choosing.

**5.4. User Interface**

For the user to interact with the simulator, for example telling it to run a step, they must be able to interface with it. The user interface could be text-based, web-based, a GUI, etc. but regardless of how it is implemented, it must meet these objectives.

**5.4.1. Menus**

The set of interactions that the user can have with the simulation must be divided up into menus that feed into each other meaning that a given interaction is only available once it is relevant. For example, the user cannot assemble a program until an assembly program has been input.

**5.4.2. Console**

To allow greater flexibility in the user interface, there must be a console-like menu that allows users to interact with the objects during runtime. This is necessary as many components have differing interfaces which cannot be given full menus in a general-purpose user interface.

# Design

## General Notes for Design

### Object Oriented Model

The key parts of the Object Oriented Model were outlined in Analysis. There are a few things to note when reading the design of classes as listed below:

#### Access Modifiers

Since this project is intended to be modified and expanded, there should be no private attributes or methods. Restricted access will instead be enforced by using protected methods so that users can define new classes that inherit from the original implemented classes without losing any private attributes. This is especially useful if the user only wants to make a small change but be able to compare the effect of that change meaning that the original version cannot be affected. There will still be public attributes and methods. All attributes should be assumed to be protected, and all methods assumed public unless stated otherwise. In Python, protected attributes are defined using an underscore for example “\_variableName”, and private attributes use a double underscore (often referred to as dunder) such as “\_\_variableName”.

#### Properties

Python has a method decorator, “@property”, which allows the return value of a method to be accessed, like an attribute, using dot notation. Additionally, setters and deleters for properties can be defined allowing attribute like values to have user defined getter, setter, and deleter methods. The getter method is used when retrieving the value of a perceived attribute. This is useful if the property has a calculated value rather than a stored one. Setters are used when assigning a value using “=”, meaning that a user-defined setter can include validation. The deleter is called using “del” and is used to “delete” that property of the class. It can be useful to have a user-defined deleter if the value being deleted has an effect on other values. Many deleters can be defined using calls to the setter with a value of None (null).

#### Encapsulation

Classes allow many attributes and methods to be referenced together in a module. Static methods or class attributes that do not require an instance of the related class, or even the existence of such a class in some cases, may still be included within these modules.

A key example of this is exceptions that are raised within or in reference to an instance of the class. Such user-defined exceptions will be encapsulated within these modules.

Additionally, simple argument processing would be included as well. These are static methods that are used to convert arguments used in many methods that can take many types into a single normalised type. For example, if an object has many methods with a “string” parameter that can take many types as an argument then an argument processing method could be used to convert these arguments into str (the boult-in string data type). This allows the normalisation algorithm to be written only once in a separate static method that can be used wherever such normalisation is required instead of having similar or the same processes written out at the beginning of all of the methods that take these arguments. Of course, if the same normalisation is required in multiple objects, it may be more appropriate to have these processing functions separate from either class as a general method in global scope. Argument processing methods will not be described in the design, however, when a series of methods share similar arguments that can be many data types, it should be expected that such a method would be implemented. This is because I am only describing key methods in this section, not all of the methods.

### Class Description Key

#### Notation

Python Equivalence (description → python):

integer → int

string → str

boolean → bool

binary → bytes

error → Exception

null → None

keyWordArguments → \*\*kwargs

*Static*

Also applies to class attributes

Abstract

**Overridden**

A method or attribute that is redefined in a child class

Does not apply to abstract methods which must be overriden

[any, any, …] is a collection of specific size (e.g. [integer, string])

[any,] is an indefinitely sized collection

{any1: any2} is a dictionary with key type any1 and value type any2

method(parameter data type, …) → return type

Procedures do not return any values

#### Labels

There are labels used to represent compound data types. These are used to provide context as to the meaning of the parameter or return value. For example, PinID is used in place of integer/string to signify that it is the unique identifier for a pin, however there could be other uses integer/string that do not relate to pins in the same way.

Bit = boolean/integer

Bits = [Bit,]/binary

BinElec = [Bit, Bit]

ConnectionID = Connection/Node

PinID = integer/string

PinsIDs = [PinID,]/slice

Mapping = [[PinID, PinID],]

InitialState = Bits, [[Component, Mapping],]

The InitialState is used as two parameters

State = {string: any}

Address = integer/binary

Addresses = [Address,]/slice

Data = [binary,]/binary

Data is used for multiple binary values

Register = string/integer

Registers = [Register,]/slice

Instruction = [Operation, AddressingMode]

Opcode = integer/binary

### Binary Electricity Model

To simulate most logic circuits, accurate simulation of electricity is not required. The only information needed is whether an electrical value should be interpreted as high or low as well as what should happen when it is combined with another value. Both of these are binary (1 or 0) / boolean (True or False), meaning that they can only take on one of two values. I call these two properties as value and activity respectively. They interact as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Xa + Xa = Xa | 1a + 0a = 1a | X1 + Y0 = X1 |  |

Where each property pair is represented as valueactivity. Additionally,X, Y, and a can all be 1 or 0 (true or false) with the statements still holding true.

This abstracts away the need for potential difference, current, and resistance making the electrical side of the simulation much simpler while having a limited effect on the logic simulation. Activity is not representative of a real-world quantity, unlike value which is a simplification of potential difference. Activity can be described as whether the value is being actively enforced for example by a power supply which will constantly be making its positive pin (more later in Pins) high (1) and its ground pin low (0). The two states of activity can be considered to be active (1) and passive (0).

Instances of binary electricity can be represented in many ways but consists of only two properties, so a user-defined class does not seem necessary. Instead, I have used two boolean values, often stored in a tuple. E.g. (True, False) which has a high value and is active. If binary electricity were implemented as a user-defined object, it would require value and activity attributes or properties, and should include a method to combine two binary electric states using the principals of behaviour stated above. In later parts of design, the data type of a binary electricity instance may be stated as “BinElec”.

## Inter-Component

“Inter-component connections” refers to the way that the simulated hardware components in a computer system exchange data. Physically, this is done by wires connecting input output pins to each other. These connections between components can be modelled as a graph. The nodes of this graph are either Pins or Wires. Wires are better described as junctions in the wires e.g. a point where multiple wires intersect; however, wire nodes can still exist with only one connected wire edge. The edges, equivalent to physical wires, are contained within Connections to allow interactions with the two different types of nodes under a uniform interface. Additionally, rather than storing the graphs edges in an adjacency matrix or list, each node stores the nodes that it is connected to. This is effectively an adjacency list where each entry in the list is stored separately. The graph is undirected and unweighted.

### Node

Node is an abstract class inherited from by the two node types: Pin and Wire. Node provides methods that are useful for both node types and allows nodes to be referred to collectively in type hints. The inter-component system has been designed in such a way that additional node types could be added by users if they deem it necessary.

##### Excluded Node Exception

The ExcludedNodeError is raised during state retrieval if the node is already in the exclude list. This should not happen as retrieveState methods should have checked the exclude list against a node before calling its retrieveState method via the connection’s retrieveState. This will make more sense after reading the descriptions of the retrieveState methods and seeing this exception’s uses.

##### Specific Connection

The SpecificConnection class attribute (meaning an attribute that is not dependant on any one instance but instead on the class similar to a static method) is used to store or define the Connection subclass that interfaces with that component. In the abstract class, SpecificComponent is assigned the abstract Component class simply to declare the attribute and its expected data type.

##### Form Connection

When connecting a node, either an existing connection can be passed to that node, or a new connection must be initialised to connect it to a given node. Since this applies to all nodes, formConnection is a method defined in the Node class that will take a connection or node and create or validate a connection from this node. This connection is then returned from the method.

##### Retrieve State

The state of a node is determined by the state of the nodes that it is connected to. As such, every node needs to fetch its state from the nodes that it is connected to. Different node types may have different processes to retrieve their states, so the retrieveState method in Node is abstract. This is what makes Node an abstract class. All retrieveState methods will take an exclude argument (more later).

### Connection

Connection is another abstract class in the inter-component system. Connections are the edges of the graph. The edges are bidirectional / undirected, so connections always come in pairs. A connection pair will always link the same two nodes, to link different nodes a new connection must be instantiated. As a result, a connection will not change after instantiation.

The reason that edges are modelled by objects is so that a single connection interface can be used to interact with nodes of a variety of interfaces. Each node type (Pin and Wire in the base project) will have a separate connection subclass associated with it. This is because each node type may have a different interface and, as such, will require different processes to map that interface to the consistent connection interface. If two or more node types have the same or a similar enough interface, they can be associated with the same connection subclass.

#### Attributes

##### Connection Types

There is a class attribute connectionTypes. This is a dictionary that maps a node type as a key to a connection subclass.

This is a public attribute meaning that errors can arise from it being changed. The reason it must be a public attribute is that subclasses of Connection are referenced within it, but those subclasses cannot exist until after Connection is defined. Having the connectionTypes public also makes it easy for a user to add new entries for any new node types that they may create.

##### Node

The protected node attribute stores a reference to the node at the end of the connection. This can be used in traversing the inter-component connections graph as is done within the retrieve state process. Traversal is also useful in saving a computer system or in modifying the graph (for example to simplify it or convert it into a tree).

##### Inverse

The inverse of a connection is the other member of the connection pair. Such that the inverse of an A to B connection is a B to A connection. Since the inverse of a connection is stored, a single connection has access to the entire connection pair.

##### Exceptions

A ConnectionNotFoundError is raised when attempting to address a connection of a node using an identifier that does not match any connections of that node. This is not relevant for Pin nodes since they do not have multiple connections, however Connection’s unified interface assumes multiple connections so the translation method between the interfaces may raise ConnectionNotFoundErrors.

An IrrelevantConnectionError is raised when attempting to connect to a node using an existing connection if that connection does not involve the node. If the node is A and the connection is a B to C connection, the connection has no relevance to the node and so cannot be connected.

A WrongConnectionTypeError is raised by specific connections when there is an attempt to initialise them with a node type that is not explicitly compatible. For example, initialising Wire.SpecificConnection with a pin as its target node.

#### Methods

##### Node Property

The node property only provides a getter as changing the node would result in a completely different connection pair. The getter returns a reference to the node stored in the protected node attribute.

##### Abstract Methods

Connection subclasses must define the following methods to comply with the connection’s consistent node interface.

The connect method is used to add a connection to a node. It takes a single argument, connector, which is either a Connection or a Node that can be used to form a new connection.

There is a disconnect method that removes a connection from a node and calls its destructor (since there is no longer a direct connection between the two nodes). It must take an argument to identify the connection to remove. Connections of a node can be uniquely identified by a Connection object or a Node object, so these are the two options for the disconnect identifier.

Every node has a part in the state retrieval process. All of these retrieveState methods have an exclude parameter, so that is the signature of the Connection.retrieveState method that will retrieve the state of its target node. For some node types, the connection’s retrieveState will be the one to check for an ExcludedNodeError.

##### Create Connection

The createConnection static method will initialise the relevant specific connection. To do this, the createConnection method must take the same arguments as the constructors of connections: source, target, and inverse where source and target are Nodes and inverse is a Connection or None. When determining the correct specific connection to initialise, the connectionTypes class attribute is used. This means that, if the connectionTypes dictionary is not up to date with all of the node-types’ specific connections, there may be an exception raised.

#### Magic Methods

##### Constructor

The constructor (\_\_init\_\_ in Python) is called by all specific connections to handle the inverse creation and node connection. It takes three arguments source, target, and inverse where source and target are Nodes and inverse is a Connection or None. The inverse argument should have a default value of None. To begin with, the node is set as the target.

If no inverse is given (inverse = None) the constructor must initialise an inverse to assign to its attribute. To do this, the createConnection static method is called with the target as the source, the source as the target, and the Connection being initialised as the inverse. Before initialising an inverse, the inverse attribute must be set as None because it is used in the inverse as it is being initialised, so it being undeclared would cause an error. After initialising an inverse, it will connect the nodes to each other. It does this by calling its connect method with the inverse as the connector then the inverse’s connect with itself as the connector.

If, instead, an inverse is given, it must be validated. A valid inverse is a Connection object that does not have an inverse yet (None). If either of these requirements is not met, an appropriate exception is raised. The inverse attribute is set as the valid inverse.

Within the constructor of a Connection subclass, the node type should be validated since only specific node types are compatible with specificConnections. It will then call the superclass constructor described above. The node attribute is re-assigned as the target after calling the superclass in the implementation so that code inspections (which are present in many Python editors) are made aware that the node attribute is of the node type that is compatible as was determined earlier in the constructor. This is not necessary for the program to run, however, in a system that is expected to be edited by other programmers, the number of potential errors identified by code inspection should be kept to a minimum.

##### Destructor

The destructor (\_\_del\_\_) will disconnect the connected node and destruct the inverse connection. There will need to be exception handling in case an exception is raised during the constructor in which case the destructor is called but the attributes may not all have been instantiated yet.

##### Invert

\_\_invert\_\_ is called using a tilde (~). The inverse is returned when the invert method is called. As such ~(~connectionA) == connectionA.

##### Equality

The \_\_eq\_\_ method is used to compare an object of a class to a given value or object. It is invoked using “==”. Connections are considered equal if they are of the same specific connection and both target the same node. If \_\_eq\_\_ is defined in any node types, this would lead to errors here where separate, but similar, nodes are considered to be the same node.

### Pin

The Pin class is a direct representation of physical input/output pins on hardware components. As such, Pin has compositional association to Component meaning that Pin should only be instantiated within Component. Pins are what allow simulated components to interface with each other through use of the other parts of the inter-component connection system.

When connecting pins to other parts of the simulated system, pins can only be connected to one node. This is to simplify the connections from pins meaning that multi-connection methods need only be defined in Wire nodes. Therefore, pins will typically have an immediate connection to a non-pin node.

#### Attributes

##### Identifier

Pin identifier is a string that can be used to identify a pin. It would be good practice to assign the identifiers listed on a component’s datasheet or, if it doesn’t have one, the pin’s number.

##### Value & Activity

Pin objects have a binary electricity state (value and activity). This could be stored as a singular BinElec attribute or as two separate attributes. I chose to use two separately stored boolean values for value and activity.

##### Connection

As I stated previously, pins can only be connected to a single node. The connection attribute exists to store a Connection. This may be any type of specific connection and does not have to be a Pin.SpecificConnection because the other Node is what determines the connection type.

##### Specific Connection

Pins have a specific connection that defines the conversion between the Pin interface and the Connection interface.

The constructor is as described in Connections. It checks that the target node is a Pin, calls the superclass’ constructor, then sets the node attribute as the target again.

The connect method calls Pin’s connection setter (below) with the connector as the value being set.

The disconnect method must validate the identifier. If the identifier is None, the Pin’s connection is disconnected. If the identifier is not None, it must be a valid identifier for the current connection. The valid connection can be given either as a Connection object or as a Node object. A valid Connection is one that is equal to the connection attribute or its inverse. The valid Node is the node at the end of the connection attribute. This is the same validation as Wire.getConnection uses for Connections and Nodes. However, the pin connection only has to compare against itself as the one connection.

The retrieveState method checks whether the node is excluded. If it is, an ExcludedNodeError is raised. If it is not, the electrical state of the Pin is returned.

#### Properties

##### Identifier

Identifier should not be changed after initialisation. Therefore, its property only has a getter function that returns the value of the identifier attribute. The attribute is protected and, as such, cannot be accessed outside of the class, however the getter function allows its value to be retrieved but not modified.

##### Connection

The connection getter will return the connection stored.

There is also a setter for connection that includes validation and conversion. This means that the setter can accept an existing connection or initialise a new connection from a given wire or pin. Node.formConnection is designed to handle the validation and conversion. The setter should also be able to correctly set the connection attribute to None, meaning not connected. The setter will disconnect any existing connection since pins can only have a one connection at a time. If the new connection is equivalent (\_\_eq\_\_) to the existing connection, the existing connection is not affected.

To disconnect, the deleter is called which sets the connection attribute to None and deconstructs the previous connection if there was one. Since deconstructing the connection will call the specificConnection’s disconnect method which calls back to this method, the connection attribute must be set to None before deconstructing otherwise it will end up in a never-ending, recursive loop.

##### Value & Activity

Value and activity should both have getter and setter methods. These are very similar methods with the only difference being the attribute that they affect. As can be expected, the getters will simply return the value (not value in the binary electric sense) of the relevant attribute. The setters must validate the argument given then set the attribute as the valid value or activity. Valid arguments would be 1, 0, true, and false. There should also be a third property of state acts on the whole BinElec state of the pin at once.

#### Methods

##### Value & Activity Methods

There are two methods for each of value and activity: set, reset, active, passive. These methods use implicit arguments to set the value or activity, for example set is equivalent to calling the value setter with true (or 1) as the value argument. These methods skip the validation and conversion step required in the setters making them more efficient in situations when the value and/or activity being set is constant every time that line of code is reached.

##### Retrieve State

The retrieveState method in Pin is the start of the process of retrieving a state. It takes a single argument of exclude which, in most cases, will be the default value of an empty tuple. The pin will check whether it is in the exclude list, raising an ExcludedNodeError if it is. If the exception is not raised, the Pin will add itself to the exclude list and call the retrieveState method of its connection with the extended exclude list as an argument. The exclude argument is more relevant in Wire.retrieveState when there are multiple connections some of which lead to nodes that have already been visited. The connection’s retrieveState returns a BinElec state that is used to set the state of the Pin. The state is also returned.

The state could be set directly, making the assumption that Connection.retrieveState will always return a valid BinElec, or the returned value can be passed into the state setter. Alternatively, if state validation was in a function of its own, the validation could be called to ensure that the state set is valid, and no unexpected errors have occurred during state retrieval.

#### Magic Methods

##### Constructor

Pin has a constructor that takes the identifier as an argument. There are also additional optional parameters that have default values, these are: state, which defaults to (False, False), used to define the initial state of the pin; and connection which is used as the argument for the connection setter which is called during instantiation and defaults to None.

##### Destructor

\_\_del\_\_ is the destructor called whenever an object is deleted. This can be caused by directly using del, or by garbage collection when all references to it are deleted. When a pin no longer exists, nothing can be connected to it, so its connections destructor must be called.

### Wire

Wire nodes are the second of the two Node types in the base project. This class is not a direct representation of a real-world part of a computer system. However, it can be considered to be similar to a joint between wires (a point where multiple wires are connected). Unlike Pins, a single wire can be connected to multiple nodes.

#### Attributes

##### Connections

The connections attribute stores a list of Connections for all of the connected nodes. The connections list can be thought of as a single entry in an adjacency list for the inter-component graph. A list is used because connections can be added and removed which is easier to do using a list which is dynamic as opposed to a tuple which is static and immutable.

By storing the entries separately, every node still has access to all of the data about its edges and does not have access to any data on nodes that it is not connected to, however nodes do not need to have data on unconnected nodes, so this is not a problem. This also means that nodes do not need to access a global adjacency list or adjacency matrix. Additionally, by avoiding the use of a global adjacency list or matrix, this also avoids a potential issue that could arise when running multiple simulations simultaneously.

##### Specific Connection

Wires have a specific connection that defines the conversion between the Wire interface and the Connection interface.

The constructor is as described in Connections just is it is for Pins. The difference between the Pin.SpecificConnection and Wire.SpecificConnection is that, in this method, Wire is the accepted Node type.

The connect method calls Wire.connect with the connector as an argument.

The disconnect method calls Wire.disconnect with the identifier as an argument. This means that, through the connection, a wire’s connections are not explicitly shown to be indexable, however passing a valid index into this method will disconnect the relevant connection.

The retrieveState method calls the Wire’s retrieveState method passing on the exclude argument.

#### Properties

##### Connections

The connections property is directly related to the connections attribute. The connections property provides a getter, setter, and deleter function.

The getter must not directly return the connections list as it would be passed by reference giving access to it undermining its protected access modifier. With access, connections could be added and removed from it without proper validation. Instead, a tuple containing the lists entries is returned. A tuple is immutable making is well suited to this.

The setter takes an ordered collection of Connections or Nodes as an argument. First, the current connected nodes are stored in a new variable and the connections deleter is called. Then each connector of the argument is used as an argument in Wire.connect to connect each one. If any exception is raised, it is caught, and the previously connected nodes are re-connected to return the node to its original state. The exception is then raised again.

The deleter method will disconnect each connection by using it as the argument in Wire.disconnect.

#### Methods

##### Connection Methods

The connections property allows all of the associated Connections to be referenced collectively. To access individual connections, there are three connection methods that act similar to a getter, setter, and deleter.

The getter equivalent is getConnection which takes one argument used to uniquely identify a connection. The ways to uniquely identify a connection are as follows: an equivalent of the connection such that Connection.\_\_eq\_\_ returns True (this includes the connection itself); the inverse of a connection that satisfies the first option; the Node connected via the connection being identified; the index of the connection in the connections attribute as can be found using the connections property’s getter. If the identifier does not fit any of these options, an exception is raised. If there is no connection that matches a valid identifier, a ConnectionNotFoundError is raised.

To add a connection to the connections collection, the connect method is used. This method works similarly to the connection setter of the Pin class in that it takes a single argument, validates that argument, and converts it if necessary. To validate and convert the argument, the formConnection method, defined in the Node base class, is used. The formed connection is compared against the connections attribute. If there is an equivalent connection, the connections attribute is not affected, otherwise the new connection is appended to the end of the list. This prevents duplicate connections.

The disconnect method takes an identifier, passes it into getConnection to retrieve the connection, removes that connection from the connections list then destructs it.

##### Retrieve State

A wire’s role in the value retrieval process is to combine the states of its connected nodes. It takes a single argument, exclude, which is a collection of the nodes that have been visited during the retrieval. Exclude is used to prevent the traversal from getting stuck in cyclical sections of the graph. This is done by comparing the node of each connection to exclude and skipping them if they are excluded meaning in the collection. If it is not skipped, the connection’s retrieveState method is called.

Before making calls to its connections, a few things must happen. The first is that the wire must check that it is not part of exclude, if it is, it should raise an ExcludedNodeError exception. Next, it must add itself to the exclude collection. Because it is frequently appended, a list would be a suitable data structure for exclude.

Pins are not added to exclude when they are retrieved from, with the exception of the first Pin in the retrieval. This may seem like it would cause problems, however, pins are only connected to a single node, so as long as that node is only visited the once, each pin will only be visited once. Hence Pins do not need to be added to exclude and not adding them is more efficient, so they are not added.

The retrieveState method will return combined binary electric state of its connected nodes. This state is calculated based on the three rules of electric interactions. These rules only define the combinations of two binary electric states, so it is most efficient to combine states as they are retrieved. 0­0 has no effect on the state it is combined with and can therefore be used as a starting value. This also means that an unconnected wire end would have a state of 0­0­ which makes logical sense.

To combine values, there is a function defined in general.py which contains many simple functions that are used in many places across the project. The design section will not necessarily refer to these functions frequently because it focuses on the major classes and objects in the object oriented model.

#### Magic Methods

##### Constructor

The constructor takes one argument to be used to set the initial state of the connections attribute by calling the Connections setter function. As such, it is an ordered collection of Connections that defaults to an empty tuple meaning there are no connections.

##### Destructor

The destructor must, as with the other inter-component classes, must disconnect itself. It does this by calling the connections property’s deleter.

##### Length

\_\_len\_\_ defines how the len function interacts with an instance of a class. The len(gth) of a wire is defined to be the number of nodes connected to it which is equal to the len of its connections attribute.

##### Item Addressing

Wires also have \_\_getitem\_\_ and \_\_delitem\_\_ methods. These two methods correspond to calling getConnection and disconnect respectively

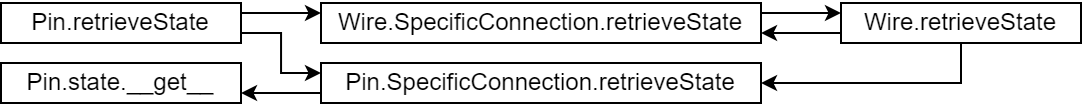
#### Diagrams

A class diagram for the inter-component classes and their relations to each other.

Shape

Description automatically generated with medium confidence

A diagram showing the methods used during state retrieval. The method called is pointed to by the method that called it.



Component.connectPin (described later) defines how pins can be connected without affecting their existing connection. This is illustrated below with filled circles representing Pin nodes, unfilled circles representing Wire nodes, lines representing Connection pairs between nodes, and the trailing arrows showing where a new Node can be connected.

Background pattern

Description automatically generated

#### Class Descriptions

Connection = Class

Public

*connectionTypes: {Type: Class}*

*ConnectionNotFoundError = Class extends error*

*IrrelevantConnectionError = Class extends error*

*WrongConnectionTypeError = Class extends error*

*Function createConnection(Node, Node, Connection)*

Function node.\_\_get\_\_() → Node

Procedure \_\_init\_\_(Node, Node, Connection)

Procedure \_\_del\_\_()

Function \_\_invert\_\_() → Connection

Function \_\_eq\_\_(Connection) → boolean

Procedure connect(ConnectionID)

Procedure disconnect(ConnectionID)

Function retrieveState([Node,]) → BinElec

Protected

node: Node

inverse: Connection

Wire = Class extends Node

Public

SpecificConnection = Class extends Connection

Public

Procedure connect(ConnectionID)

Procedure disconnect(ConnectionID)

Function retrieveState([Node,]) → BinElec

Procedure \_\_init\_\_(Node, Node, Connection)

Function connections.\_\_get\_\_() → [Connection,]

Procedure connections.\_\_set\_\_([ConnectionID,])

Procedure connections.\_\_delete\_\_()

Function getConnection(ConnectionID/integer) → Connection

Procedure connect(ConnectionID)

Procedure disconnect(ConnectionID/integer)

Function retrieveState([Node,]) → BinElec

Procedure \_\_init\_\_([ConnectionID,])

Procedure \_\_del\_\_()

Function \_\_len\_\_() → integer

Function \_\_getitem\_\_(ConnectionID/integer) → Connection

Procedure \_\_delitem\_\_(ConnectionID/integer)

Protected

connections: [Connection,]Node = Class

Public

*ExcludedNodeError = Class extends error*

***SpecificConnection = Class extends Connection***

Function formConnection(ConnectionID) → Connection

Function retrieveState(exclude: [Node,]) → BinElec

Pin = Class extends Node

Public

SpecificConnection = Class extends Connection

Public

Procedure connect(ConnectionID)

Procedure disconnect(ConnectionID)

Function retrieveState([Node,]) → BinElec

Procedure \_\_init\_\_(Node, Node, Connection)

Function identifier.\_\_get\_\_() → string

Function value.\_\_get\_\_() → Bit

Procedure value.\_\_set\_\_(Bit)

Procedure set()

Procedure reset()

Function activity.\_\_get\_\_() → Bit

Procedure activity.\_\_set\_\_(Bit)

Procedure active()

Procedure passive()

Function state.\_\_get\_\_() → BinElec

Procedure state.\_\_set\_\_(BinElec)

Function connection.\_\_get\_\_() → Connection

Procedure connection.\_\_set\_\_(ConnectionID)

Procedure connection.\_\_delete\_\_()

Function retrieveState([Node,]) → BinElec

Procedure \_\_init\_\_(string, BinElec, ConnectionID)

Procedure \_\_del\_\_()

Protected

identifier: string

value: boolean

activity: boolean

connection: Connection

## Component

Component is an abstract class used to define the interface for all simulated hardware components. Being an abstract class, Component cannot be instantiated, but all hardware components should inherit its interface and implement its abstract methods. Many parts of this class may be changed in some of its child classes.

#### Attributes

##### Pins

The pins attribute is a collection of all of the pins associated with a component. Pins are compositionally associated with components meaning that pin objects are only ever be instantiated within a component. While, technically, a pin can be created separate from a component, this pin would have no meaningful use. The pins must be stored in an ordered collection which is not expected to change after the component has been initialised. This collection is the pins attribute. Since they are not changed after initialisation, a tuple is an ideal data structure for storing these pins since it is ordered and immutable.

##### Exceptions

PinNotFoundError is a class attribute meaning that it is not dependant on an instance. It is an exception, specifically a KeyError, used when the component does not have a pin with the identifier provided.

PinIndexError is an IndexError that is raised when a pin is addressed by an index (integer) that is outside of the range of valid indexes. A valid index is between one and the total number of pins that a component has.

NoComponentError is a TypeError used to signify that None has been used in place of an instance of a simulated hardware component. This can be raised instead of a generic error to provide more detail.

StateError is used to signify that an attempt to load a state failed because there were not keys missing from the state that prevent it from being a valid state of that component.

#### Properties

##### Pins

To make identifier-based pin addressing easier, the pins property returns an ordered collection of the identifiers of the pins of a component. The order is that of the pins’ indexes. As can be expected, the pins property only has a getter because pins do not change after object instantiation. Despite being a property, for large components the pins getter takes a significant time to execute, not enough to be impactful, but for efficiency it is better to call the getter once and store its return value than to call it multiple times.

##### Pin Count

Component has a pinCount property that returns the number of pins that the component has. The pinCount property only has a getter because a component’s pins cannot be changed after instantiation, and, as such, the number of pins will not change either.

##### Active Pins

Similar to the pins property, this getter returns a tuple of identifiers. The identifiers of activePins are only those of pins that are active (pin.activity == True). This could be done by having a list of the pins that are active stored permanently, however that is not memory efficient especially if activePins is rarely called. Instead, the getter iterates through the pins and adds the active ones to a list and returns a tuple conversion of the finished list. As with pins, fewer gets are more efficient.

##### State

The state property has a getter, setter, and deleter. The getter is used to “save” the state of a component so that a user can “load” that state at any point using the setter. The setter should never instantiate a new component object, but, instead, change the component to match the state given. The deleter should return the component to its initial state, with initial here disregarding the arguments passed into the constructor.

In the superclass, the state is a dictionary with a single key: “pins”. The value associated with the pins key is an ordered collection (e.g. a tuple) of the BinElec states of the pins in order of index. The deleter resets the state of a component, so, as a result, any two instances of the same component will be the same after their state is deleted. The deleter can also be utilised in the constructor so set all pins to 0­0 before applying the modifications caused by the arguments.

It should be noted that state does not store any data about the connections of a component. This is a deliberate choice that was made for several reasons:

First, it cannot be assumed that any specific connection objects still exist since a state can be loaded after an indefinite amount of time or even in a different time that the simulator is run. As such, any references to connections stored may become invalid.

The alternative to storing connections is to store data that would allow connections to be constructed. This, however, would be likely a very large amount of data and it would not be possible to use this data to integrate a component into an inter-component graph.

Finally, the state of a component is focused on the component and not on the computer system it is in. The connections a component has does not directly affect its internal logic and, resultantly, should not be stored as part of its state. When states are used to save an entire computer system, there will need to be data stored on the connections between components in order for the system to be reconstructed.

The state property is expected to be overridden in other components but make a call back to the property in the superclass. This allows the state saving of pins, a constant in all components, to be defined only the once. Some components may not need to save any additional factors that affect state. In these classes, the state property will not be overridden.

The state of one type of component may be able to be loaded into a different component if they are similar enough. This may not be possible in both directions. For example, if the first component has additional keys in its state, these keys would be ignored when loading its state into the second component, however the state of the second component would be missing these states.

#### Methods

##### Pin Methods

There are a several methods that allow access to pins. They do not directly return Pin instances but can be used to access their BinElec states or connections (more later). To access the pins, since the object cannot be directly accessed, either its identifier, which is an attribute of the pin, or its pin index can be used to uniquely identify them. The pin index is one greater than a pin’s index in the pins attribute. This means that the pin indexes start at 1, the same as they do on many datasheets.

To make Pin addressing easier, Component has two methods: pinIndex and pinIdentifier. These allow conversion between identifiers and indexes which can be useful in normalising pin identification techniques both inside and outside the class. These methods work for both identifiers and indexes so that an identifier of unknown type can be passed into either to get a known type. Additionally, there is a protected pinSelect method that is used to retrieve a pin instance based on a single argument of either an index or identifier.

There are several methods to indirectly access pins. These are: getPin, setPin, resetPin, setPinValue, getActivity, makePinActive, makePinPassive, setPinActivity, getPinState, and setPinState. All of these methods take an identifier as an argument. This can be either a pin index or a pin identifier. The pinSelect method described above can be given this argument to validate it, raise errors where necessary, and return the Pin instance that matches the identifier so that the relevant data can be accessed and affected.

The first four methods and the four methods after that are equivalent methods that affect value and activity respectively. They each correspond to a method of Pin: getPin and getActivity use the value and activity getters respectively; set, reset, active, and passive are mapped to their obvious counterparts; setPinValue and setPinActivity make use of the value and activity setters.

The final two methods, getPinState and setPinState, are used to address both the value and activity simultaneously. Pins have methods that allow this to done easily (the state property).

##### Multi-Pin Methods

When applying functions to multiple pins, there are methods provided to contain the iteration through the pins. There is a multi-pin method for each of the single pin methods as well as some additional methods. The multi-pin equivalence includes pinIndex, pinIdentifier, and pinSelect.

The parameters when multiple pins are being addressed are of course different. Instead of a pin parameter, there must be a pins parameter. Pins can be a collection of pin indexes and identifiers. There should be functionality to process a collection that contains a mixture of both indexes and identifiers. Additionally, a collection of indexes can be expressed as a slice.

The additional methods mentioned above are setPinsValues, setPinsActivities, and setPinsStates. These methods are used to set multiple pins with different values and/or activities within a single function, as opposed to setPinsValue, setPinsActivity, setPinsState which set all of the pins addressed to a single given value. For setPinsValues and setPinsActivities, instead of a value/activity argument, they must take a values/activities argument. These arguments could be a collection of 1s, 0s, Trues, or Falses. This gives a direct equivalence to calling the single pin version multiple times. However, a collection of 1s and 0s or Trues and Falses is just binary data and as such can be expressed using the bytes data structure. Either way, the values should be the same length or longer than the number of pins being affected. If there are more values given than pins being affected, the additional values are ignored. A similar solution applies to the argument of setPinsStates except that there are two collections: one for values, one for activities.

##### Connection Methods

To connect a pin of one component to a pin of another, there is a connectPin method. It takes three arguments: pin, connectedComponent, and connectedPin. The two pin arguments are pin selectors for the respective components. The connectedComponent argument is a reference to a component object. Connecting pins in this way should not remove any existing connection relationships, as such the connectPin method may have to create new wire nodes to resolve the connections.

There is a disconnectPin method. This method calls the connection deleter of a pin given as an argument. The pin argument is either a pin index or identifier and the pinSelect method is used to retrieve the pin instance in order to call its connection property’s deleter.

There are connectPins and disconnectPins methods. Every pins argument is a collection of integers or strings, or a slice as described in Multi-Pin Methods. For connectPins, both sets of pins must be of equal length as the pins are connected sequentially not all to each other. To connect the pins of multiple components, connectPins must be called multiple times.

The connectComponent method is equivalent to connectPins except that the pins being connected are given in a different format that more explicitly shows the pairs of pins being connected. Having the pins directly matched up also avoids the potential for errors when pins collections of different lengths are given to connectPins.

There is also a connectComponents method which takes a collection of tuples of component and mapping pairs. It is a simple method that iterates through the connections given and calls connectComponent to connect each one.

##### Is Component?

There is a static isComponent method that is used to contain the exception raising for checking components’ types. If a variable is expected to be a Component, it can be passed into this method. If it is None, the appropriate NoComponentError is raised whereas if it is any other non-component type, a generic TypeError is raised. If no exception is raised, the function returns True so that it can be used in “if” statements.

##### Internal Logic

The response method is invoked to process the internal logic of a component. The method is given an @abstractmethod decorator meaning that a child class will be abstract unless it overrides this method. As such, every hardware component is expected define its own internal logic. They may use multiple methods, but this method is the one that is called to begin the internal logic simulation.

Component has a retrievePinStates method which calls the retrieveState method of its pins, passing itself as an argument to authorise the retrieval. This method should be called at the start the response method so that the component is responding to the current state of its pins. To package these two together, there is a respond method that retrieves states then calls response.

#### Magic Methods

##### Constructor

The constructor takes three arguments: pins, pinValues, and connections. Pins is either an ordered collection of identifiers to be given to the pins of the component, or pins can be the number of pins that a component has. If identifiers are given, the number of pins is defined by the number of identifiers given. If an integer is given, the pins’ identifiers will be set as a string of their pin indexes e.g. the first pins identifier would be “1”. Next, the state deleter is called to set the components state to the initial state. The pins attributes must be initialised before the deleter is called. The pinValues argument is passed into the setPinsValues method to set the current state of the component’s pins. The default value of values is an empty tuple which will cause every pin to be reset. The connections argument is a collection of collections. Each collection within the collection is a Component and a mapping. This is the same mapping as is used in connectComponent and connectComponents.

##### Destructor

The destructor must delete all of its pins causing the component to be entirely disconnected. Specific components may override this method if additional attributes must be destructed, but they should make a call back to this method as it is necessary for the pins to be deleted.

##### Other Magic Methods

No other magic methods are defined for the abstract component class. This is to allow these methods to be defined for the children of Component. For example, \_\_getitem\_\_ could be used to index the pins of a component which would be especially useful when using a slice to retrieve states from multiple pins at once. However, in memory components, \_\_getitem\_\_ would be useful to address memory addresses. If both were defined, there would be errors if the first were used on a memory device. As such, the magic methods have been left to be used in specific hardware simulations.

#### Class Description

Component = Class

Public

*PinNotFoundError = Class extends error*

*PinIndexError = Class extends error*

*NoComponentError = Class extends error*

*StateError = Class extends error*

*Function isComponent(any) → true*

*Function normalisePinValues(Bits) → [bool,]*

Function pins.\_\_get\_\_() → [string,]

Function pinCount.\_\_get\_\_() → integer

Function activePins.\_\_get\_\_() → [string,]

Function pinIndex(PinID) → integer

Function pinIdentifier(PinID) → string

Function pinSelect(PinID) → Pin

Function pinsIndexes(PinsIDs) → [integer,]

Function pinsIdentifiers(PinsIDs) → [string,]

Function pinsSelect(PinsIDs) → [Pin,]

Function getPin(PinID) → Bit

Procedure setPin(PinID)

Procedure resetPin(PinID)

Procedure setPinValue(PinID, Bit)

Function getPins(PinsIDs) → Bits

Procedure setPins(PinsIDs)

Procedure resetPins(PinsIDs)

Procedure setPinsValue(PinsIDs, Bit)

Procedure setPinsValues(PinsIDs, Bits)

Function getActivity(PinID) → Bit

Procedure makePinActive(PinID)

Procedure makePinPassive(PinID)

Procedure setPinActivity(PinID, Bit)

Function getActivities(PinsIDs) → Bits

Procedure makePinsActive(PinsIDs)

Procedure makePinsPassive(PinsIDs)

Procedure setPinsActivity(PinsIDs, Bit)

Procedure setPinsActivities(PinsIDs, Bits)

Function getPinState(PinID) → BinElec

Procedure setPinState(PinID, BinElec)

Function getPinsStates(PinsIDs) → [BinElec,]

Procedure setPinsState(PinsIDs, BinElec)

Procedure setPinsStates(PinsIDs, [BinElec,])

**Function state.\_\_get\_\_() → State** // keys: pins

**Procedure state.\_\_set\_\_(State)**

**Procedure state.\_\_delete\_\_()**

Procedure connectPin(PinID, Component, PinID)

Procedure connectPins(PinsIDs, Component, PinsIDs)

Procedure disconnectPin(PinID)

Procedure disconnectPins(PinsIDs)

Procedure connectComponent(Component, Mapping)

Procedure connectComponents([[Component, Mapping],])

Procedure retrievePinStates()

Procedure response()

Procedure respond()

**Procedure \_\_init\_\_(integer/[string,], InitialState)**

Procedure \_\_del\_\_()

Protected

pins: [Pin,]

## Instruction Set

Processors have complex internal logic systems. To assist with simulating these and to make it easier and more intuitive for the user to create processor components, instruction sets have their own objects in the model. Instruction sets can also be used in other components that use instruction-set-like logic.

Each instruction in the set is made up of an operation and an addressing mode.

### Addressing Mode

The addressing mode of an instruction defines what data about the system should be given to the operation and in what form. The addressing mode class is abstract and static. An addressing mode does not change during runtime therefore it does not need to be dynamic. Specific addressing modes will inherit from the AddressingMode superclass.

#### Exceptions

##### Addressing Mode Assemble

The AddressingModeAssembleError exception is used when an AddressingMode cannot assemble the operands it has been given (more later).

##### Labels Not Supported

LabelsNotSupportedError is used whenever the assembleLabel method is called for an addressing mode that cannot have labels meaning an addressing mode that does not use addresses.

##### Label Address

LabelAddressError is raised whenever an addressing mode is given a label to assemble that has an address that is not valid for that address.

#### Methods

##### Assemble

The assembler method takes two arguments. The first is an operand string from a line of an assembly program. This operand string is what is being assembled in this method. The other argument, labels, is a collection of all of the identifiers for the labels that could be used in these operands. The addressing mode will attempt to assemble the operands, interpreting the use of any of the labels. Since the addresses of the labels are not known, they are assumed to be absolute addresses rather than zero-page addresses. This is reasonable to assume since the zero page is designed to be used as additional registers rather than instruction memory. If the operands cannot be assembled by this addressing mode, the operands are likely in a different addressing mode, so AddressingModeAssembleError is raised. If any labels are found to be in the operands, there must be a placeholder put into the machine code of the appropriate length in bytes.

The method will return the machine code containing any place holders and a collection of collections that contains data on the uses of labels within the instruction. The inner collection is made up of the byte number within the assembled operands where the label placeholder starts, and the label’s identifier. The collection is two-dimensional collection so that instruction sets can be made where it is valid to use multiple labels in a single instruction’s operands. Additionally, the branch bit instructions could have multiple labels in some assemblers, however, as mentioned above, labels are assumed to be 16-bit rather than zero page.

##### Assemble Label

Later on in the assembling process, the addresses of labels are known (more later in assembler) and, as such, the placeholders can be replaced with actual values. Not all addressing modes will use labels in the same way. In the 6502 and 65C02 instruction sets, absolute addressing, and relative addressing are examples of such differences. Absolute addressing uses the label’s 16-bit address as the operand whereas relative addressing uses an 8-bit relative address.

The assembleLabel method takes the label’s address and the address of the start of the label’s placeholder as arguments. The method returns a bytes object containing the assembled label.

##### Fetch Operands

The fetch operands method is called by operations during execution to prompt the fetching of an operand. The method is given the processor or other instruction-set-using component as an argument allowing them to use their registers and pins. The method will return a boolean value for whether the operand has been fetched or calculated and a bytes value for the operand if it has been. If the operand fetch is not complete, for example when reading from memory which can take multiple cycles, the bytes returned should be an empty bytes object. The operand returned on an incomplete return state does not have any effect, but it is good practice to have it as a null value of the correct data type.

### Operation

The operation part of an instruction is what defines how the instruction affects the computer system. Like AddressingMode, Operation is a static abstract class that all operations inherit from.

##### Mnemonic (Attribute)

The mnemonic public class attribute is used by assemblers to identify which operation is being used in a given line of assembly. In the superclass it is an empty string.

##### Execute (Method)

The execute abstract static method takes a component and an addressing mode as its arguments. The component is the processor or other instruction-set-using component, that the operation is affecting. The addressing mode is given so that the operation can make calls to its fetchOperands method.

### Dynamic Instructions

The abstract static AddressingMode and Operation classes are useful, however they require subclasses to be defined. This means that it is very difficult to generate new AddressingModes or Operations since they have to be manually programmed. DynamicAddressingMode and DynamicOperation allow the methods and attributes needed to define new addressing modes and operations to be given as arguments in the constructor. The dynamic objects have the same interface as their static counterparts. The static classes are intended to be the default since passing functions and procedures as arguments is not very common especially at more basic levels of programming so it cannot be assumed that users will be comfortable with that concept. As such, its use in this project is kept to a minimum and alternatives are available.

### Instruction Set

An InstructionSet is, first and foremost, a container for many operation-addressing-mode pairs. Each pair is associated with an opcode. An InstructionSet also has many methods to select data from its instructions and an execute method.

##### Instructions (Attribute)

The instructions (operation and addressing mode pairs) are stored as a tuple in a protected attribute. This is suitable as the instructions should not be changed after initialisation. The index of an instruction defines its opcode. At indexes where there is no instruction, (None, None) is stored so that it can be interpreted as the operation and addressing mode each being equal to None instead of Operation and AddressingMode instances respectively.

#### Properties

##### Instructions

The instructions getter returns a tuple of all of the instructions that are not (None, None).

##### Operations

The operations getter returns a tuple of all of the Operation classes or DynamicOperation instances used in the instruction set.

##### Addressing Modes

The addressingModes getter returns a tuple of all of the addressing modes used in the instruction set.

##### Opcodes

Similar to the instructions getter, the opcodes getter returns all of the opcodes as a tuple of integers where the instruction is not (None, None).

#### Static Methods

##### Validate Instruction

validateInstruction takes an instruction then validates and normalises it to return a tuple of an operation and addressing mode or (None, None).

##### Validate Instructions

The validateInstructions methods takes a collection of instructions and calls validateInstruction on each one. It also checks that there are no repeated mnemonics. The returned instructions are in a tuple.

##### Instructions Formats

Instructions can be expressed in four accepted formats. The format used in InstructionSet is a collection of operation and addressing mode pairs. There are three methods to convert the other formats into a collection of operation and addressing mode pairs.

Opcode dictionary: the instructions are given as a dictionary where the key is the opcode, and the associated value is an operation and addressing mode pair.

Operation dictionary: the instructions are given as a dictionary where the key is an operation (Operation subclass or DynamicOperation instance) and the associated value is a collection of addressing-mode-opcode pairs with the opcode given second.

Addressing mode dictionary: similar to an operation dictionary, the instructions are given as a dictionary mapping a collection of operations and opcode pairs to addressing modes.

#### Methods

##### Operation Identification

Every Operation in an instruction set is uniquely identifiable by its mnemonic. As such, the getOperationByMnemonic method takes a mnemonic and returns an Operation. The operation parameters in InstructionSet methods accept Operation subclasses, DynamicOperation instances, and mnemonics as valid argument types.

##### Get Instruction

The getInstruction method converts an opcode to an instruction. The opcode can be given as an integer or as bytes. The byteorder used is little in this method and in almost other places where the byte order must be specified.

##### Get Opcode

The inverse function of getInstruction is getOpcode. As stated before, the operation part of the instruction can be given as an Operation subclass, DynamicOperation instance, or mnemonic string.

##### Operation and Addressing Mode Relationships

There are two methods, operationAddressingModes and addressingModeOperations, that are very similar. The operationAddressingModes method takes an operation as an argument and returns all of the addressing modes that the given operation is paired with in the instructions. The addressingModeOperations method does the same but with operation and addressing mode the other way around. The return collections should be given as tuples.

##### Execute

The execute method takes a processor Component (or other instruction-set-using component) and an opcode and executes a step of the instruction associated with that opcode. The opcode can be given as either an integer or a bytes object. The method uses the getInstruction method to find the operation and addressing mode then calls the operation’s execute method using the component and addressing mode as arguments.

##### Initialisation Methods

The constructor takes a collection of instructions as its one argument. This collection is passed through validateInstructions before it is set to the protected instructions attribute.

To initialise an instruction set using instructions in one of the three other accepted formats, there are three respective static methods that use the instruction format conversion methods to convert the instructions given into a collection of operation-addressing-mode pairs to be used in the constructor.

### Assembler

Assembler objects are used to allow for a persistence of symbols and labels between calls to the assemble method.

#### Attributes

##### Assembler Error

An AssemblerError is a generic exception that can be used to signal that something about the assembly program that is being assembled is preventing the assembling process to be completed.

##### Instruction Set

An Assembler instance must be associated with an InstructionSet instance otherwise it has no instructions of which to assemble. Assemblers are not part of instruction sets because a single instruction set could be used with multiple component instances that may have different memory domains and, as such, labels from one cannot be used in the same way in the other.

##### Symbols and Labels

The symbols and labels attributes are dictionaries that associate identifiers with strings and addresses respectively.

#### Properties

##### Instruction Set

The instructionSet getter returns the InstructionSet stored in the related attribute.

##### Symbols and Labels

The getter methods for symbols and labels return a copy of their respective attributes’ dictionaries. A copy must be returned otherwise changes to the returned value would affect the attribute.

Each setter takes either a dictionary or a collection where each item is equivalent to a key-value pair. The current dictionary is saved to a variable then the attribute is set to an empty dictionary. The setter then passes the attribute to addSymbols or addLabels. If an exception is raised, it is caught, and the saved dictionary is restored before the exception is raised again.

The deleters set the relevant attribute to an empty dictionary.

##### Symbol and Label Identifiers

The symbols and labels have properties with getters for their identifiers. The value returned from these getters is a tuple of the keys of the dictionary stored in the attribute in question.

#### Methods

##### Symbol and Label Methods

There are methods to add and remove symbols and labels. Both adding and removing require a string identifier to be given and adding requires an additional string for a symbol or address for a label.

##### Symbols and Labels Methods

In addition to the add and remove methods, there are methods that can be used to add or remove multiple symbols or labels at once. For both methods, the current dictionary must be unaffected if an exception is raised when adding or removing.

For adding, the argument is either a dictionary of identifiers and associated strings or addresses, or that same data can be given in a collection where each item is a pair of identifier and string or address.

The multiple symbols/labels remove method takes a collection of identifiers and iterates through them calling the remove method on each identifier.

##### Assemble Methods

To assemble a program, there is a public method, assemble. assemble takes two arguments: the program to assemble and the starting memory address of this program. The start address is necessary for labels. The start address is set to zero by default. Assembly programs can be either a string or a collection of lines each line being a string. Since the assembling process can add symbols and labels, a restoration clause is included so that anything added by an invalid program is undone when the error is caught and raised again.

At the beginning of the assemble process the protected preprocessing method is called. It takes the assembly argument, splits it into lines, and identifies whether each line is a symbol declaration, label, or instruction. Each symbol is added using addSymbol. Each instruction is split into its mnemonic and operands and any uses of symbolIdentifiers in the operands are replaced with the corresponding symbol meaning. The processed instructions are added to a collection. For labels, their identifier is added to a dictionary under a key of the number of instructions before it in the program. The address of the label is not known at this point because it is dependent on the length of the machine code of the assembled instructions before it. Hence the index of the instruction that the label points to in the collection of processed instructions is the closest value to a memory address that is known at this point. The method returns the collection of mnemonic-operands pairs and the dictionary for labels.

The assembler then iterates through the processed instructions collection. At each index, if there is a label in the labels dictionary returned from preprocessing that is at that index, it is added to the Assembler’s labels using addLabel with the identifier that was the value associated in the dictionary and the address as the sum of the length of the machine code that has been assembled so far and the starting address that was given as an argument. The instruction is then assembled using the protected assemble line method.

Assemble line takes two arguments. The first is the mnemonic-operands pair of that instruction. The second is a collection of the identifiers of all of the labels (both those that are in the attribute and those that are being added by this program). The instruction set associated with the assembler is used to find the operation using the mnemonic given. Technically, the mnemonic does not need to be converted into an operation since all of the instruction set methods used accept either a mnemonic or operation, but the mnemonics accepted are converted to operations using the getOperationByMnemonic method, so it is more efficient to convert once in the assembler than multiple times in many InstructionSet method calls. Then the addressing modes associated with the operation are found (using the instruction set’s operationAddressingModes method) and iterated through. For each addressing mode its assemble method is called with the relevant arguments. If the assemble attempt results in an error, the next addressing mode is tried. If all addressing modes fail, the Assembler will raise an AssembleError with a relevant message about failing to identify the addressing mode. If an addressing mode succeeds in assembling the operands, InstructionSet.getOpcode is used to find the opcode for the operation-addressing-mode pair identified to be valid for this mnemonic and operands pair. The opcode must be converted from an integer into a bytes object. Assembler.\_assembleLine returns the addressing mode identified, the machine code for that instruction made up of the opcode followed by the assembled operands returned by the addressing mode, and the label-uses returned by the addressing mode.

The machine code from the assembled lines is added to the end of a bytes object. The other returned values are used to add a new entry in a collection of label-uses. The entry is a collection of the instructions address in the program, the identifier of the label, and the addressing mode. The instructions address in the program is calculated by adding the length in bytes of all of the machine code assembled before this instruction, and the byte index of the start of the placeholder.

After all of the instructions have been assembled, the label-uses collection is iterated through, and the placeholders are replaced since the addresses of all of the labels are now defined. The addressing mode given in the label-use is used to assemble the label. As arguments, the addressing mode’s assembleLabel is given the address associated with the label in the assembler’s labels attribute, and the program address in the label-use added to the start address. This returns the assembled label that is used to replace the placeholder.

Once all of the placeholders have been replaced, the assembly program has been fully converted to machine code, so the method ends, and the machine code is returned.

##### Constructor

The constructor takes three arguments: an InstructionSet to be set to the relevant attribute, and symbols and labels that are passed to the corresponding properties’ setters. The symbols and labels can also be set to None and are by default.

#### Class Descriptions

*AddressingMode = Class*

Public

*AddressingModeAssembleError = Class extends error*

*LabelsNotSupportedError = Class extends error*

*LabelAddressError = Class extends error*

*Function assemble(string, [string,])*

*→ binary, [[integer, string],]*

***Function assembleLabel(integer, integer) → binary***

***Function fetchOperand(Component) → boolean, binary***

*Operation = Class*

Public

***mnemonic: string***

*Procedure execute(Component, AddressingMode)*

InstructionSet = Class

Public

*Function validateInstruction(Instruction) → Instruction*

*Function validateInstructions([Instruction,])*

*→ [Instriction,]*

Function instructions.\_\_get\_\_() → [Instruction,]

Function operations.\_\_get\_\_() → [Operation,]

Function addressingModes.\_\_get\_\_() → [AddressingMode,]

Function opcodes.\_\_get\_\_() → [integer,]

Function getInstruction(Opcode) → Instruction

Function getOpcode(Instruction) → integer

Function operationAddressingModes(Operation)

→ [AddressingMode,]

Function addressingModeOperations(AddressingMode)

→ [Operation,]

Procedure execute(Component, Opcode)

Procedure \_\_init\_\_([Instruction,])

*Function instructionsFromOpcodeDict({integer: instruction})*

*→ [Instruction,]*

*Function initialiseFromOpcodeDict({integer: instruction})*

*→ InstructionSet*

*Function instructionsFromOperationDict*

*({Operation: [[AddressingMode, integer],]})*

*→ [Instruction,]*

*Function initialiseFromOperationDict*

*({Operation: [[AddressingMode, integer],]})*

*→ InstructionSet*

*Function instructionsFromAddressingModeDict*

*({AddressingMode: [[Operation, integer],]})*

*→ [Instruction,]*

*Function initialiseFromAddressingModeDict*

*({AddressingMode: [[Operation, integer],]})*

*→ InstructionSet*

Protected

instructions: [Instruction,]DynamicAddressingMode = Class extends AddressingMode

Public

Function assemble*(string, [string,])*

*→ binary, [[integer, string],]*

Function assembleLabel(integer, integer) → binary

Function fetchOperand(Component) → boolean, binary

Procedure \_\_init\_\_(Function, Function, Function)

Protected

assemble: Function

assembleLabel: Function

fetchOperand: Function

DynamicOperation = Class extends Operation

Public

Function mnemonic.\_\_get\_\_() → string

Procedure execute(Component, AddressingMode)

Procedure \_\_init\_\_(string, Procedure)

Protected

mnemonic: string

execute: Procedure

Assembler = Class

Public

*AssemblerError = Class extends error*

Function instructionSet.\_\_get\_\_() → InstructionSet

Function symbols.\_\_get\_\_() → {string: string}

Procedure symbols.\_\_set\_\_

({string: string}/[[string, string],])

Procedure symbols.\_\_delete\_\_()

Function symbolIdentifiers.\_\_get\_\_() → [string,]

Procedure addSymbol(string, string)

Procedure addSymbols({string: string}/[[string, string],])

Procedure removeSymbol(string)

Procedure removeSymbols([string,])

Function labels.\_\_get\_\_() → {string: integer}

Procedure labels.\_\_set\_\_

({string: integer}/[[string, integer],])

Procedure labels.\_\_delete\_\_()

Function labelIdentifiers.\_\_get\_\_() → [string,]

Procedure addLabel(string, integer)

Procedure addLabels({string: integer}/[[string, integer],])

Procedure removeLabel(string)

Procedure removeLabels([string,])

Function assemble(string/[string,], integer) → binary

Procedure \_\_init\_\_(InstructionSet,

{string: string}/[[string, string],],

{string: integer}/[[string, integer],])

Protected

instructionSet: InstructionSet

symbols: {string: string}

labels: {string: integer}

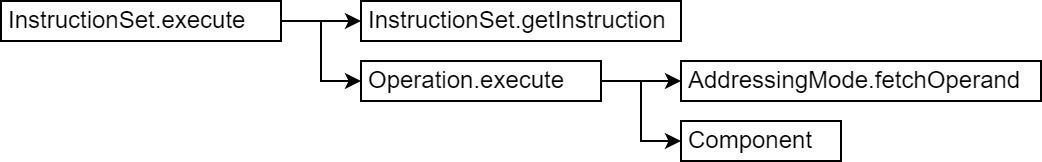
Function preprocessing(string/[string,])

→ [[string, string],], {integer: string}

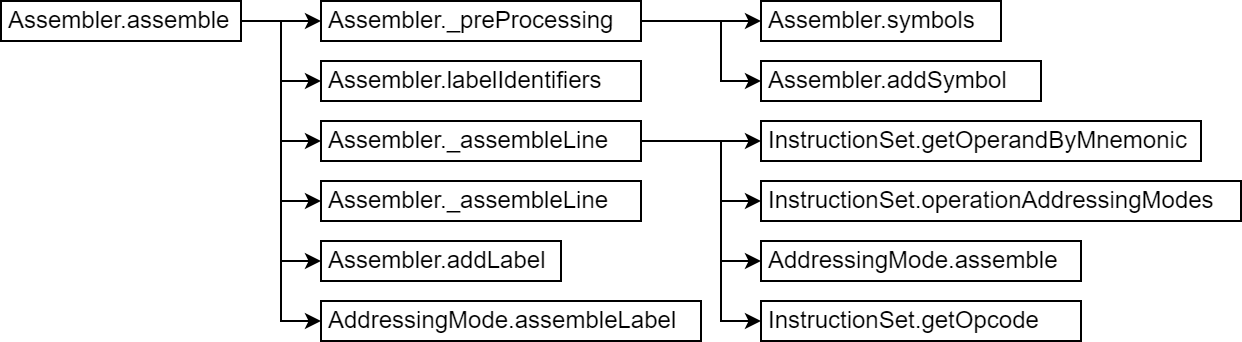
Function assembleLine([string, string], [string,])

#### Diagrams

To execute, the instruction set makes calls as follows (where Component means that any methods of the expected component associated with this instruction set may be called)



To assemble, the assembler makes method calls as follows.



## Simulated Hardware Components

// …

### Processor

// …

### Memory Devices

// …

### Other Components

// …

#### Power Supply

// …

#### Button

// …

#### Clock

// …

#### NAND Gate

// …

#### Resistor

// …

#### Class Descriptions

Processor = Class extends Component

Public

*InvalidRegisterError = Class extends Error*

Function registers.\_\_get\_\_() → [string,]

Function registerSelect(Register) → string

Function registersSelect(Registers) → [string,]

Function getRegister(Register) → binary

Procedure setRegister(Register, binary)

Function getRegisters(Registers) → [binary,]

Procedure setRegisters(Registers, Data)

Function state.\_\_get\_\_() → State // keys: pins, registers

Procedure state.\_\_set\_\_(State)

Procedure state.\_\_delete\_\_()

Procedure response()

Procedure \_\_init\_\_(InstructionSet, Data, InitialState)

Protected

instructionSet: InstructionSet

registers: {string: binary}

Procedure execute()

PowerSupply = Class extends Component

Public

Function power.\_\_get\_\_() → boolean

Procedure power.\_\_set\_\_(boolean/integer)

Procedure togglePower()

Procedure turnOn()

Procedure turnOff()

Function state.\_\_get\_\_() → State // keys: pins, power

Procedure state.\_\_set\_\_(State)

Procedure state.\_\_delete\_\_()

Procedure response()

Procedure \_\_init\_\_(boolean/integer, InitialState)

Protected

power: boolean

Button = Class extends Component

Public

Function pressed.\_\_get\_\_() → boolean

Procedure pressed.\_\_set\_\_(boolean/integer)

Procedure togglePress()

Procedure press()

Procedure unpress()

Function state.\_\_get\_\_() → State // keys: pins, pressed

Procedure state.\_\_set\_\_(State)

Procedure state.\_\_delete\_\_()

Procedure response()

Procedure \_\_init\_\_(boolean/integer, InitialState)

Protected

pressed: boolean

NANDgate = Class extends Component

Public

Procedure response()

Procedure \_\_init\_\_(InitialState)

Resistor = Class extends Component

Public

Procedure response()

Procedure \_\_init\_\_(InitialState)Memory = Class extends Component

Public

*InvalidMemoryAddressError = Class extends error*

Function validateAddress(Address) → integer

Function validateAddresses(Addresses) → [integer,]

Function read(Address) → binary

Procedure write(Address, binary)

Function readAddresses(Addresses) → [binary,]

Procedure writeAddresses(Addresses, Data)

**Function data.\_\_get\_\_() → binary**

**Procedure data.\_\_set\_\_(data)**

**Procedure data.\_\_delete\_\_()**

Procedure save(string)

Procedure load(string)

Function state.\_\_get\_\_() → State // keys: pins, data

Procedure state.\_\_set\_\_(State)

Procedure state.\_\_delete\_\_()

Procedure \_\_init\_\_

([string,]/integer, Data/string, InitialState)

Function \_\_len\_\_() → integer

Function \_\_getitem\_\_(Address/Addresses) → binary/[binary,]

Procedure \_\_setitem\_\_(Address/Addresses, binary/Data)

SpecificMemory = Class extends Memory

Public

Function read(Address) → binary

Procedure write(Address, binary)

Function readAddresses(Addresses) → [binary,]

Function writeAddresses(Addresses, Data)

Function data.\_\_get\_\_() → binary

Procedure data.\_\_set\_\_(data)

Procedure data.\_\_delete\_\_()

**Procedure response()**

**Procedure \_\_init\_\_**

**([string,]/integer, Data/string, InitialState)**

Function \_\_len\_\_() → integer

Protected

data: binary

ROM = Class extends SpecificMemory

Public

Procedure \_\_init\_\_(Data/string, InitialState)

RAM = Class extends SpecificMemory

Public

Procedure response()

Procedure \_\_init\_\_(Data/string, InitialState)

Clock = Class extends Component

Public

Function output.\_\_get\_\_() → boolean

Procedure output.\_\_set\_\_(boolean)

Procedure step()

Function state.\_\_get\_\_() → State // keys: pins, output

Procedure state.\_\_set\_\_(State)

Procedure state.\_\_delete\_\_()

Procedure response()

Procedure \_\_init\_\_(Bit, InitialState)

Protected

output: boolean

## Simulation

// …

### Simulator

// …

### User Interface

// …

#### Class Diagrams

Simulator = Class

Public

Function componentDict.\_\_get\_\_() → {string: Component}

Function components.\_\_get\_\_() → [Component,]

Function componentNames.\_\_get\_\_() → [string,]

Function identifyComponent(Component/string/integer) → string

Function getComponent(Component/string/integer) → Component

Procedure addComponent(string, Component)

Procedure removeComponent(Component/string/integer)

Function assemblerDict.\_\_get\_\_() → {string: Assembler}

Function assemblers.\_\_get\_\_() → [Assembler,]

Function assemblerNames.\_\_get\_\_() → [string,]

Function identifyAssembler(Assembler/string/integer) → string

Function getAssembler(Assembler/string/integer) → Assembler

Procedure addAssembler(string, Assembler)

Procedure removeAssembler(Assembler/string/integer)

Procedure step()

Procedure runSteps()

Function stateMenu(Component/string/integer) → boolean

Function componentMenu(Component/string/integer) → boolean

Procedure componentSelect()

*Function writeAssembly(string/[string,]) → [string,]*

*Function normaliseAssembly(string/[string,]) → [string,]*

*Procedure displayAssembly(string/[string,])*

*Function machineCodeMenu(binary) → boolean*

Function assemblyMenu(string, Assembler/string/integer) → boolean

Function assemblerMenu(assembler/string/integer) → boolean

Procedure assemblerSelect()

Procedure mainMenu()

Procedure \_\_init\_\_({string: Component}, Procedure, {string: Assembler})

Protected

components: {string: Component}

assemblers: {string: Assembler}

step: Procedure*UserInterface = Class*

Public

*UnknownBooleanReaponseError = Class extends error*

*Function format(any) → string*

*Procedure output(any)*

*Function input(any) → string*

*Function booleanInput(any) → boolean*

*Function menu([string,]) → integer*

*Function loadFile(boolean) → boolean, string/binary*

*Function saveFile(string/binary, boolean) → boolean*

*Procedure console(keyWordArguments)*

# Testing

// …

## General

// …

## Inter-Component

// …

## Components

// …

## Instruction Sets

// …

## Component Subclasses

// …

## Simulation

// …

# Evaluation

// …

# Table of References

A collection of all of the references to external sources, in this document. Although these are listed as references at specific locations in this document, many of these sources are used throughout many sections but only referenced at their first use.

|  |  |  |
| --- | --- | --- |
| **Location in document** | **Context** | **Source** |
| Analysis 6 | Peter Higginson’s AQA assembly language simulator analysed as a similar existing system | <https://www.peterhigginson.co.uk/AQA/> |
| Analysis 6 | Peter Higginson’s Little Man Computer analysed as a similar existing system | <http://peterhigginson.co.uk/LMC/> |
| Analysis 6 | Peter Higginson’s ARMLite simulator analysed as a similar existing system | <https://www.peterhigginson.co.uk/ARMlite/> |
| Analysis 6 | The visual 6502 analysed as a similar existing system | <http://www.visual6502.org/JSSim/index.html>/ |
| Analysis 6 | Emulator 101 analysed as a similar existing system | <http://emulator101.com/> |
| Analysis 7 | Ben Eater’s 65C02 based computer that is being simulated in this project | <https://eater.net/6502>/ |
| Analysis 7 | An article used for information regarding the MOS 6502 microprocessor  A datasheet for 6501 – 6505 microprocessors used for details on the technical description of these devices | <https://en.wikipedia.org/wiki/MOS_Technology_6502>  <http://archive.6502.org/datasheets/mos_6501-6505_mpu_preliminary_aug_1975.pdf> |
| Analysis 8 | A more in-depth description of the function of the 6502’s overflow flag | <http://6502.org/tutorials/vflag.html> |
| Analysis 11 | One of the sources used for information regarding the WDC 65C02 microprocessor  A datasheet for the 65C02S which is the processor in the computer system that is being simulated | <https://en.wikipedia.org/wiki/WDC_65C02>  <http://archive.6502.org/datasheets/wdc_w65c02s_oct_8_2018.pdf> |
| Analysis 15 | The datasheet for the clock used in Ben Eater’s computer system | https://www.jameco.com/Jameco/Products/ProdDS/27861.pdf |
| Analysis 16 | The datasheet for the quad 2-input NAND gate used in Ben Eater’s computer system | <https://eater.net/datasheets/74hc00.pdf> |
| Analysis 17 | The datasheet for the read only memory (ROM) chip used in Ben Eater’s computer system | https://eater.net/datasheets/28c256.pdf |
| Analysis 17 | The datasheet for the random access memory (RAM) chip used in Ben Eater’s computer system | https://eater.net/datasheets/hm62256b.pdf |
| Analysis 19 | Two surveys used to find commonly used programming languages | <https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-wanted/>  <https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages>/ |
| Analysis 20 | A comparison of programming language speeds  A comparison of programming language speeds  Used to compare of the speed of C# against Java and C++ to aid in ranking C# which did not appear in other comparisons  A comparison of the speed of JavaScript and Java to determine which should be ranked higher as they were both ranked similarly in other comparisons | <https://github.com/niklas-heer/speed-comparison/>  <http://www.hildstrom.com/projects/langcomp/index.html/>  <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/csharp.html/>  <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/javascript.html/> |

# 

# Code Dump

### File Arrangement

Computer\_system\_simulator/

instruction\_set\_65C02/

addressing\_modes.py

instructions.py

opcode\_matrix.txt

operations.py

saved\_assembly/

…

saved\_machine\_code/

…

saved\_memory/

…

additional\_hardware.py

assembler.py

component.py

general.py

instruction\_set.py

main.py

memory.py

processor.py

simulator.py

testing.py

user\_interface.py

### general.py

def int\_to\_bool(value: int or bool) -> bool:  
 if isinstance(value, bool):  
 return value  
 elif isinstance(value, int):  
 if value == 0:  
 return False  
 elif value == 1:  
 return True  
 else:  
 raise ValueError(f"Cannot convert {value} to bool (integers must be 1 or 0 to be converted)")  
 else:  
 raise TypeError(f"int\_to\_bool only converts integers to boolean not {type(value)} ({value})")  
  
def bytes\_to\_tuple(value: bytes) -> [bool,]:  
 output = list()  
 for byte in value:  
 for bit in range(8, 0, -1):  
 output.append((byte // (2 \*\* bit)) == 1)  
 byte %= 2 \*\* bit  
 return tuple(output[::-1])  
  
def slice\_to\_tuple(value: slice, maximum: int = None, minimum: int = 0) -> [int,]:  
 start, stop, step = value.start, value.stop, value.step  
 if start is None:  
 start = minimum  
 if stop is None:  
 if maximum is None:  
 raise ValueError("Cannot convert slice with no stop value to tuple if not maximum is given")  
 stop = maximum  
 if step is None:  
 step = 1  
 return tuple(range(start, stop, step))  
  
class BinaryElectric:  
 @staticmethod  
 def validateState(state: [bool or int, bool or int]) -> [bool, bool]:  
 if len(state) != 2:  
 raise ValueError(f"Binary electric states must be composed of a value and activity (not {state})")  
 value, activity = state  
 value = int\_to\_bool(value)  
 activity = int\_to\_bool(activity)  
 return value, activity  
  
 @staticmethod  
 def validState(state: [bool or int, bool or int]) -> True:  
 BinaryElectric.validateState(state)  
 return True  
  
 @staticmethod  
 def combine(A: [bool or int, bool or int], B: [bool or int, bool or int]) -> [bool, bool]:  
 A, B = BinaryElectric.validateState(A), BinaryElectric.validateState(B)  
 if A == B:  
 return A  
 elif A == (True, True) or B == (True, True):  
 return True, True  
 elif A[1] or B[1]:  
 return False, True  
 elif A[0] or B[0]:  
 return True, False  
 else:  
 return False, False

### component.py

from \_\_future\_\_ import annotations  
from abc import ABC, abstractmethod  
from general import int\_to\_bool, bytes\_to\_tuple, slice\_to\_tuple, BinaryElectric as BinElec  
  
class Component(ABC):  
 class PinNotFoundError(KeyError):  
 def \_\_init\_\_(self, identifier: str):  
 super().\_\_init\_\_(f"No pin with identifier: {identifier}")  
  
 class PinIndexError(IndexError):  
 def \_\_init\_\_(self, index: int, maxIndex: int):  
 super().\_\_init\_\_(f"Pin index, {index}, is not in pin range (1 <= index <= {maxIndex})")  
  
 class NoComponentError(TypeError):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_("The component given is None (there is no component)")  
  
 class StateError(KeyError):  
 def \_\_init\_\_(self, key: str, state: {str: any}):  
 super().\_\_init\_\_(f"Insufficient data to load state ({state} has no '{key}' state)")  
  
 @staticmethod  
 def isComponent(potentialComponent) -> True:  
 if potentialComponent is None:  
 raise Component.NoComponentError()  
 elif not isinstance(potentialComponent, Component):  
 raise TypeError(f"All components must inherit from the component class ({type(potentialComponent).\_\_name\_\_} does not)")  
 else:  
 return True  
  
 @staticmethod  
 def normalisePinValues(pinValues: [bool or int,] or bytes):  
 normalisedValues = list()  
 if isinstance(pinValues, bytes):  
 return bytes\_to\_tuple(pinValues)  
 else:  
 for value in pinValues:  
 normalisedValues.append(int\_to\_bool(value))  
 return tuple(normalisedValues)  
  
 def \_\_init\_\_(self, pins: int or [str,], pinValues: [bool or int,] or bytes = tuple(),

connections: [[Component, [[int or str, int or str],]],] = tuple()):  
 self.\_pins = list()  
 pinsIterable = pins  
 if isinstance(pins, int):  
 pinsIterable = range(1, pins + 1)  
 for pin in pinsIterable:  
 self.\_pins.append(Pin(str(pin), (False, False)))  
 self.\_pins = tuple(self.\_pins)  
 del self.state  
 if pinValues:  
 if len(pinValues) > 0:  
 self.setPinsValues(slice(len(pinValues)),Component.normalisePinValues(pinValues))  
 self.connectComponents(connections)  
  
 def \_\_del\_\_(self):  
 try:  
 for pin in self.\_pins:  
 del pin  
 except AttributeError:  
 pass  
  
 @property  
 def pins(self) -> [str,]:  
 pinNames = list()  
 for pin in self.\_pins:  
 pinNames.append(pin.identifier)  
 return pinNames  
  
 @property  
 def pinCount(self) -> int:  
 return len(self.\_pins)  
  
 @property  
 def activePins(self) -> [str,]:  
 activePins = list()  
 for pin in self.\_pins:  
 if pin.activity:  
 activePins.append(pin.identifier)  
 return tuple(activePins)  
  
 def pinIndex(self, pin: int or str) -> int:  
 if isinstance(pin, int):  
 if 1 <= pin <= len(self.\_pins):  
 return pin  
 raise Component.PinIndexError(pin, len(self.\_pins))  
 else:  
 identifier = str(pin)  
 for index in range(len(self.\_pins)):  
 if identifier == self.\_pins[index].identifier:  
 return index + 1  
 raise Component.PinNotFoundError(identifier)  
  
 def pinIdentifier(self, pin: int or str) -> str:  
 return self.\_pins[self.pinIndex(pin)].identifier  
  
 def pinSelect(self, pin: int or str) -> Pin:  
 return self.\_pins[self.pinIndex(pin) - 1]  
  
 def pinsIndexes(self, pins: [int or str,] or slice) -> [int,]:  
 if isinstance(pins, slice):  
 return slice\_to\_tuple(pins, len(self.\_pins) + 1, 1)  
 else:  
 indexes = list()  
 for pin in pins:  
 indexes.append(self.pinIndex(pin))  
 return tuple(indexes)  
  
 def pinsIdentifiers(self, pins: [int or str,] or slice) -> [str,]:  
 indexes = self.pinsIndexes(pins)  
 identifiers = list()  
 for index in indexes:  
 identifiers.append(self.\_pins[index - 1].identifier)  
 return identifiers  
  
 def pinsSelect(self, pins: [int or str,] or slice) -> [Pin,]:  
 pinsObjects = list()  
 for index in self.pinsIndexes(pins):  
 pinsObjects.append(self.\_pins[index - 1])  
 return tuple(pinsObjects)  
  
 def getPin(self, pin: int or str) -> bool:  
 return self.pinSelect(pin).value  
  
 def setPin(self, pin: int or str):  
 self.pinSelect(pin).set()  
  
 def resetPin(self, pin: int or str):  
 self.pinSelect(pin).reset()  
  
 def setPinValue(self, pin: int or str, value: bool or int):  
 self.pinSelect(pin).value = value  
  
 def getPins(self, pins: [int or str,] or slice) -> [bool,]:  
 values = list()  
 for pin in self.pinsSelect(pins):  
 values.append(pin.value)  
 return tuple(values)  
  
 def setPins(self, pins: [int or str,] or slice):  
 for pin in self.pinsSelect(pins):  
 pin.set()  
  
 def resetPins(self, pins: [int or str,] or slice):  
 for pin in self.pinsSelect(pins):  
 pin.reset()  
  
 def setPinsValue(self, pins: [int or str,] or slice, value: bool or int):  
 for pin in self.pinsSelect(pins):  
 pin.value = value  
  
 def setPinsValues(self, pins: [int or str,] or slice, values: [bool or int,] or bytes):  
 values = Component.normalisePinValues(values)  
 indexes = self.pinsIndexes(pins)  
 if len(indexes) > len(values):  
 raise ValueError(

f"Cannot set pins values with fewer values given than pins ({self.pinsIdentifiers(indexes)} set to values: {values})"

)  
 state = self.state  
 try:  
 for index in indexes:  
 self.\_pins[index].value = values[index]  
 except Exception as error:  
 self.state = state  
 raise error  
  
 def getPinActivity(self, pin: int or str) -> bool:  
 return self.pinSelect(pin).activity  
  
 def makePinActive(self, pin: int or str):  
 self.pinSelect(pin).active()  
  
 def makePinPassive(self, pin: int or str):  
 self.pinSelect(pin).passive()  
  
 def setPinActivity(self, pin: int or str, activity: bool or int):  
 self.pinSelect(pin).activity = activity  
  
 def getPinsActivities(self, pins: [int or str,] or slice) -> [bool,]:  
 activities = list()  
 for pin in self.pinsSelect(pins):  
 activities.append(pin.activity)  
 return tuple(activities)  
  
 def makePinsActive(self, pins: [int or str,] or slice):  
 for pin in self.pinsSelect(pins):  
 pin.active()  
  
 def makePinsPassive(self, pins: [int or str,] or slice):  
 for pin in self.pinsSelect(pins):  
 pin.passive()  
  
 def setPinsActivity(self, pins: [int or str,] or slice, activity: bool or int):  
 for pin in self.pinsSelect(pins):  
 pin.activity = activity  
  
 def setPinsActivities(self, pins: [int or str,] or slice, activities: [bool or int,] or bytes):  
 activities = Component.normalisePinValues(activities)  
 indexes = self.pinsIndexes(pins)  
 if len(indexes) > len(activities):  
 raise ValueError(

f"Cannot set pins activities with fewer activities given than pins ({self.pinsIdentifiers(indexes)}" +

f" set to activities: {activities})"

)  
 state = self.state  
 try:  
 for index in indexes:  
 self.\_pins[index].activity = activities[index]  
 except Exception as error:  
 self.state = state  
 raise error  
  
 def getPinState(self, pin: int or str) -> [bool, bool]:  
 return self.pinSelect(pin).state  
  
 def setPinState(self, pin: int or str, state: [bool or int, bool or int]):  
 self.pinSelect(pin).state = state  
  
 def getPinsStates(self, pins: [int or str,] or slice) -> [[bool, bool],]:  
 states = list()  
 for pin in self.pinsSelect(pins):  
 states.append(pin.state)  
 return tuple(states)  
  
 def setPinsState(self, pins: [int or str,] or slice, state: [bool or int, bool or int]):  
 for pin in self.pinsSelect(pins):  
 pin.state = state  
  
 def setPinsStates(self, pins: [int or str,] or slice, states: [[bool or int, bool or int],]):  
 indexes = self.pinsIndexes(pins)  
 if len(indexes) > len(states):  
 raise ValueError(

f"Cannot set pins states with fewer states given than pins ({self.pinsIdentifiers(indexes)} set to states: {states})"

)  
 state = self.state  
 try:  
 for index in indexes:  
 self.setPinState(index, states[index - 1])  
 except Exception as error:  
 self.state = state  
 raise error  
  
 @property  
 def state(self) -> {str: any}:  
 return {"pins": self.getPinsStates(slice(None))}  
  
 @state.setter  
 def state(self, state: {str: any}):  
 prevState = self.state  
 try:  
 pinsState = state["pins"]  
 except KeyError:  
 raise Component.StateError("pins", state)  
 try:  
 self.setPinsStates(slice(None), pinsState)  
 except Exception as error:  
 self.state = prevState  
 raise error  
  
 @state.deleter  
 def state(self):  
 for pin in self.\_pins:  
 pin.state = (False, False)  
  
 def connectPin(self, pin: int or str, connectedComponent: Component, connectedPin: int or str):  
 if Component.isComponent(connectedComponent):  
 nodes = [self.pinSelect(pin), connectedComponent.pinSelect(connectedPin)]  
 for node in range(2):  
 existingConnection = nodes[node].connection  
 if existingConnection is not None:  
 if isinstance(existingConnection.node, Wire):  
 nodes[node] = existingConnection.node  
 else:  
 nodes[node] = Wire((nodes[node], existingConnection.node))  
 Connection.createConnection(nodes[0], nodes[1])  
  
 def connectPins(self, pins: [int or str,] or slice, connectedComponent: Component, connectedPins: [int or str,] or slice):  
 if Component.isComponent(connectedComponent):  
 pins = self.pinsIndexes(pins)  
 connectedPins = connectedComponent.pinsSelect(connectedPins)  
 length = len(pins)  
 if len(connectedPins) != length:  
 raise ValueError(

f"Cannot connect {length} pins of {self} to {len(connectedPins)} pins of {connectedComponent}" +

" (Component.connectPins makes 1:1 connections)"

)  
 for index in range(length):  
 self.connectPin(pins[index], connectedComponent, connectedPins[index])  
  
 def disconnectPin(self, pin: int or str):  
 del self.pinSelect(pin).connection  
  
 def disconnectPins(self, pins: [int or str,] or slice):  
 pins = self.pinsSelect(pins)  
 for pin in pins:  
 del pin.connection  
  
 def connectComponent(self, component: Component, mapping: [[int or str, int or str],]):  
 for pin1, pin2 in mapping:  
 self.connectPin(pin1, component, pin2)  
  
 def connectComponents(self, connections: [[Component, [[int or str, int or str],]]]):  
 for component, mapping in connections:  
 self.connectComponent(component, mapping)  
  
 @abstractmethod  
 def response(self):  
 pass  
  
 def retrievePinStates(self):  
 for pin in self.\_pins:  
 pin.retrieveState()  
  
 def respond(self):  
 self.retrievePinStates()  
 self.response()  
  
class Connection(ABC):  
 connectionTypes = dict()  
  
 class ConnectionNotFoundError(ValueError):  
 pass  
  
 class IrrelevantConnectionError(ValueError):  
 pass  
  
 class WrongConnectionTypeError(TypeError):  
 def \_\_init\_\_(self, expectedType: type, givenType: type):  
 super().\_\_init\_\_(f"{expectedType.\_\_name\_\_} connections cannot be formed using type {givenType.\_\_name\_\_}")  
  
 @staticmethod  
 def createConnection(source: Node, target: Node, inverse: Connection = None) -> Connection:  
 try:  
 return Connection.connectionTypes[type(target)](source, target, inverse)  
 except KeyError:  
 raise TypeError(f"No known connection type for node type {type(target)}")  
  
 def \_\_init\_\_(self, source: Node, target: Node, inverse: Connection = None):  
 self.\_node = target  
 self.\_inverse = None  
 if inverse is not None:  
 if not isinstance(inverse, Connection):  
 raise TypeError(

"The inverse should only be used during initialisation of a connection pair" +

" and must itself be a Connection instance"

)  
 if inverse.\_inverse is not None:  
 raise ValueError(f"Cannot initialise Connection: inverse connection, {inverse}, already has an inverse")  
 self.\_inverse = inverse  
 else:  
 self.\_inverse = Connection.createConnection(target, source, self)  
 self.connect(self.\_inverse)  
 self.\_inverse.connect(self)  
  
 def \_\_del\_\_(self):  
 try:  
 self.disconnect(self)  
 del self.\_inverse  
 except AttributeError:  
 pass  
  
 def \_\_invert\_\_(self) -> Connection:  
 return self.\_inverse  
  
 def \_\_eq\_\_(self, other: Connection) -> bool:  
 if isinstance(other, type(self)):  
 return self.\_node == other.\_node  
 return False  
  
 @property  
 def node(self) -> Node:  
 return self.\_node  
  
 @abstractmethod  
 def connect(self, connector: Connection or Node):  
 pass  
  
 @abstractmethod  
 def disconnect(self, identifier: Connection or Node):  
 pass  
  
 @abstractmethod  
 def retrieveState(self, exclude: [Node,]) -> [bool, bool]:  
 pass  
  
class Node(ABC):  
 class ExcludedNodeError(Exception):  
 pass  
  
 SpecificConnection = Connection  
  
 def formConnection(self, connector: Connection or Node) -> Connection:  
 if isinstance(connector, Connection):  
 if connector.node == self:  
 return ~connector  
 elif (~connector).node == self:  
 return connector  
 else:  
 raise Connection.IrrelevantConnectionError(f"Connection, {connector}, does not involve node, {self}")  
 elif isinstance(connector, Node):  
 return Connection.createConnection(self, connector)  
 else:  
 raise TypeError(f"Can only form connection using type Connection or Node not {type(connector).\_\_name\_\_} ({connector})")  
  
 @abstractmethod  
 def retrieveState(self, exclude: [Node,]) -> [bool, bool]:  
 pass  
  
class Pin(Node):  
 class SpecificConnection(Connection):  
 def \_\_init\_\_(self, source: Node, target: Node, inverse: Connection = None):  
 if not isinstance(target, Pin):  
 raise Connection.WrongConnectionTypeError(type(self), type(target))  
 super().\_\_init\_\_(source, target, inverse)  
 self.\_node = target  
  
 def connect(self, connector: Connection or Node):  
 self.\_node.connection = connector  
  
 def disconnect(self, identifier: Connection or Node = None):  
 if identifier is not None:  
 if isinstance(identifier, Connection):  
 if identifier != self and identifier != ~self:  
 raise Connection.ConnectionNotFoundError(f"Pin's connection is not {identifier}")  
 elif isinstance(identifier, Node):  
 if identifier != (~self).\_node:  
 raise Connection.ConnectionNotFoundError(f"Pins is not connected to node {identifier}")  
 del self.\_node.connection  
  
 def retrieveState(self, exclude: [Node,]):  
 if self.\_node in exclude:  
 raise Node.ExcludedNodeError  
 return self.\_node.state  
  
 def \_\_init\_\_(self, identifier: str, state: [bool or int, bool or int] = (False, False), connection: Connection or Node = None):  
 self.\_identifier = str(identifier)  
 self.\_value, self.\_activity = BinElec.validateState(state)  
 self.\_connection = None  
 if connection is not None:  
 self.connection = connection  
  
 def \_\_del\_\_(self):  
 try:  
 del self.connection  
 except AttributeError:  
 pass  
  
 @property  
 def identifier(self) -> str:  
 return self.\_identifier  
  
 @property  
 def value(self) -> bool:  
 return self.\_value  
  
 @value.setter  
 def value(self, value: bool or int):  
 self.\_value = int\_to\_bool(value)  
  
 def set(self):  
 self.\_value = True  
  
 def reset(self):  
 self.\_value = False  
  
 @property  
 def activity(self) -> bool:  
 return self.\_activity  
  
 @activity.setter  
 def activity(self, activity: bool or int):  
 self.\_activity = int\_to\_bool(activity)  
  
 def active(self):  
 self.\_activity = True  
  
 def passive(self):  
 self.\_activity = False  
  
 @property  
 def state(self) -> [bool or int, bool or int]:  
 return self.\_value, self.\_activity  
  
 @state.setter  
 def state(self, state: [bool or int, bool or int]):  
 prevValue, prevActivity = self.\_value, self.\_activity  
 try:  
 self.\_value, self.\_activity = int\_to\_bool(state[0]), int\_to\_bool(state[1])  
 except Exception as error:  
 self.\_value, self.\_activity = prevValue, prevActivity  
 raise error  
  
 @property  
 def connection(self) -> Connection:  
 return self.\_connection  
  
 @connection.setter  
 def connection(self, connector: Connection or Node):  
 newConnection = self.formConnection(connector)  
 if newConnection != self.\_connection:  
 del self.connection  
 self.\_connection = newConnection  
  
 @connection.deleter  
 def connection(self):  
 if self.\_connection is not None:  
 connection = self.\_connection  
 self.\_connection = None  
 del connection  
  
 def retrieveState(self, exclude: [Node,] = tuple()) -> [bool, bool]:  
 if self in exclude:  
 raise Node.ExcludedNodeError(f"{self} is already excluded in {exclude}")  
 if self.\_connection is None:  
 self.\_value = self.\_activity = False  
 else:  
 self.\_value, self.\_activity = BinElec.validateState(self.\_connection.retrieveState(tuple(list(exclude) + [self])))  
 return self.state  
  
class Wire(Node):  
 class SpecificConnection(Connection):  
 def \_\_init\_\_(self, source: Node, target: Node, inverse: Connection = None):  
 if not isinstance(target, Wire):  
 raise Connection.WrongConnectionTypeError(type(self), type(target))  
 super().\_\_init\_\_(source, target, inverse)  
 self.\_node = target  
  
 def connect(self, connector: Connection or Node):  
 self.\_node.connect(connector)  
  
 def disconnect(self, identifier: Connection or Node):  
 self.\_node.disconnect(identifier)  
  
 def retrieveState(self, exclude: [Node,]) -> [bool, bool]:  
 return self.\_node.retrieveState(exclude)  
  
 def \_\_init\_\_(self, connections: [Connection or Node,] = tuple()):  
 self.\_connections = list()  
 self.connections = connections  
  
 def \_\_del\_\_(self):  
 try:  
 del self.connections  
 except AttributeError:  
 pass  
  
 def \_\_len\_\_(self) -> int:  
 return len(self.\_connections)  
  
 @property  
 def connections(self) -> [Connection,]:  
 return tuple(self.\_connections)  
  
 @connections.setter  
 def connections(self, connectors: [Connection or Node,]):  
 prevConnections = list()  
 for connection in self.\_connections:  
 prevConnections.append(connection.node)  
 del self.connections  
 try:  
 for connector in connectors:  
 self.connect(connector)  
 except Exception as error:  
 self.connections = prevConnections  
 raise error  
  
 @connections.deleter  
 def connections(self):  
 for connection in self.\_connections:  
 self.disconnect(connection)  
  
 def getConnection(self, identifier: Connection or Node or int) -> Connection:  
 if isinstance(identifier, Connection):  
 connection = identifier  
 if connection.node == self:  
 connection = ~connection  
 if connection in self.\_connections:  
 return connection  
 raise Connection.ConnectionNotFoundError(f"{connection} not in {self.\_connections}")  
 elif isinstance(identifier, Node):  
 node = identifier  
 for connection in self.\_connections:  
 if connection.node == node:  
 return connection  
 raise Connection.ConnectionNotFoundError(f"No connection to {node} in {self.\_connections}")  
 elif isinstance(identifier, int):  
 return self.\_connections[identifier]  
 raise TypeError(f"Cannot identify connections using type {type(identifier).\_\_name\_\_} ({identifier})")  
  
 def connect(self, connector: Connection or Node):  
 connection = self.formConnection(connector)  
 if not connection in self.\_connections:  
 self.\_connections.append(connection)  
  
 def disconnect(self, identifier: Connection or Node or int):  
 connection = self.getConnection(identifier)  
 self.\_connections.remove(connection)  
 del connection  
  
 def retrieveState(self, exclude: [Node,] = tuple()) -> [bool, bool]:  
 if self in exclude:  
 raise Node.ExcludedNodeError(f"{self} is already excluded in {exclude}")  
 exclude = list(exclude)  
 exclude.append(self)  
 state = (False, False)  
 for connection in self.\_connections:  
 if connection.node not in exclude:  
 state = BinElec.combine(state, connection.retrieveState(exclude))  
 if state == (True, True):  
 break  
 return state  
  
 def \_\_getitem\_\_(self, identifier: Connection or Node or int) -> Connection:  
 return self.getConnection(identifier)  
  
 def \_\_delitem\_\_(self, identifier: Connection or Node or int):  
 self.disconnect(identifier)  
  
Connection.connectionTypes = {Pin: Pin.SpecificConnection, Wire: Wire.SpecificConnection}

### instruction\_set.py

from \_\_future\_\_ import annotations  
from component import Component  
from abc import ABC, abstractmethod  
  
class Operation(ABC):  
 mnemonic = ""  
  
 @staticmethod  
 @abstractmethod  
 def execute(processor: Component, addressingMode: AddressingMode):  
 pass  
  
class AddressingMode(ABC):  
 class AddressingModeAssembleError(Exception):  
 pass  
  
 @staticmethod  
 @abstractmethod  
 def fetchOperand(processor: Component, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 @staticmethod  
 @abstractmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 class LabelModes:  
 @staticmethod  
 def immediateLabel16Bit(labelAddress: int, instructionAddress: int = None) -> bytes:  
 return labelAddress.to\_bytes(2, "little")  
  
 @staticmethod  
 def relativeLabel8Bit(labelAddress: int, instructionAddress: int) -> bytes:  
 return bytes([labelAddress - instructionAddress])  
  
 @staticmethod  
 def assembleLabel(labelAddress: int, instructionAddress: int) -> bytes:  
 return AddressingMode.LabelModes.immediateLabel16Bit(labelAddress)  
  
class DynamicOperation(Operation):  
 def \_\_init\_\_(self, mnemonic: str, execute: callable):  
 self.\_mnemonic = str(mnemonic)  
 self.\_execute = execute  
  
 @property  
 def mnemonic(self) -> str:  
 return self.\_mnemonic  
  
 def execute(self, processor: Component, addressingMode: AddressingMode):  
 self.\_execute(processor, addressingMode)  
  
class DynamicAddressingMode(AddressingMode):  
 def \_\_init\_\_(self, fetchOperand: callable, assemble: callable,

assembleLabel: callable = AddressingMode.LabelModes.immediateLabel16Bit):  
 self.\_assemble = assemble  
 self.\_fetchOperand = fetchOperand  
 self.\_assembleLabel = assembleLabel  
  
 def fetchOperand(self, processor: Component, fetchCount: int) -> [bool, bytes]:  
 return self.\_fetchOperand(processor, fetchCount)  
  
 def assemble(self, operandString: str, address: int = 0, symbols: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 return self.\_assemble(operandString, address, symbols)  
  
 def assembleLabel(self, labelAddress: int, instructionAddress: int) -> bytes:  
 return self.\_assembleLabel(labelAddress, instructionAddress)  
  
class InstructionSet:  
 @staticmethod  
 def validateInstruction(instruction: [Operation, AddressingMode]) -> [Operation, AddressingMode]:  
 if len(instruction) != 2:  
 raise ValueError(f"Instructions must be of the form (Operation, AddressingMode) not {instruction}")  
 if not (issubclass(instruction[0], Operation) and issubclass(instruction[1], AddressingMode)):  
 raise TypeError(

f"Invalid instruction types {type(instruction[0]).\_\_name\_\_, type(instruction[1]).\_\_name\_\_}" +

" must be (Operation, AddressingMode)"

)  
 return tuple(instruction)  
  
 @staticmethod  
 def validateInstructions(instructions: [[Operation, AddressingMode],]) -> [[Operation, AddressingMode],]:  
 instructions = list(instructions)  
 validInstructions = list()  
 mnemonics = {}  
 for instruction in instructions:  
 if instruction is None or instruction == (None, None):  
 validInstructions.append((None, None))  
 else:  
 instruction = InstructionSet.validateInstruction(instruction)  
 operation = instruction[0]  
 mnemonic = operation.mnemonic  
 if mnemonic in mnemonics:  
 if operation != mnemonics[mnemonic]:  
 raise ValueError(f"Operations in an instruction set must have unique mnemonics ({mnemonic} is repeated)")  
 else:  
 mnemonics[mnemonic] = operation  
 validInstructions.append(tuple(instruction))  
 return tuple(validInstructions)  
  
 @staticmethod  
 def instructionsFromOpcodeDict(opcodeDict: {int: [[Operation, AddressingMode],]}):  
 instructions = [None] \* (max(tuple(opcodeDict.keys())) + 1)  
 for opcode in opcodeDict:  
 instructions[opcode] = InstructionSet.validateInstruction(opcodeDict[opcode])  
 return InstructionSet.validateInstructions(instructions)  
  
 @staticmethod  
 def instructionsFromOperationDict(operationsDict: {Operation: [[AddressingMode, int],]}) -> [Operation, InstructionSet]:  
 instructions = {}  
 for operation in operationsDict:  
 for addressingMode, opcode in operationsDict[operation]:  
 if opcode in instructions:  
 raise ValueError(f"An opcode cannot have multiple instructions ({opcode})")  
 else:  
 instructions[opcode] = InstructionSet.validateInstruction((operation, addressingMode))  
 return InstructionSet.instructionsFromOpcodeDict(instructions)  
  
 @staticmethod  
 def instructionsFromAddressingModeDict(addressingModeDict: {AddressingMode: [[Operation, int],]}) -> [Operation, InstructionSet]:  
 instructions = {}  
 for addressingMode in addressingModeDict:  
 for operation, opcode in addressingModeDict[addressingMode]:  
 if opcode in instructions:  
 raise ValueError(f"An opcode cannot have multiple instructions ({opcode})")  
 else:  
 instructions[opcode] = InstructionSet.validateInstruction((operation, addressingMode))  
 return InstructionSet.instructionsFromOpcodeDict(instructions)  
  
 def \_\_init\_\_(self, instructions: [[Operation, AddressingMode],]):  
 self.\_instructions = InstructionSet.validateInstructions(instructions)  
  
 @property  
 def instructions(self) -> [[Operation, AddressingMode]]: *# TODO* return self.\_instructions  
  
 @property  
 def operations(self) -> [Operation,]:  
 operations = list()  
 for operation, addressingMode in self.\_instructions:  
 if operation is not None:  
 if operation not in operations:  
 operations.append(operation)  
 return tuple(operations)  
  
 @property  
 def addressingModes(self) -> [AddressingMode,]:  
 addressingModes = list()  
 for addressingMode, addressingMode in self.\_instructions:  
 if addressingMode is not None:  
 if addressingMode not in addressingModes:  
 addressingModes.append(addressingMode)  
 return tuple(addressingModes)  
  
 @property  
 def opcodes(self) -> [int,]:  
 opcodes = list()  
 for opcode in range(len(self.\_instructions)):  
 if self.\_instructions[opcode] != (None, None):  
 if opcode not in opcodes:  
 opcodes.append(opcode)  
 return tuple(opcodes)  
  
 def getOperationByMnemonic(self, mnemonic: str) -> Operation:  
 for operation, addressingMode in self.\_instructions:  
 if operation is not None:  
 if mnemonic.lower() == operation.mnemonic.lower():  
 return operation  
 raise ValueError(f"No operation in instruction set, {self}, with mnemonic, {mnemonic}")  
  
 def getInstruction(self, opcode: int or bytes) -> [Operation, AddressingMode]:  
 if isinstance(opcode, bytes):  
 opcode = int.from\_bytes(opcode, "little")  
 return self.\_instructions[opcode]  
  
 def getOpcode(self, operation: Operation or str, addressingMode: AddressingMode) -> int:  
 if isinstance(operation, str):  
 operation = self.getOperationByMnemonic(operation)  
 return self.\_instructions.index((operation, addressingMode))  
  
 def operationAddressingModes(self, operation: Operation or str) -> [AddressingMode]:  
 if isinstance(operation, str):  
 operation = self.getOperationByMnemonic(operation)  
 addressingModes = list()  
 for instruction in self.\_instructions:  
 if operation == instruction[0]:  
 addressingModes.append(instruction[1])  
 return addressingModes  
  
 def addressingModeOperations(self, addressingMode: AddressingMode) -> [Operation]:  
 operations = list()  
 for instruction in self.\_instructions:  
 if addressingMode == instruction[1]:  
 operations.append(instruction[0])  
 return operations  
  
 def execute(self, processor: Component, opcode: int or bytes):  
 operation, addressingMode = self.getInstruction(opcode)  
 operation.execute(processor, addressingMode)  
  
 @staticmethod  
 def initialiseFromOpcodeDict(opcodeDict: {int: [[Operation, AddressingMode],]}) -> InstructionSet:  
 return InstructionSet(InstructionSet.instructionsFromOpcodeDict(opcodeDict))  
  
 @staticmethod  
 def initialiseFromOperationDict(operationDict: {Operation: [[AddressingMode, int],]}) -> InstructionSet:  
 return InstructionSet(InstructionSet.instructionsFromOperationDict(operationDict))  
  
 @staticmethod  
 def initialiseFromAddressingModeDict(addressingModeDict: {AddressingMode: [[Operation, int],]}) -> InstructionSet:  
 return InstructionSet(InstructionSet.instructionsFromAddressingModeDict(addressingModeDict))

### assembler.py

from instruction\_set import InstructionSet, AddressingMode  
  
class Assembler:  
 class AssemblerError(Exception):  
 pass  
  
 def \_\_init\_\_(self, instructionSet: InstructionSet, symbols: {str: str} = None, labels: {str: int} = None):  
 if not isinstance(instructionSet, InstructionSet):  
 raise TypeError(f"An assembler must be associated with an instruction set ({instructionSet} is not valid)")  
 self.\_instructionSet = instructionSet  
 if symbols is None:  
 self.\_symbols = dict()  
 elif isinstance(symbols, dict):  
 self.\_symbols = symbols  
 else:  
 raise TypeError(

"Symbols must be given as a dictionary with the key as the symbol identifier and the value as associated assembly " +

f"(not {symbols})"

)  
 if labels is None:  
 self.\_labels = dict()  
 elif isinstance(labels, dict):  
 self.\_labels = labels  
 else:  
 raise TypeError(

"Labels must be given as a dictionary with the key as the label identifier and the value as the address in memory " +

f"(not {labels})"

)  
  
 @property  
 def instructionSet(self) -> InstructionSet:  
 return self.\_instructionSet  
  
 @property  
 def symbols(self) -> {str: str}:  
 return self.\_symbols.copy()  
   
 @symbols.deleter  
 def symbols(self):  
 self.\_symbols = dict()  
  
 @property  
 def symbolIdentifiers(self) -> [str,]:  
 return tuple(self.\_symbols.keys())  
  
 def addSymbol(self, identifier: str, meaning: str):  
 if isinstance(meaning, int):  
 self.addLabel(identifier, meaning)  
 else:  
 self.\_symbols[str(identifier).strip()] = str(meaning).strip()  
  
 def removeSymbol(self, identifier: str):  
 del self.\_symbols[identifier]  
  
 @property  
 def labels(self) -> {str: str}:  
 return self.\_labels.copy()  
  
 @labels.deleter  
 def labels(self):  
 self.\_labels = dict()  
  
 @property  
 def labelIdentifiers(self) -> [str,]:  
 return tuple(self.\_labels.keys())  
  
 def addLabel(self, identifier: str, address: int):  
 if isinstance(address, int):  
 self.\_labels[str(identifier).split()] = address  
 else:  
 self.addSymbol(identifier, str(address))  
  
 def removeLabel(self, identifier: str):  
 del self.\_labels[identifier]  
  
 def \_preprocessing(self, assembly: str or [str,]) -> [[[str, str],], {int: str}]:  
 if isinstance(assembly, str):  
 lines = assembly.split("\n")  
 else:  
 lines = tuple(assembly)  
 instructionCalls = list()  
 labelLines = dict()  
 for line in lines:  
 line = line.strip()  
 if line[-1] == ":":  
 labelLines[len(instructionCalls)] = line[:-1]  
 elif "=" in line:  
 split = line.index("=")  
 self.addSymbol(line[:split], line[split + 1:])  
 else:  
 try:  
 split = line.index(" ")  
 except ValueError:  
 split = len(line)  
 mnemonic = line[:split]  
 operands = line[split:].strip()  
 for symbol in self.symbols:  
 operands = operands.replace(symbol, self.symbols[symbol])  
 instructionCalls.append((mnemonic, operands))  
 return instructionCalls, labelLines  
  
 def \_assembleLine(self, line: [str, str], address: int = 0, labels: [str,] = tuple()) -> [AddressingMode, bytes, [[str, int]]]:  
 mnemonic, operands = line  
 operation = self.\_instructionSet.getOperationByMnemonic(mnemonic)  
 for addressingMode in self.\_instructionSet.operationAddressingModes(operation):  
 try:  
 assembledOperands, labelUses = addressingMode.assemble(operands, address, labels)  
 opcode = self.\_instructionSet.getOpcode(operation, addressingMode)  
 return addressingMode, bytes((opcode,)) + assembledOperands, labelUses  
 except AddressingMode.AddressingModeAssembleError:  
 pass  
 raise Assembler.AssemblerError(f"Could not identify addressing mode: '{mnemonic} {operands}'")  
  
 def assemble(self, assembly: str or [str,], startAddress: int = 0) -> bytes: *# TODO labels without placeholders* machineCode = bytes()  
 lines, labels = self.\_preprocessing(assembly)  
 labelUses = list()  
 labelAddresses = dict()  
 labelIdentifiers = self.labelIdentifiers + tuple(labels.values())  
 for line in range(len(lines)):  
 if line in labels:  
 labelAddresses[labels[line]] = len(machineCode)  
 addressingMode, lineMachineCode, lineLabelUses = self.\_assembleLine(lines[line], startAddress + line, labelIdentifiers)  
 for identifier, byte in lineLabelUses:  
 labelUses.append((len(machineCode) + 1 + byte, identifier, addressingMode))  
 machineCode += lineMachineCode  
 for address, label, addressingMode in labelUses:  
 if label in labelAddresses:  
 labelAddress = labelAddresses[label]  
 self.\_labels[label] = labelAddress  
 else:  
 labelAddress = self.\_labels[label]  
 assembledLabel = addressingMode.assembleLabel(labelAddress, address)  
 machineCode = machineCode[:address] + assembledLabel + machineCode[address + len(assembledLabel):]  
 return machineCode

### processor.py

from instruction\_set import InstructionSet  
from component import Component  
from general import slice\_to\_tuple  
  
class Processor(Component):  
 class InvalidRegisterError(Exception):  
 pass  
  
 def \_\_init\_\_(self, instructionSet: InstructionSet, registerValues: [bytes,] or bytes = tuple(),

pinValues: [bool or int,] or bytes = tuple(), connections: [[Component, [[int or str, int or str],]],] = tuple()):  
 if not isinstance(instructionSet, InstructionSet):  
 raise TypeError(f"A processor's instruction set must inherit from InstructionSet ({instructionSet} does not)")  
 self.\_instructionSet = InstructionSet  
 self.\_registers = {  
 "PC" : bytes(2),  
 "MAR" : bytes(2),  
 "MBR" : bytes(1),  
 "IR" : bytes(1),  
 "P" : bytes(1),  
 "TCU" : bytes(1),  
 "S" : bytes(1),  
 "A" : bytes(1),  
 "X" : bytes(1),  
 "Y" : bytes(1)  
 }  
 super().\_\_init\_\_(  
 (  
 "VPB", "RDY", "PHI1O", "IRQB", "MLB",  
 "NMIB", "SYNC", "VDD", "A0", "A1",  
 "A2", "A3", "A4", "A5", "A6",  
 "A7", "A8", "A9", "A10", "A11",  
 "VSS", "A12", "A13", "A14", "A15",  
 "D7", "D6", "D5", "D4", "D3",  
 "D2", "D1", "D0", "RWB", "NC",  
 "BE", "PHI2", "SOB", "PHI2O", "RESB"  
 ),  
 pinValues, connections  
 )  
 if registerValues:  
 self.setRegisters(slice(None), registerValues)  
  
 @property  
 def registers(self) -> [str,]:  
 return tuple(self.\_registers.keys())  
  
 def registerSelect(self, register: str or int) -> str:  
 if isinstance(register, int):  
 return self.registers[register]  
 else:  
 register = str(register)  
 if register in self.\_registers:  
 return register  
 raise Processor.InvalidRegisterError(f"No register called {register}. Registers are: {self.registers}")  
  
 def registersSelect(self, registers: [int or str,] or slice) -> [str,]:  
 if isinstance(registers, slice):  
 registers = slice\_to\_tuple(registers, len(self.\_registers))  
 registerNames = list()  
 for register in registers:  
 registerNames.append(self.registerSelect(register))  
 return tuple(registerNames)  
  
 def getRegister(self, register: str or int) -> bytes:  
 return self.\_registers[self.registerSelect(register)]  
  
 def setRegister(self, register: str or int, value: bytes):  
 register = self.registerSelect(register)  
 if not isinstance(value, bytes):  
 raise TypeError(f"Registers store binary data as bytes not {type(value).\_\_name\_\_} ({value})")  
 if len(value) != len(self.\_registers[register]):  
 raise ValueError(

f"{register} is a {len(self.\_registers[register])}-byte register so cannot be set " +

f"with a {len(value)}-byte value ({value})"

)  
 self.\_registers[register] = value  
  
 def getRegisters(self, registers: [int or str,] or slice) -> [bytes,]:  
 registers = self.registersSelect(registers)  
 values = list()  
 for register in registers:  
 values.append(self.\_registers[register])  
 return tuple(values)  
  
 def setRegisters(self, registers: [int or str,] or slice, values: [bytes,] or bytes):  
 prevValues = self.getRegisters(slice(None))  
 registers = self.registersSelect(registers)  
 try:  
 if isinstance(values, bytes):  
 pointer = 0  
 for register in registers:  
 nextPointer = pointer + len(self.\_registers[register])  
 self.setRegister(register, values[pointer : nextPointer])  
 pointer = nextPointer  
 else:  
 if len(values) != len(registers):  
 raise ValueError(f"{len(registers)} registers cannot be set with {len(values)} values ({values})")  
 for index in range(len(registers)):  
 self.setRegister(registers[index], values[index])  
 except Exception as error:  
 self.setRegisters(slice(None), prevValues)  
 raise error  
  
 @property  
 def state(self) -> {str: any}:  
 state = Component.state.\_\_get\_\_(self)  
 state["registers"] = self.getRegisters(slice(None))  
 return state  
  
 @state.setter  
 def state(self, state: {str: any}):  
 prevState = self.state  
 Component.state.\_\_set\_\_(self, state)  
 try:  
 registersState = state["registers"]  
 except KeyError:  
 raise Component.StateError("registers", state)  
 try:  
 self.setRegisters(slice(None), registersState)  
 except Exception as error:  
 self.state = prevState  
 raise error  
  
 @state.deleter  
 def state(self):  
 Component.state.\_\_delete\_\_(self)  
 for register in self.\_registers:  
 self.\_registers[register] = bytes(len(self.\_registers[register]))  
  
 def execute(self): *# TODO  
 # if TCU == \x00  
 # currentInstruction = instructionSet.getInstruction(IR)  
 # else  
 # currentInstruction[0].execute(self, currentInstruction[1])  
 # if TCU == \x00  
 # currentInstruction = (None, None)  
 # TCU -= 1  
 # TCU += 1* pass  
  
 def response(self): *# TODO  
 # # process pins  
 # if clock has changed  
 # self.execute()* pass

### memory.py

from component import Component  
from abc import ABC, abstractmethod  
from general import bytes\_to\_tuple, slice\_to\_tuple  
  
class Memory(Component, ABC):  
 class InvalidMemoryAddressError(Exception):  
 pass  
  
 def validateAddress(self, address: int or bytes) -> int:  
 if isinstance(address, bytes):  
 address = int.from\_bytes(address, "big")  
 elif not isinstance(address, int):  
 raise TypeError(f"Address must be int or bytes, not type {type(address).\_\_name\_\_} ({address})")  
 if 0 <= address < len(self):  
 return address  
 else:  
 raise ValueError(f"Address {address} is out of range")  
  
 def validateAddresses(self, addresses: [int or bytes,] or slice) -> [int,]:  
 if isinstance(addresses, slice):  
 addresses = slice\_to\_tuple(addresses, len(self))  
 validatedAddresses = list()  
 for address in addresses:  
 validatedAddresses.append(self.validateAddress(address))  
 return tuple(validatedAddresses)  
  
 @abstractmethod  
 def \_\_len\_\_(self) -> int:  
 pass  
  
 @abstractmethod  
 def read(self, address: int or bytes) -> bytes:  
 pass  
  
 @abstractmethod  
 def write(self, address: int or bytes, value: bytes):  
 pass  
  
 @abstractmethod  
 def readAddresses(self, addresses: [int or bytes,] or slice) -> [bytes,]:  
 pass  
  
 @abstractmethod  
 def writeAddresses(self, addresses: [int or bytes,] or slice, values: [bytes,] or bytes):  
 pass  
  
 @property  
 def data(self) -> bytes:  
 return self.readAddresses(slice(None))  
  
 @data.setter  
 def data(self, data: bytes or [bytes,]):  
 self.writeAddresses(slice(None), data)  
  
 @data.deleter  
 def data(self):  
 self.writeAddresses(slice(None), bytes())  
  
 def \_\_getitem\_\_(self, addresses: int or bytes or [int or bytes,] or slice) -> bytes or [bytes,]:  
 if isinstance(addresses, int) or isinstance(addresses, bytes):  
 return self.read(addresses)  
 else:  
 return self.readAddresses(addresses)  
  
 def \_\_setitem\_\_(self, addresses: int or bytes or [int or bytes,] or slice, values: bytes or [bytes,]):  
 if isinstance(addresses, int) or isinstance(addresses, bytes):  
 return self.write(addresses, values)  
 else:  
 return self.writeAddresses(addresses, values)  
  
 def save(self, fileName: str):  
 with open(fileName, "wb") as file:  
 file.write(self.data)  
  
 def load(self, fileName: str):  
 with open(fileName, "rb") as file:  
 self.data = file.read(len(self))  
  
 @property  
 def state(self) -> {str: any}:  
 state = Component.state.\_\_get\_\_(self)  
 state["data"] = self.data  
 return state  
  
 @state.setter  
 def state(self, state: {str: any}):  
 prevState = self.state  
 Component.state.\_\_set\_\_(self, state)  
 try:  
 dataState = state["data"]  
 except KeyError:  
 raise Component.StateError("data", state)  
 try:  
 self.data = dataState  
 except Exception as error:  
 self.state = prevState  
 raise error  
  
 @state.deleter  
 def state(self):  
 Component.state.\_\_delete\_\_(self)  
 del self.data  
  
class SpecificMemory(Memory):  
 def \_\_init\_\_(self, pins: [str,] = None, data: [bytes,] or bytes or str = bytes(), pinValues: [bool or int,] or bytes = bytes(), connections: [[Component, [[int or str, int or str],]],] = tuple()):  
 self.\_data = bytes()  
 super().\_\_init\_\_(pins, pinValues, connections)  
 if data is not None:  
 if len(data) > 0:  
 if isinstance(data, str):  
 self.load(data)  
 else:  
 self.writeAddresses(slice(len(data)), data)  
  
 @property  
 def data(self) -> bytes:  
 return self.\_data  
  
 @data.deleter  
 def data(self):  
 self.\_data = bytes(2 \*\* 15)  
  
 def \_\_len\_\_(self) -> int:  
 return len(self.\_data)  
  
 def read(self, address: int or bytes) -> bytes:  
 address = self.validateAddress(address)  
 return self.\_data[address : address + 1]  
  
 def write(self, address: int or bytes, value: bytes):  
 if not isinstance(value, bytes):  
 raise TypeError(f"Can only write bytes type data to memory not {type(value).\_\_name\_\_} ({value})")  
 if len(value) != 1:  
 raise ValueError(f"Memory addresses of {type(self).\_\_name\_\_} only store one byte")  
 address = self.validateAddress(address)  
 self.\_data = self.\_data[:address] + value + self.\_data[address + 1:]  
  
 def readAddresses(self, addresses: [int or bytes,] or slice) -> [bytes,]:  
 addresses = self.validateAddresses(addresses)  
 data = list()  
 for address in addresses:  
 data.append(self.read(address))  
 return tuple(data)  
  
 def writeAddresses(self, addresses: [int or bytes,] or slice, values: [bytes,] or bytes):  
 addresses = self.validateAddresses(addresses)  
 for index in range(len(addresses)):  
 self.write(addresses[index], values[index : index + 1])  
  
 def response(self):  
 high, low = self.getPinsStates((28, 14))  
 self.makePinsPassive(slice(None))  
 if self.getPin(20) == high[0]:  
 addressPins = 10, 9, 8, 7, 6, 5, 4, 3, 25, 24, 21, 23, 2, 26, 1  
 dataPins = 11, 12, 13, 15, 16, 17, 18, 19  
 address = 0  
 for bit in range(15):  
 address += self.getPin(addressPins[bit]) \* (2 \*\* bit)  
 modePins = self.getPins((22, 27))  
 if modePins == (high[0], low[0]):  
 data = 0  
 for bit in range(8):  
 data += self.getPin(dataPins[bit] \* (2 \*\* bit))  
 self.write(address, bytes([data]))  
 elif modePins == (low[0], high[0]):  
 data = bytes\_to\_tuple(self.read(address))[::-1]  
 for bit in range(8):  
 if data[bit]:  
 self.setPinState(dataPins[bit], high)  
 else:  
 self.setPinState(dataPins[bit], low)  
  
class ReadOnlyMemory(SpecificMemory):  
 def \_\_init\_\_(self, data: [bytes,] or bytes or str = tuple(), pinValues: [bool or int,] or bytes = tuple(), connections: [[Component, [[int or str, int or str],]],] = tuple()):  
 super().\_\_init\_\_(  
 (  
 "A14", "A12", "A7", "A6", "A5", "A4", "A3",  
 "A2", "A1", "A0", "I/O0", "I/O1", "I/O2", "GND",  
 "I/O3", "I/O4", "I/O5", "I/O6", "I/O7", "CEB", "A10",  
 "OEB", "A11", "A9", "A8", "A13", "WEB", "VCC"  
 ),  
 data, pinValues, connections)  
  
class RandomAccessMemory(SpecificMemory):  
 def \_\_init\_\_(self, data: [bytes,] or bytes or str = tuple(), pinValues: [bool or int,] or bytes = tuple(), connections: [[Component, [[int or str, int or str],]],] = tuple()):  
 super().\_\_init\_\_(  
 (  
 "A14", "A12", "A7", "A6", "A5", "A4", "A3",  
 "A2", "A1", "A0", "I/O0", "I/O1", "I/O2", "Vss",  
 "I/O3", "I/O4", "I/O5", "I/O6", "I/O7", "CSB", "A10",  
 "OEB", "A11", "A9", "A8", "A13", "WEB", "Vcc"  
 ),  
 data, pinValues, connections)  
  
 def response(self):  
 if not self.getPin(28):  
 del self.data  
 self.makePinsPassive(slice(None))  
 else:  
 super().response()

### additional\_hardware.py

from component import Component  
from general import int\_to\_bool, BinaryElectric as BinElec  
  
class PowerSupply(Component):  
 def \_\_init\_\_(self, hasPower: bool or int = True, pinValues: [bool or int,] or bytes = tuple(),

connections: [[Component, [[int or str, int or str],]],] = tuple()):  
 self.\_power = False  
 super().\_\_init\_\_(  
 ("Power", "Ground"),  
 pinValues, connections  
 )  
 self.\_power = int\_to\_bool(hasPower)  
  
 @property  
 def power(self) -> bool:  
 return self.\_power  
  
 @power.setter  
 def power(self, hasPower: bool or int):  
 self.\_power = int\_to\_bool(hasPower)  
  
 def togglePower(self):  
 self.\_power = not self.\_power  
  
 def turnOn(self):  
 self.\_power = True  
  
 def turnOff(self):  
 self.\_power = False  
  
 @property  
 def state(self) -> {str: any}:  
 state = Component.state.\_\_get\_\_(self)  
 state["power"] = self.\_power  
 return state  
  
 @state.setter  
 def state(self, state: {str: any}):  
 prevState = self.state  
 Component.state.\_\_set\_\_(self, state)  
 try:  
 powerState = state["power"]  
 except KeyError:  
 raise Component.StateError("power", state)  
 try:  
 self.power = powerState  
 except Exception as error:  
 self.state = prevState  
 raise error  
  
 @state.deleter  
 def state(self):  
 Component.state.\_\_delete\_\_(self)  
 self.makePinsActive((1, 2))  
 self.\_power = False  
  
 def response(self):  
 self.setPinsStates((1, 2), ((self.\_power, True), (False, True)))  
  
class Clock(Component):  
 def \_\_init\_\_(self, output: bool or int = False, pinValues: [bool or int,] or bytes = tuple(),  
 connections: [[Component, [[int or str, int or str],]],] = tuple()):  
 self.\_output = False  
 super().\_\_init\_\_(  
 ("N/C", "GND", "VCC", "Output"),  
 pinValues, connections  
 )  
 self.\_output = int\_to\_bool(output)  
  
 @property  
 def output(self) -> bool:  
 return self.\_output  
  
 @output.setter  
 def output(self, output: bool):  
 self.\_output = int\_to\_bool(output)  
  
 def step(self):  
 self.\_output = not self.\_output  
  
 @property  
 def state(self) -> {str: any}:  
 state = Component.state.\_\_get\_\_(self)  
 state["output"] = self.\_output  
 return state  
  
 @state.setter  
 def state(self, state: {str: any}):  
 prevState = self.state  
 Component.state.\_\_set\_\_(self, state)  
 try:  
 outputState = state["output"]  
 except KeyError:  
 raise Component.StateError("output", state)  
 try:  
 self.output = outputState  
 except Exception as error:  
 self.state = prevState  
 raise error  
  
 @state.deleter  
 def state(self):  
 Component.state.\_\_delete\_\_(self)  
 self.\_output = False  
  
 def response(self):  
 high, low = self.getPinsStates(("VCC", "GND"))  
 self.makePinsPassive(slice(None))  
 if self.\_output:  
 self.setPinState("Output", high)  
 else:  
 self.setPinState("Output", low)  
  
class QuadNANDGate(Component):  
 def \_\_init\_\_(self, pinValues: [bool or int,] or bytes = tuple(),

connections: [[Component, [[int or str, int or str],]],] = tuple()):  
 super().\_\_init\_\_(  
 (  
 "A1", "B1", "Y1", "A2", "B2", "Y2", "GND",  
 "Y3", "A3", "B3", "Y4", "A4", "B4", "VCC"  
 ),  
 pinValues, connections  
 )  
  
 def response(self):  
 high, low = self.getPinsStates(("VCC", "GND"))  
 self.makePinsPassive(slice(None))  
 for gate in range(1, 5):  
 value = self.getPin(f"A{gate}") + self.getPin(f"B{gate}")  
 if value == 1:  
 self.setPinState(f"Y{gate}", high)  
 else:  
 self.setPinState(f"Y{gate}", low)  
  
class Button(Component):  
 def \_\_init\_\_(self, isPressed: bool or int = False, pinValues: [bool or int,] or bytes = tuple(),

connections: [[Component, [[int or str, int or str],]],] = tuple()):  
 self.\_pressed = False  
 super().\_\_init\_\_(4, pinValues, connections)  
 self.\_pressed = int\_to\_bool(isPressed)  
  
 @property  
 def pressed(self) -> bool:  
 return self.\_pressed  
  
 @pressed.setter  
 def pressed(self, isPressed: bool or int):  
 self.\_pressed = int\_to\_bool(isPressed)  
  
 def togglePress(self):  
 self.\_pressed = not self.\_pressed  
  
 def press(self):  
 self.\_pressed = True  
  
 def unpress(self):  
 self.\_pressed = False  
  
 @property  
 def state(self) -> {str: any}:  
 state = Component.state.\_\_get\_\_(self)  
 state["pressed"] = self.\_pressed  
 return state  
  
 @state.setter  
 def state(self, state: {str: any}):  
 prevState = self.state  
 Component.state.\_\_set\_\_(self, state)  
 try:  
 pressedState = state["pressed"]  
 except KeyError:  
 raise Component.StateError("pressed", state)  
 try:  
 self.pressed = pressedState  
 except Exception as error:  
 self.state = prevState  
 raise error  
  
 @state.deleter  
 def state(self):  
 Component.state.\_\_delete\_\_(self)  
 self.\_pressed = False  
  
 def response(self):  
 pin1, pin2, pin3, pin4 = self.getPinsStates(slice(None))  
 side1 = BinElec.combine(pin1, pin2)  
 side2 = BinElec.combine(pin3, pin4)  
 if self.\_pressed:  
 side1 = side2 = BinElec.combine(side1, side2)  
 self.setPinsStates((1, 2), side1)  
 self.setPinsStates((3, 4), side2)  
  
class Resistor(Component):  
 def \_\_init\_\_(self, pinValues: [bool or int,] or bytes = tuple(),  
 connections: [[Component, [[int or str, int or str],]],] = tuple()):  
 super().\_\_init\_\_(2, pinValues, connections)  
  
 def response(self):  
 pin1, pin2 = self.getPinsStates(slice(None))  
 if pin1[1] != pin2[1]:  
 if pin1[1]:  
 self.setPinState(2, pin1)  
 else:  
 self.setPinState(1, pin2)

### instruction\_set\_65C02/addressing\_modes.py

from processor import Processor  
from instruction\_set import AddressingMode  
  
class AddressingModes: *# TODO* class Absolute(AddressingMode): *# a* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class AbsoluteIndexedIndirect(AddressingMode): *# (a,x)* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class XIndexedAbsolute(AddressingMode): *# a,x* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class YIndexedAbsolute(AddressingMode): *# a,y* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class AbsoluteIndirect(AddressingMode): *# (a)* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class Accumulator(AddressingMode): *# A* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class Immediate(AddressingMode): *# #* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class Implied(AddressingMode): *# i* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 if operandString == "":  
 return bytes(), tuple()  
 else:  
 raise AddressingMode.AddressingModeAssembleError("Implied addressing mode takes no operands")  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 return True, bytes()  
  
 class Relative(AddressingMode): *# r* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class Stack(AddressingMode): *# s* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class ZeroPage(AddressingMode): *# zp* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class ZeroPageIndexedIndirect(AddressingMode): *# (zp,x)* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class XIndexedZeroPage(AddressingMode): *# zp,x* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class YIndexedZeroPage(AddressingMode): *# zp,y* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class ZeroPageIndirect(AddressingMode): *# (zp)* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class ZeroPageIndirectIndexed(AddressingMode): *# (zp),y* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 class BranchBit(AddressingMode): *# zp,r* @staticmethod  
 def assemble(operandString: str, address: int = 0, labels: [str,] = tuple()) -> [bytes, [[int, str],]]:  
 pass  
  
 @staticmethod  
 def fetchOperand(processor: Processor, fetchCount: int) -> [bool, bytes]:  
 pass  
  
 @staticmethod  
 def assembleLabel(labelAddress: int, instructionAddress: int) -> bytes:  
 return AddressingMode.LabelModes.relativeLabel8Bit(labelAddress, instructionAddress)

### instruction\_set\_65C02/operations.py

from processor import Processor  
from instruction\_set import Operation, AddressingMode  
  
def RMB(bit: int, processor: Processor, addressingMode: AddressingMode):  
 pass  
  
def SMB(bit: int, processor: Processor, addressingMode: AddressingMode):  
 pass  
  
def BBR(bit: int, processor: Processor, addressingMode: AddressingMode):  
 pass  
  
def BBS(bit: int, processor: Processor, addressingMode: AddressingMode):  
 pass  
  
class Operations: *# TODO* class BRK(Operation):  
 mnemonic = "BRK"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BPL(Operation):  
 mnemonic = "BPL"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class JSR(Operation):  
 mnemonic = "JSR"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BMI(Operation):  
 mnemonic = "BMI"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class RTI(Operation):  
 mnemonic = "RTI"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BVC(Operation):  
 mnemonic = "BVC"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class RTS(Operation):  
 mnemonic = "RTS"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BVS(Operation):  
 mnemonic = "BVS"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BRA(Operation):  
 mnemonic = "BRA"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BCC(Operation):  
 mnemonic = "BCC"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class LDY(Operation):  
 mnemonic = "LDY"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BCS(Operation):  
 mnemonic = "BCS"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class CPY(Operation):  
 mnemonic = "CPY"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BNE(Operation):  
 mnemonic = "BNE"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class CPX(Operation):  
 mnemonic = "CPX"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BEQ(Operation):  
 mnemonic = "BEQ"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class ORA(Operation):  
 mnemonic = "ORA"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class AND(Operation):  
 mnemonic = "AND"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class EOR(Operation):  
 mnemonic = "EOR"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class ADC(Operation):  
 mnemonic = "ADC"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class STA(Operation):  
 mnemonic = "STA"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class LDA(Operation):  
 mnemonic = "LDA"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class CMP(Operation):  
 mnemonic = "CMP"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class SBC(Operation):  
 mnemonic = "SBC"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class LDX(Operation):  
 mnemonic = "LDX"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class TSB(Operation):  
 mnemonic = "TSB"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class TRB(Operation):  
 mnemonic = "TRB"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BIT(Operation):  
 mnemonic = "BIT"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class STZ(Operation):  
 mnemonic = "STZ"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class STY(Operation):  
 mnemonic = "STY"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class ASL(Operation):  
 mnemonic = "ASL"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class ROL(Operation):  
 mnemonic = "ROL"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class LSR(Operation):  
 mnemonic = "LSR"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class ROR(Operation):  
 mnemonic = "ROR"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class STX(Operation):  
 mnemonic = "STX"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class DEC(Operation):  
 mnemonic = "DEC"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class INC(Operation):  
 mnemonic = "INC"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class RMB0(Operation):  
 mnemonic = "RMB0"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 RMB(0, processor, addressingMode)  
  
 class RMB1(Operation):  
 mnemonic = "RMB1"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 RMB(1, processor, addressingMode)  
  
 class RMB2(Operation):  
 mnemonic = "RMB2"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 RMB(2, processor, addressingMode)  
  
 class RMB3(Operation):  
 mnemonic = "RMB3"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 RMB(3, processor, addressingMode)  
  
 class RMB4(Operation):  
 mnemonic = "RMB4"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 RMB(4, processor, addressingMode)  
  
 class RMB5(Operation):  
 mnemonic = "RMB5"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 RMB(5, processor, addressingMode)  
  
 class RMB6(Operation):  
 mnemonic = "RMB6"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 RMB(6, processor, addressingMode)  
  
 class RMB7(Operation):  
 mnemonic = "RMB7"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 RMB(7, processor, addressingMode)  
  
 class SMB0(Operation):  
 mnemonic = "SMB0"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 SMB(0, processor, addressingMode)  
  
 class SMB1(Operation):  
 mnemonic = "SMB1"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 SMB(1, processor, addressingMode)  
  
 class SMB2(Operation):  
 mnemonic = "SMB2"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 SMB(2, processor, addressingMode)  
  
 class SMB3(Operation):  
 mnemonic = "SMB3"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 SMB(3, processor, addressingMode)  
  
 class SMB4(Operation):  
 mnemonic = "SMB4"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 SMB(4, processor, addressingMode)  
  
 class SMB5(Operation):  
 mnemonic = "SMB5"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 SMB(5, processor, addressingMode)  
  
 class SMB6(Operation):  
 mnemonic = "SMB6"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 SMB(6, processor, addressingMode)  
  
 class SMB7(Operation):  
 mnemonic = "SMB7"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 SMB(7, processor, addressingMode)  
  
 class PHP(Operation):  
 mnemonic = "PHP"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class CLC(Operation):  
 mnemonic = "CLC"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class PLP(Operation):  
 mnemonic = "PLP"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class SEC(Operation):  
 mnemonic = "SEC"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class PHA(Operation):  
 mnemonic = "PHA"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class CLI(Operation):  
 mnemonic = "CLI"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class PLA(Operation):  
 mnemonic = "PLA"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class SEI(Operation):  
 mnemonic = "SEI"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class DEY(Operation):  
 mnemonic = "DEY"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class TYA(Operation):  
 mnemonic = "TYA"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class TAY(Operation):  
 mnemonic = "TAY"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class CLV(Operation):  
 mnemonic = "CLV"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class INY(Operation):  
 mnemonic = "INY"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class CLD(Operation):  
 mnemonic = "CLD"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class INX(Operation):  
 mnemonic = "INX"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class SED(Operation):  
 mnemonic = "SED"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class PHY(Operation):  
 mnemonic = "PHY"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class PLY(Operation):  
 mnemonic = "PLY"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class TXA(Operation):  
 mnemonic = "TXA"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class TXS(Operation):  
 mnemonic = "TXS"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class TAX(Operation):  
 mnemonic = "TAX"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class TSX(Operation):  
 mnemonic = "TSX"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class DEX(Operation):  
 mnemonic = "DEX"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class PHX(Operation):  
 mnemonic = "PHX"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class NOP(Operation):  
 mnemonic = "NOP"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 processor.setRegister("TCU", bytes(1))  
  
 class PLX(Operation):  
 mnemonic = "PLX"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class WAI(Operation):  
 mnemonic = "WAI"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class STP(Operation):  
 mnemonic = "STP"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class JMP(Operation):  
 mnemonic = "JMP"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 pass  
  
 class BBR0(Operation):  
 mnemonic = "BBR0"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBR(0, processor, addressingMode)  
  
 class BBR1(Operation):  
 mnemonic = "BBR1"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBR(1, processor, addressingMode)  
  
 class BBR2(Operation):  
 mnemonic = "BBR2"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBR(2, processor, addressingMode)  
  
 class BBR3(Operation):  
 mnemonic = "BBR3"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBR(3, processor, addressingMode)  
  
 class BBR4(Operation):  
 mnemonic = "BBR4"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBR(4, processor, addressingMode)  
  
 class BBR5(Operation):  
 mnemonic = "BBR5"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBR(5, processor, addressingMode)  
  
 class BBR6(Operation):  
 mnemonic = "BBR6"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBR(6, processor, addressingMode)  
  
 class BBR7(Operation):  
 mnemonic = "BBR7"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBR(7, processor, addressingMode)  
  
 class BBS0(Operation):  
 mnemonic = "BBS0"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBS(0, processor, addressingMode)  
  
 class BBS1(Operation):  
 mnemonic = "BBS1"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBS(1, processor, addressingMode)  
  
 class BBS2(Operation):  
 mnemonic = "BBS2"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBS(2, processor, addressingMode)  
  
 class BBS3(Operation):  
 mnemonic = "BBS3"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBS(3, processor, addressingMode)  
  
 class BBS4(Operation):  
 mnemonic = "BBS4"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBS(4, processor, addressingMode)  
  
 class BBS5(Operation):  
 mnemonic = "BBS5"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBS(5, processor, addressingMode)  
  
 class BBS6(Operation):  
 mnemonic = "BBS6"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBS(6, processor, addressingMode)  
  
 class BBS7(Operation):  
 mnemonic = "BBS7"  
  
 @staticmethod  
 def execute(processor: Processor, addressingMode: AddressingMode):  
 BBS(7, processor, addressingMode)

### instruction\_set\_65C02/instructions.py

from instruction\_set\_65C02.operations import Operations  
from instruction\_set\_65C02.addressing\_modes import AddressingModes  
  
instructions = (  
 (Operations.BRK, AddressingModes.Stack),  
 (Operations.ORA, AddressingModes.ZeroPageIndexedIndirect),  
 None,  
 None,  
 (Operations.TSB, AddressingModes.ZeroPage),  
 (Operations.ORA, AddressingModes.ZeroPage),  
 (Operations.ASL, AddressingModes.ZeroPage),  
 (Operations.RMB0, AddressingModes.ZeroPage),  
 (Operations.PHP, AddressingModes.Stack),  
 (Operations.ORA, AddressingModes.Immediate),  
 (Operations.ASL, AddressingModes.Accumulator),  
 None,  
 (Operations.TSB, AddressingModes.Absolute),  
 (Operations.ORA, AddressingModes.Absolute),  
 (Operations.ASL, AddressingModes.Absolute),  
 (Operations.BBR0, AddressingModes.BranchBit),  
 (Operations.BPL, AddressingModes.Relative),  
 (Operations.ORA, AddressingModes.ZeroPageIndirectIndexed),  
 (Operations.ORA, AddressingModes.ZeroPageIndirect),  
 None,  
 (Operations.TRB, AddressingModes.ZeroPage),  
 (Operations.ORA, AddressingModes.XIndexedZeroPage),  
 (Operations.ASL, AddressingModes.XIndexedZeroPage),  
 (Operations.RMB1, AddressingModes.ZeroPage),  
 (Operations.CLC, AddressingModes.Implied),  
 (Operations.ORA, AddressingModes.YIndexedAbsolute),  
 (Operations.INC, AddressingModes.Accumulator),  
 None,  
 (Operations.TRB, AddressingModes.Absolute),  
 (Operations.ORA, AddressingModes.XIndexedAbsolute),  
 (Operations.ASL, AddressingModes.XIndexedAbsolute),  
 (Operations.BBR1, AddressingModes.BranchBit),  
 (Operations.JSR, AddressingModes.Absolute),  
 (Operations.AND, AddressingModes.ZeroPageIndexedIndirect),  
 None,  
 None,  
 (Operations.BIT, AddressingModes.ZeroPage),  
 (Operations.AND, AddressingModes.ZeroPage),  
 (Operations.ROL, AddressingModes.ZeroPage),  
 (Operations.RMB2, AddressingModes.ZeroPage),  
 (Operations.PLP, AddressingModes.Stack),  
 (Operations.AND, AddressingModes.Immediate),  
 (Operations.ROL, AddressingModes.Accumulator),  
 None,  
 (Operations.BIT, AddressingModes.Absolute),  
 (Operations.AND, AddressingModes.Absolute),  
 (Operations.ROL, AddressingModes.Absolute),  
 (Operations.BBR2, AddressingModes.BranchBit),  
 (Operations.BMI, AddressingModes.Relative),  
 (Operations.AND, AddressingModes.ZeroPageIndirectIndexed),  
 (Operations.AND, AddressingModes.ZeroPageIndirect),  
 None,  
 (Operations.BIT, AddressingModes.XIndexedZeroPage),  
 (Operations.AND, AddressingModes.XIndexedZeroPage),  
 (Operations.ROL, AddressingModes.XIndexedZeroPage),  
 (Operations.RMB3, AddressingModes.ZeroPage),  
 (Operations.SEC, AddressingModes.Implied),  
 (Operations.AND, AddressingModes.YIndexedAbsolute),  
 (Operations.DEC, AddressingModes.Accumulator),  
 None,  
 (Operations.BIT, AddressingModes.XIndexedAbsolute),  
 (Operations.AND, AddressingModes.XIndexedAbsolute),  
 (Operations.ROL, AddressingModes.XIndexedAbsolute),  
 (Operations.BBR3, AddressingModes.BranchBit),  
 (Operations.RTI, AddressingModes.Stack),  
 (Operations.EOR, AddressingModes.ZeroPageIndexedIndirect),  
 None,  
 None,  
 None,  
 (Operations.EOR, AddressingModes.ZeroPage),  
 (Operations.LSR, AddressingModes.ZeroPage),  
 (Operations.RMB4, AddressingModes.ZeroPage),  
 (Operations.PHA, AddressingModes.Stack),  
 (Operations.EOR, AddressingModes.Immediate),  
 (Operations.LSR, AddressingModes.Accumulator),  
 None,  
 (Operations.JMP, AddressingModes.Absolute),  
 (Operations.EOR, AddressingModes.Absolute),  
 (Operations.LSR, AddressingModes.Absolute),  
 (Operations.BBR4, AddressingModes.BranchBit),  
 (Operations.BVC, AddressingModes.Relative),  
 (Operations.EOR, AddressingModes.ZeroPageIndirectIndexed),  
 (Operations.EOR, AddressingModes.ZeroPageIndirect),  
 None,  
 None,  
 (Operations.EOR, AddressingModes.XIndexedZeroPage),  
 (Operations.LSR, AddressingModes.XIndexedZeroPage),  
 (Operations.RMB5, AddressingModes.ZeroPage),  
 (Operations.CLI, AddressingModes.Implied),  
 (Operations.EOR, AddressingModes.YIndexedAbsolute),  
 (Operations.PHY, AddressingModes.Stack),  
 None,  
 None,  
 (Operations.EOR, AddressingModes.XIndexedAbsolute),  
 (Operations.LSR, AddressingModes.XIndexedAbsolute),  
 (Operations.BBR5, AddressingModes.BranchBit),  
 (Operations.RTS, AddressingModes.Stack),  
 (Operations.ADC, AddressingModes.ZeroPageIndexedIndirect),  
 None,  
 None,  
 (Operations.STZ, AddressingModes.ZeroPage),  
 (Operations.ADC, AddressingModes.ZeroPage),  
 (Operations.ROR, AddressingModes.ZeroPage),  
 (Operations.RMB6, AddressingModes.ZeroPage),  
 (Operations.PLA, AddressingModes.Stack),  
 (Operations.ADC, AddressingModes.Immediate),  
 (Operations.ROR, AddressingModes.Accumulator),  
 None,  
 (Operations.JMP, AddressingModes.AbsoluteIndirect),  
 (Operations.ADC, AddressingModes.Absolute),  
 (Operations.ROR, AddressingModes.Absolute),  
 (Operations.BBR6, AddressingModes.BranchBit),  
 (Operations.BVS, AddressingModes.Relative),  
 (Operations.ADC, AddressingModes.ZeroPageIndirectIndexed),  
 (Operations.ADC, AddressingModes.ZeroPageIndirect),  
 None,  
 (Operations.STZ, AddressingModes.XIndexedZeroPage),  
 (Operations.ADC, AddressingModes.XIndexedZeroPage),  
 (Operations.ROR, AddressingModes.XIndexedZeroPage),  
 (Operations.RMB7, AddressingModes.ZeroPage),  
 (Operations.SEI, AddressingModes.Implied),  
 (Operations.ADC, AddressingModes.YIndexedAbsolute),  
 (Operations.PLY, AddressingModes.Stack),  
 None,  
 (Operations.JMP, AddressingModes.AbsoluteIndexedIndirect),  
 (Operations.ADC, AddressingModes.XIndexedAbsolute),  
 (Operations.ROR, AddressingModes.XIndexedAbsolute),  
 (Operations.BBR7, AddressingModes.BranchBit),  
 (Operations.BRA, AddressingModes.Relative),  
 (Operations.STA, AddressingModes.ZeroPageIndexedIndirect),  
 None,  
 None,  
 (Operations.STY, AddressingModes.ZeroPage),  
 (Operations.STA, AddressingModes.ZeroPage),  
 (Operations.STX, AddressingModes.ZeroPage),  
 (Operations.SMB0, AddressingModes.ZeroPage),  
 (Operations.DEY, AddressingModes.Implied),  
 (Operations.BIT, AddressingModes.Immediate),  
 (Operations.TXA, AddressingModes.Implied),  
 None,  
 (Operations.STY, AddressingModes.Absolute),  
 (Operations.STA, AddressingModes.Absolute),  
 (Operations.STX, AddressingModes.Absolute),  
 (Operations.BBS0, AddressingModes.BranchBit),  
 (Operations.BCC, AddressingModes.Relative),  
 (Operations.STA, AddressingModes.ZeroPageIndirectIndexed),  
 (Operations.STA, AddressingModes.ZeroPageIndirect),  
 None,  
 (Operations.STY, AddressingModes.XIndexedZeroPage),  
 (Operations.STA, AddressingModes.XIndexedZeroPage),  
 (Operations.STX, AddressingModes.YIndexedZeroPage),  
 (Operations.SMB1, AddressingModes.ZeroPage),  
 (Operations.TYA, AddressingModes.Implied),  
 (Operations.STA, AddressingModes.YIndexedAbsolute),  
 (Operations.TXS, AddressingModes.Implied),  
 None,  
 (Operations.STZ, AddressingModes.Absolute),  
 (Operations.STA, AddressingModes.XIndexedAbsolute),  
 (Operations.STZ, AddressingModes.XIndexedAbsolute),  
 (Operations.BBS1, AddressingModes.BranchBit),  
 (Operations.LDY, AddressingModes.Immediate),  
 (Operations.LDA, AddressingModes.ZeroPageIndexedIndirect),  
 (Operations.LDX, AddressingModes.Immediate),  
 None,  
 (Operations.LDY, AddressingModes.ZeroPage),  
 (Operations.LDA, AddressingModes.ZeroPage),  
 (Operations.LDX, AddressingModes.ZeroPage),  
 (Operations.SMB2, AddressingModes.ZeroPage),  
 (Operations.TAY, AddressingModes.Implied),  
 (Operations.LDA, AddressingModes.Immediate),  
 (Operations.TAX, AddressingModes.Implied),  
 None,  
 (Operations.LDY, AddressingModes.Absolute),  
 (Operations.LDA, AddressingModes.Absolute),  
 (Operations.LDX, AddressingModes.Absolute),  
 (Operations.BBS2, AddressingModes.BranchBit),  
 (Operations.BCS, AddressingModes.Relative),  
 (Operations.LDA, AddressingModes.ZeroPageIndirectIndexed),  
 (Operations.LDA, AddressingModes.ZeroPageIndirect),  
 None,  
 (Operations.LDY, AddressingModes.XIndexedZeroPage),  
 (Operations.LDA, AddressingModes.XIndexedZeroPage),  
 (Operations.LDX, AddressingModes.YIndexedZeroPage),  
 (Operations.SMB3, AddressingModes.ZeroPage),  
 (Operations.CLV, AddressingModes.Implied),  
 (Operations.LDA, AddressingModes.YIndexedAbsolute),  
 (Operations.TSX, AddressingModes.Implied),  
 None,  
 (Operations.LDY, AddressingModes.XIndexedAbsolute),  
 (Operations.LDA, AddressingModes.XIndexedAbsolute),  
 (Operations.LDX, AddressingModes.YIndexedAbsolute),  
 (Operations.BBS3, AddressingModes.BranchBit),  
 (Operations.CPY, AddressingModes.Immediate),  
 (Operations.CMP, AddressingModes.ZeroPageIndexedIndirect),  
 None,  
 None,  
 (Operations.CPY, AddressingModes.ZeroPage),  
 (Operations.CMP, AddressingModes.ZeroPage),  
 (Operations.DEC, AddressingModes.ZeroPage),  
 (Operations.SMB4, AddressingModes.ZeroPage),  
 (Operations.INY, AddressingModes.Implied),  
 (Operations.CMP, AddressingModes.Immediate),  
 (Operations.DEX, AddressingModes.Implied),  
 (Operations.WAI, AddressingModes.Implied),  
 (Operations.CPY, AddressingModes.Absolute),  
 (Operations.CMP, AddressingModes.Absolute),  
 (Operations.DEC, AddressingModes.Absolute),  
 (Operations.BBS4, AddressingModes.BranchBit),  
 (Operations.BNE, AddressingModes.Relative),  
 (Operations.CMP, AddressingModes.ZeroPageIndirectIndexed),  
 (Operations.CMP, AddressingModes.ZeroPageIndirect),  
 None,  
 None,  
 (Operations.CMP, AddressingModes.XIndexedZeroPage),  
 (Operations.DEC, AddressingModes.XIndexedZeroPage),  
 (Operations.SMB5, AddressingModes.ZeroPage),  
 (Operations.CLD, AddressingModes.Implied),  
 (Operations.CMP, AddressingModes.YIndexedAbsolute),  
 (Operations.PHX, AddressingModes.Stack),  
 (Operations.STP, AddressingModes.Implied),  
 None,  
 (Operations.CMP, AddressingModes.XIndexedAbsolute),  
 (Operations.DEC, AddressingModes.XIndexedAbsolute),  
 (Operations.BBS5, AddressingModes.BranchBit),  
 (Operations.CPX, AddressingModes.Immediate),  
 (Operations.SBC, AddressingModes.ZeroPageIndexedIndirect),  
 None,  
 None,  
 (Operations.CPX, AddressingModes.ZeroPage),  
 (Operations.SBC, AddressingModes.ZeroPage),  
 (Operations.INC, AddressingModes.ZeroPage),  
 (Operations.SMB6, AddressingModes.ZeroPage),  
 (Operations.INX, AddressingModes.Implied),  
 (Operations.SBC, AddressingModes.Immediate),  
 (Operations.NOP, AddressingModes.Implied),  
 None,  
 (Operations.CPX, AddressingModes.Absolute),  
 (Operations.SBC, AddressingModes.Absolute),  
 (Operations.INC, AddressingModes.Absolute),  
 (Operations.BBS6, AddressingModes.BranchBit),  
 (Operations.BEQ, AddressingModes.Relative),  
 (Operations.SBC, AddressingModes.ZeroPageIndirectIndexed),  
 (Operations.SBC, AddressingModes.ZeroPageIndirect),  
 None,  
 None,  
 (Operations.SBC, AddressingModes.XIndexedZeroPage),  
 (Operations.INC, AddressingModes.XIndexedZeroPage),  
 (Operations.SMB7, AddressingModes.ZeroPage),  
 (Operations.SED, AddressingModes.Implied),  
 (Operations.SBC, AddressingModes.YIndexedAbsolute),  
 (Operations.PLX, AddressingModes.Stack),  
 None,  
 None,  
 (Operations.SBC, AddressingModes.XIndexedAbsolute),  
 (Operations.INC, AddressingModes.XIndexedAbsolute),  
 (Operations.BBS7, AddressingModes.BranchBit)  
)

### instruction\_set\_65C02/opcode\_matrix.txt

A more readable representation of the 65C02 instructions than the instructions list in instruction\_set\_65C02/instructions.py. Stored within the project files to make it more easily accessible. Shown rotated 90° clockwise due to its size.

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| 0 | BRK | ORA | | | TSB | ORA | ASL | RMB0 | PHP | ORA | ASL | | TSB | ORA | ASL | BBR0 | 0 |

| | s | (zp,x) | | | zp | zp | zp | zp | s | # | A | | a | a | a | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| 1 | BPL | ORA | ORA | | TRB | ORA | ASL | RMB1 | CLC | ORA | INC | | TRB | ORA | ASL | BBR1 | 1 |

| | r | (zp),y | (zp) | | zp | zp,x | zp,x | zp | i | a,y | A | | a | a,x | a,x | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| 2 | JSR | AND | | | BIT | AND | ROL | RMB2 | PLP | AND | ROL | | BIT | AND | ROL | BBR2 | 2 |

| | a | (zp,x) | | | zp | zp | zp | zp | s | # | A | | a | a | a | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| 3 | BMI | AND | AND | | BIT | AND | ROL | RMB3 | SEC | AND | DEC | | BIT | AND | ROL | BBR3 | 3 |

| | r | (zp),y | (zp) | | zp,x | zp,x | zp,x | zp | i | a,y | A | | a,x | a,x | a,x | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| 4 | RTI | EOR | | | | EOR | LSR | RMB4 | PHA | EOR | LSR | | JMP | EOR | LSR | BBR4 | 4 |

| | s | (zp,x) | | | | zp | zp | zp | s | # | A | | a | a | a | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| 5 | BVC | EOR | EOR | | | EOR | LSR | RMB5 | CLI | EOR | PHY | | | EOR | LSR | BBR5 | 5 |

| | r | (zp),y | (zp) | | | zp,x | zp,x | zp | i | a,y | s | | | a,x | a,x | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| 6 | RTS | ADC | | | STZ | ADC | ROR | RMB6 | PLA | ADC | ROR | | JMP | ADC | ROR | BBR6 | 6 |

| | s | (zp,x) | | | zp | zp | zp | zp | s | # | A | | (a) | a | a | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| 7 | BVS | ADC | ADC | | STZ | ADC | ROR | RMB7 | SEI | ADC | PLY | | JMP | ADC | ROR | BBR7 | 7 |

| | r | (zp),y | (zp) | | zp,x | zp,x | zp,x | zp | i | a,y | s | | (a,x) | a,x | a,x | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| 8 | BRA | STA | | | STY | STA | STX | SMB0 | DEY | BIT | TXA | | STY | STA | STX | BBS0 | 8 |

| | r | (zp,x) | | | zp | zp | zp | zp | i | # | i | | a | a | a | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| 9 | BCC | STA | STA | | STY | STA | STX | SMB1 | TYA | STA | TXS | | STZ | STA | STZ | BBS1 | 9 |

| | r | (zp),y | (zp) | | zp,x | zp,x | zp,y | zp | i | a,y | i | | a | a,x | a,x | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| A | LDY | LDA | LDX | | LDY | LDA | LDX | SMB2 | TAY | LDA | TAX | | LDY | LDA | LDX | BBS2 | A |

| | # | (zp,x) | # | | zp | zp | zp | zp | i | # | i | | a | a | a | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| B | BCS | LDA | LDA | | LDY | LDA | LDX | SMB3 | CLV | LDA | TSX | | LDY | LDA | LDX | BBS3 | B |

| | r | (zp),y | (zp) | | zp,x | zp,x | zp,y | zp | i | a,y | i | | a,x | a,x | a,y | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| C | CPY | CMP | | | CPY | CMP | DEC | SMB4 | INY | CMP | DEX | WAI | CPY | CMP | DEC | BBS4 | C |

| | # | (zp,x) | | | zp | zp | zp | zp | i | # | i | i | a | a | a | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| D | BNE | CMP | CMP | | | CMP | DEC | SMB5 | CLD | CMP | PHX | STP | | CMP | DEC | BBS5 | D |

| | r | (zp),y | (zp) | | | zp,x | zp,x | zp | i | a,y | s | i | | a,x | a,x | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| E | CPX | SBC | | | CPX | SBC | INC | SMB6 | INX | SBC | NOP | | CPX | SBC | INC | BBS6 | E |

| | # | (zp,x) | | | zp | zp | zp | zp | i | # | i | | a | a | a | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| F | BEQ | SBC | SBC | | | SBC | INC | SMB7 | SED | SBC | PLX | | | SBC | INC | BBS7 | F |

| | r | (zp),y | (zp) | | | zp,x | zp,x | zp | i | a,y | s | | | a,x | a,x | zp,r | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | |

+---+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---+

### user\_interface.py

class UserInterface:  
 class UnknownResponseError(ValueError):  
 pass  
  
 @staticmethod  
 def format(data: any, indent: int = 0) -> str:  
 if isinstance(data, str):  
 return " " \* indent + data  
 elif isinstance(data, tuple) or isinstance(data, list):  
 formattedItems = list()  
 totalLength = 0  
 multiLine = False  
 for item in data:  
 formattedItem = UserInterface.format(item, indent + 2)  
 totalLength += len(formattedItem)  
 if "\n" in formattedItem:  
 multiLine = True  
 formattedItems.append(formattedItem)  
 if totalLength + indent > 64:  
 multiLine = True  
 output = " " \* indent  
 if isinstance(data, tuple):  
 output += "("  
 else:  
 output += "["  
 if multiLine:  
 for item in formattedItems:  
 output += f"\n{item},"  
 output = output[:-1] + "\n" + " " \* indent  
 else:  
 for item in formattedItems:  
 output += item[indent + 2:] + ", "  
 output = output[:-2]  
 if isinstance(data, tuple):  
 return output + ")"  
 else:  
 return output + "]"  
 elif isinstance(data, dict):  
 formattedDict = dict()  
 maxKeyLength = 0  
 maxValueLength = 0  
 totalLength = 0  
 multiLine = False  
 for key in data:  
 formattedKey = UserInterface.format(key, indent + 2)[indent + 2:]  
 keyLength = len(formattedKey)  
 if keyLength > maxKeyLength:  
 maxKeyLength = keyLength  
 workingIndent = indent + 2  
 if "\n" in formattedKey:  
 workingIndent += 2  
 formattedValue = UserInterface.format(data[key], workingIndent)[workingIndent:]  
 valueLength = len(formattedValue)  
 if valueLength > maxValueLength:  
 maxValueLength = valueLength  
 totalLength += keyLength + valueLength - 1 - 2 \* indent  
 formattedDict[formattedKey] = formattedValue  
 if (not multiLine) and (maxKeyLength + maxValueLength + indent > 57 or totalLength > 66):  
 multiLine = True  
 output = " " \* indent + "{"  
 for key in formattedDict:  
 value = formattedDict[key]  
 if multiLine:  
 output += "\n" + " " \* (indent + 2)  
 output += key  
 if "\n" in key:  
 output += f"\n{indent + 2}: "  
 if "\n" in value:  
 output += f"\n{' ' \* (indent + 2)}{value}"  
 else:  
 output += value  
 else:  
 if multiLine:  
 output += f"{' ' \* (maxKeyLength - len(key))}"  
 output += " : "  
 if "\n" in value:  
 output += value.replace("\n", "\n" + " " \* (maxKeyLength + 3))  
 else:  
 output += value  
 output += ", "  
 output = output[:-2]  
 if "\n" in output:  
 output += "\n" + " " \* indent  
 return output + "}"  
 elif isinstance(data, bytes):  
 output = " " \* indent  
 for byte in range(len(data)):  
 value = data[byte]  
 for bit in range(7, -1, -1):  
 if value >= 2 \*\* bit:  
 value -= 2 \*\* bit  
 output += "1"  
 else:  
 output += "0"  
 if byte % 8 == 7:  
 output += "\n" + " " \* indent  
 else:  
 output += " "  
 return output[:-1]  
 else:  
 return " " \* indent + str(data)  
  
 @staticmethod  
 def output(data: any = ""):  
 print(UserInterface.format(data))  
  
 @staticmethod  
 def booleanInput(response: str, additionalResponses: {str: bool} = None) -> bool:  
 knownResponses = {"yes": True, "no": False,  
 "true": True, "false": False,  
 "y": True, "n": False,  
 "1": True, "0": False}  
 if additionalResponses is not None:  
 for newResponse in additionalResponses:  
 knownResponses[str(newResponse)] = bool(additionalResponses[newResponse])  
 try:  
 return knownResponses[response.lower().strip()]  
 except KeyError:  
 raise UserInterface.UnknownResponseError  
  
 @staticmethod  
 def input(prompt: any = "") -> str:  
 return input(UserInterface.format(prompt))  
  
 @staticmethod  
 def menu(options: (str,)) -> int:  
 UserInterface.output("")  
 for option in range(len(options)):  
 UserInterface.output(f"{option + 1}. {options[option]}")  
 while True:  
 choice = UserInterface.input("> ").lower().strip()  
 if choice.isnumeric():  
 choiceInt = int(choice)  
 if 1 <= choiceInt <= len(options):  
 return choiceInt  
 for option in range(len(options)):  
 if choice == options[option].lower().strip():  
 return option + 1  
 UserInterface.output("/!\ INVALID SELECTION")  
  
 @staticmethod  
 def loadFile(binary: bool = False) -> [bool, str or bytes]:  
 fileName = UserInterface.input("File: ")  
 try:  
 open(fileName).close()  
 except FileNotFoundError:  
 UserInterface.output("/!\ FILE NOT FOUND")  
 return False, None  
 if binary:  
 mode = "rb"  
 else:  
 mode = "r"  
 with open(fileName, mode) as file:  
 return True, file.read()  
  
 @staticmethod  
 def saveFile(data: str or bytes, binary: bool = False) -> bool:  
 fileName = UserInterface.input("File: ")  
 try:  
 open(fileName).close()  
 while True:  
 try:  
 overwrite = UserInterface.booleanInput(UserInterface.input("Overwrite? "), {"overwrite": True})  
 if overwrite:  
 break  
 else:  
 return False  
 except UserInterface.UnknownResponseError:  
 UserInterface.output("/!\ COULD NOT INTERPRET")  
 except FileNotFoundError:  
 pass  
 if binary:  
 mode = "wb"  
 else:  
 mode = "w"  
 with open(fileName, mode) as file:  
 file.write(data)  
 return True  
  
 @staticmethod  
 def console(\*\*kwargs):  
 for key, arg in kwargs.items():  
 keyWord = key  
 exec(f"{key} = arg")  
 UserInterface.output(f'{keyWord} = {arg}')  
 UserInterface.output("/END to exit console\n")  
 while True:  
 command = UserInterface.input()  
 if command.strip().lower() == "/end":  
 return  
 else:  
 try:  
 exec(command)  
 except Exception as error:  
 UserInterface.output(f"/!\ COULD NOT EXECUTE ({type(error).\_\_name\_\_}): {error}")  
  
 @staticmethod  
 def strToDict(string: str):  
 dictionary = dict()  
 string = str(string).strip()  
 if string[0] == "{":  
 string = string[1:]  
 else:  
 string += "}"  
 bracketDepth = 0  
 value = ""  
 key = ""  
 for character in string:  
 if (character == "," or character == "}") and bracketDepth == 0:  
 exec(f"dictionary[{key}] = {value.strip()}")  
 value = key = ""  
 elif character == ":":  
 key = value.strip()  
 value = ""  
 else:  
 value += character  
 if character == "(" or character == "[" or character == "}":  
 bracketDepth += 1  
 elif character == ")" or character == "]" or character == "}":  
 bracketDepth -= 1  
 return dictionary

### simulator.py

from user\_interface import UserInterface  
from assembler import Assembler  
from component import Component  
  
class Simulator:  
 def \_\_init\_\_(self, components: {str: Component} = None, step: callable = lambda components: None,

assemblers: {str: Assembler} = None):  
 self.\_components = dict()  
 if isinstance(components, dict):  
 for key in components:  
 component = components[key]  
 if Component.isComponent(component):  
 key = str(key).strip()  
 if not key.isalnum():  
 acceptable = True  
 for character in key:  
 if not character.isalnum():  
 if character not in (" ", "\_"):  
 acceptable = False  
 break  
 if not acceptable:  
 raise ValueError(f"Component identifier must not contain symbols ({key})")  
 self.\_components[key] = component  
 elif components is not None:  
 raise TypeError(  
 f"Components must be given as a dictionary where the key is an identifier used in menus ({components} is not valid)"   
 )  
 self.\_step = step  
 self.\_assemblers = dict()  
 if isinstance(assemblers, dict):  
 for key in assemblers:  
 assembler = assemblers[key]  
 if not isinstance(assembler, Assembler):  
 raise TypeError(  
 f"{assembler} of type {type(assembler).\_\_name\_\_} is not a valid assembler (does not inherit from Assembler)"   
 )  
 else:  
 self.\_assemblers[str(key)] = assembler  
 elif assemblers is not None:  
 raise TypeError(  
 f"Assemblers must be given as a dictionary where the key is an identifier used in menus ({assemblers} is not valid)"   
 )  
  
 @property  
 def componentDict(self) -> {str: Component}:  
 return self.\_components.copy()  
  
 @property  
 def components(self) -> [Component,]:  
 return tuple(self.\_components.values())  
  
 @property  
 def componentNames(self) -> [str,]:  
 return tuple(self.\_components.keys())  
  
 def identifyComponent(self, identifier: Component or str or int) -> str:  
 if isinstance(identifier, str):  
 return identifier  
 elif isinstance(identifier, Component):  
 return self.componentNames[self.components.index(identifier)]  
 elif isinstance(identifier, int):  
 return self.componentNames[identifier - 1]  
 else:  
 raise TypeError(f"Cannot identify Component using {identifier} of type {type(identifier).\_\_name\_\_}")  
  
 def getComponent(self, identifier: Component or str or int) -> Component:  
 return self.\_components[self.identifyComponent(identifier)]  
  
 def addComponent(self, name: str, component: Component):  
 self.\_components[name] = component  
  
 def removeComponent(self, identifier: Component or str or int) -> Component:  
 key = self.identifyComponent(identifier)  
 component = self.\_components[key]  
 self.\_components.pop(key)  
 return component  
  
 @property  
 def assemblerDict(self) -> {str: Assembler}:  
 return self.\_assemblers.copy()  
  
 @property  
 def assemblers(self) -> [Assembler,]:  
 return tuple(self.\_assemblers.values())  
  
 @property  
 def assemblerNames(self) -> [str,]:  
 return tuple(self.\_assemblers.keys())  
  
 def identifyAssembler(self, identifier: Assembler or str or int) -> str:  
 if isinstance(identifier, str):  
 return identifier  
 elif isinstance(identifier, Assembler):  
 return self.assemblerNames[self.assemblers.index(identifier)]  
 elif isinstance(identifier, int):  
 return self.assemblerNames[identifier - 1]  
 else:  
 raise TypeError(f"Cannot identify Assembler using {identifier} of type {type(identifier).\_\_name\_\_}")  
  
 def getAssembler(self, identifier: Assembler or str or int) -> Assembler:  
 return self.\_assemblers[self.identifyAssembler(identifier)]  
  
 def addAssembler(self, name: str, assembler: Assembler):  
 self.\_assemblers[name] = assembler  
  
 def removeAssembler(self, identifier: Assembler or str or int) -> Assembler:  
 key = self.identifyAssembler(identifier)  
 assembler = self.\_assemblers[key]  
 self.\_assemblers.pop(key)  
 return assembler  
  
 def step(self):  
 self.\_step(self.\_components)  
  
 def runSteps(self):  
 while True:  
 try:  
 steps = int(UserInterface.input("Steps: "))  
 for step in range(0, steps):  
 self.step()  
 return  
 except TypeError:  
 UserInterface.output("/!\ STEPS MUST BE AN INTEGER")  
  
 def stateMenu(self, component) -> bool:  
 component = self.getComponent(component)  
 UserInterface.output(component.state)  
 menuOptions = ("Raw state",  
 "Load state",  
 "Back",  
 "Return to menu")  
 while True:  
 choice = UserInterface.menu(menuOptions)  
 if choice == 1:  
 UserInterface.output(str(component.state))  
 elif choice == 2:  
 prevState = component.state  
 state = UserInterface.input("State = ")  
 try:  
 component.state = UserInterface.strToDict(state)  
 except Exception as error:  
 UserInterface.output(f"/!\ COULD NOT LOAD STATE ({type(error).\_\_name\_\_}): {error}")  
 component.state = prevState  
 elif choice == 3:  
 return False  
 elif choice == 4:  
 return True  
 else:  
 UserInterface.output("/!\ UNKNOWN MENU ERROR")  
  
 def componentMenu(self, component) -> bool:  
 menuOptions = ("State",  
 "Call method",  
 "Delete component",  
 "Component select",  
 "Return to menu")  
 while True:  
 choice = UserInterface.menu(menuOptions)  
 if choice == 1:  
 returnDepth = self.stateMenu(component)  
 if returnDepth:  
 return True  
 elif choice == 2:  
 componentName = self.identifyComponent(component)  
 component = self.\_components[componentName]  
 componentName = componentName.replace(" ", "\_")  
 if componentName[0].isnumeric():  
 componentName = "\_" + componentName  
 exec(f"UserInterface.console({componentName} = component)")  
 elif choice == 3:  
 component = self.removeComponent(component)  
 del component  
 return False  
 elif choice == 4:  
 return False  
 elif choice == 5:  
 return True  
 else:  
 UserInterface.output("/!\ UNKNOWN MENU ERROR")  
  
 def componentsSelect(self):  
 while True:  
 if len(self.\_components) == 0:  
 return  
 menuOptions = self.componentNames + ("Return to menu",)  
 choice = UserInterface.menu(menuOptions)  
 if choice == len(menuOptions):  
 return  
 elif choice > len(menuOptions) or choice < 1:  
 UserInterface.output("/!\ UNKNOWN MENU ERROR")  
 else:  
 returnDepth = self.componentMenu(self.getComponent(choice))  
 if returnDepth:  
 return  
  
 @staticmethod  
 def machineCodeMenu(machineCode) -> bool:  
 menuOptions = ("Save to file",  
 "Restart assembler",  
 "Return to menu")  
 while True:  
 choice = UserInterface.menu(menuOptions)  
 if choice == 1:  
 UserInterface.saveFile(machineCode, True)  
 elif choice == 2:  
 return False  
 elif choice == 3:  
 return True  
 else:  
 UserInterface.output("/!\ UNKNOWN MENU ERROR")  
  
 @staticmethod  
 def normaliseAssembly(assembly: str or [str,]) -> [str,]:  
 if isinstance(assembly, str):  
 if assembly[-1] == "\n":  
 assembly = assembly[:-1]  
 return assembly.split("\n")  
 else:  
 return list(assembly)  
  
 @staticmethod  
 def displayAssembly(assembly: str or [str,]):  
 assembly = Simulator.normaliseAssembly(assembly)  
 for line in range(len(assembly)):  
 UserInterface.output(f"{line + 1}: {assembly[line]}")  
  
 @staticmethod  
 def writeAssembly(existingAssembly: str or [str,] = tuple()) -> [str,]:  
 UserInterface.output("/UNDO to delete line\n/END to finish program\n")  
 assembly = list()  
 if existingAssembly:  
 assembly = Simulator.normaliseAssembly(existingAssembly)  
 Simulator.displayAssembly(assembly)  
 while True:  
 line = UserInterface.input(f"{len(assembly) + 1}: ")  
 if line.strip().lower() == "/end":  
 UserInterface.output()  
 return tuple(assembly)  
 elif line.strip().lower() == "/undo":  
 if len(assembly) == 0:  
 UserInterface.output("/!\ NO LINE TO UNDO")  
 else:  
 del assembly[len(assembly) - 1]  
 UserInterface.output("LINE UNDONE")  
 else:  
 assembly += Simulator.normaliseAssembly(line)  
  
 def assemblyMenu(self, assembly: str or [str,], assembler: Assembler) -> bool:  
 assembly = Simulator.normaliseAssembly(assembly)  
 Simulator.displayAssembly(assembly)  
 menuOptions = ("Save to file",  
 "Assemble",  
 "Continue writing",  
 "Discard")  
 while True:  
 choice = UserInterface.menu(menuOptions)  
 if choice == 1:  
 strAssembly = ""  
 for line in assembly:  
 strAssembly += line + "\n"  
 UserInterface.saveFile(strAssembly)  
 elif choice == 2:  
 assembler = self.getAssembler(assembler)  
 startAddress = UserInterface.input("Start address: ")  
 try:  
 startAddress = int(startAddress)  
 try:  
 machineCode = assembler.assemble(assembly, startAddress)  
 UserInterface.output(machineCode)  
 return Simulator.machineCodeMenu(machineCode)  
 except Exception as error:  
 UserInterface.output(f"/!\ COULD NOT ASSEMBLE ({type(error).\_\_name\_\_}): {error}")  
 except ValueError:  
 UserInterface.output("/!\ START ADDRESS MUST BE AN INTEGER")  
 elif choice == 3:  
 assembly = Simulator.writeAssembly(assembly)  
 elif choice == 4:  
 return False  
 else:  
 UserInterface.output("/!\ UNKNOWN MENU ERROR")  
  
 def assemblerMenu(self, assembler) -> bool:  
 menuOptions = ("Assemble from file",  
 "Write assembly",  
 "Remove instruction set",  
 "Instruction set select",  
 "Return to menu")  
 while True:  
 choice = UserInterface.menu(menuOptions)  
 if choice == 1:  
 success, assembly = UserInterface.loadFile()  
 if success:  
 assembly = Simulator.normaliseAssembly(assembly)  
 returnDepth = self.assemblyMenu(assembly, assembler)  
 if returnDepth:  
 return True  
 elif choice == 2:  
 assembly = Simulator.writeAssembly()  
 returnDepth = self.assemblyMenu(assembly, assembler)  
 if returnDepth:  
 return True  
 elif choice == 3:  
 self.removeAssembler(assembler)  
 return False  
 elif choice == 4:  
 return False  
 elif choice == 5:  
 return True  
 else:  
 UserInterface.output("/!\ UNKNOWN MENU ERROR")  
  
 def assemblerSelect(self):  
 while True:  
 if len(self.\_assemblers) == 0:  
 return  
 menuOptions = self.assemblerNames + ("Return to menu",)  
 choice = UserInterface.menu(menuOptions)  
 if choice == len(menuOptions):  
 return  
 elif choice > len(menuOptions) or choice < 1:  
 UserInterface.output("/!\ UNKNOWN MENU ERROR")  
 else:  
 returnDepth = self.assemblerMenu(choice)  
 if returnDepth:  
 return  
  
 def mainMenu(self):  
 UserInterface.output("===== Computer System Simulator =====")  
 while True:  
 menuOptions = ["Step",  
 "Run steps",  
 "Components",  
 "Assembler",  
 "Console",  
 "End"]  
 if len(self.\_components) == 0:  
 menuOptions.remove("Components")  
 if len(self.\_assemblers) == 0:  
 menuOptions.remove("Assembler")  
 choice = menuOptions[UserInterface.menu(menuOptions) - 1]  
 if choice == "Step":  
 self.step()  
 elif choice == "Run steps":  
 self.runSteps()  
 elif choice == "Components":  
 self.componentsSelect()  
 elif choice == "Assembler":  
 self.assemblerSelect()  
 elif choice == "Console":  
 UserInterface.console(simulator = self)  
 elif choice == "End":  
 return  
 else:  
 UserInterface.output("/!\ UNKNOWN MENU ERROR")

### main.py

from simulator import Simulator  
from processor import Processor  
from memory import ReadOnlyMemory as ROM, RandomAccessMemory as RAM  
from additional\_hardware import PowerSupply, Clock, QuadNANDGate as NAND, Button, Resistor  
from assembler import Assembler  
from instruction\_set import InstructionSet  
from instruction\_set\_65C02.instructions import instructions  
  
instructionSet = InstructionSet(instructions)  
  
powerSupply = PowerSupply()  
  
processor = Processor(  
 instructionSet,  
 connections = (  
 (  
 powerSupply, (  
 ("VDD", "Power"),  
 ("VSS", "Ground")  
 )  
 ),  
 )  
)  
  
rom = ROM(  
 connections = (  
 (  
 powerSupply, (  
 ("VCC", "Power"),  
 ("GND", "Ground"),  
 ("WEB", "Power"),  
 ("OEB", "Ground")  
 )  
 ),  
 (  
 processor, (  
 ("I/O0", "D0"),  
 ("I/O1", "D1"),  
 ("I/O2", "D2"),  
 ("I/O3", "D3"),  
 ("I/O4", "D4"),  
 ("I/O5", "D5"),  
 ("I/O6", "D6"),  
 ("I/O7", "D7"),  
 ("A0", "A0"),  
 ("A1", "A1"),  
 ("A2", "A2"),  
 ("A3", "A3"),  
 ("A4", "A4"),  
 ("A5", "A5"),  
 ("A6", "A6"),  
 ("A7", "A7"),  
 ("A8", "A8"),  
 ("A9", "A9"),  
 ("A10", "A10"),  
 ("A11", "A11"),  
 ("A12", "A12"),  
 ("A13", "A13"),  
 ("A14", "A14")  
 )  
 )  
 )  
)  
  
ram = RAM(  
 connections = (  
 (  
 powerSupply, (  
 ("Vcc", "Power"),  
 ("Vss", "Ground")  
 )  
 ),  
 (  
 processor, (  
 ("I/O0", "D0"),  
 ("I/O1", "D1"),  
 ("I/O2", "D2"),  
 ("I/O3", "D3"),  
 ("I/O4", "D4"),  
 ("I/O5", "D5"),  
 ("I/O6", "D6"),  
 ("I/O7", "D7"),  
 ("A0", "A0"),  
 ("A1", "A1"),  
 ("A2", "A2"),  
 ("A3", "A3"),  
 ("A4", "A4"),  
 ("A5", "A5"),  
 ("A6", "A6"),  
 ("A7", "A7"),  
 ("A8", "A8"),  
 ("A9", "A9"),  
 ("A10", "A10"),  
 ("A11", "A11"),  
 ("A12", "A12"),  
 ("A13", "A13"),  
 ("A14", "A14"),  
 ("WEB", "RWB"),  
 ("OEB", "A14")  
 )  
 )  
 )  
)  
  
clock = Clock(  
 connections = (  
 (  
 powerSupply, (  
 ("VCC", "Power"),  
 ("GND", "Ground")  
 )  
 ),  
 (processor, (("Output", "PHI2"),))  
 )  
)  
  
nand = NAND(  
 connections = (  
 (  
 powerSupply, (  
 ("VCC", "Power"),  
 ("GND", "Ground"),  
 ("A1", "Power"),  
 ("B1", "Power"),  
 ("B3", "Power"),  
 ("B3", "Power")  
 )  
 ),  
 (  
 processor, (  
 ("A4", "A15"),  
 ("B4", "A15")  
 )  
 ),  
 (rom, (("Y4", "CEB"),)),  
 (ram, (("Y2", "CSB"),)),  
 (clock, (("A2", "Output"),))  
 )  
)  
nand.connectPin("Y4", nand, "B2")  
  
reset = Button(  
 connections = (  
 (powerSupply, ((3, "Ground"),)),  
 (processor, ((1, "RESB"),))  
 )  
)  
  
r1 = Resistor(  
 connections = (  
 (powerSupply, ((1, "Power"),)),  
 (reset, ((2, 1),))  
 )  
)  
  
r2 = Resistor(  
 connections = (  
 (powerSupply, ((1, "Power"),)),  
 (processor, ((2, "RDY"),))  
 )  
)  
  
def step(components): *# TODO* components["System clock"].step()  
 pass  
  
presetSimulator = Simulator(  
 components = {  
 "65C02 microprocessor": processor,  
 "AT28C256 ROM": rom,  
 "HM62256B RAM": ram,  
 "NAND gates": nand,  
 "System clock": clock,  
 "Power supply": powerSupply,  
 "RESET button": reset,  
 "Resistor R1": r1,  
 "Resistor R2": r2  
 },  
 step = step,  
 assemblers = {"65C02": Assembler(instructionSet)}  
)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 presetSimulator.mainMenu()

### testing.py

from simulator import Simulator  
from user\_interface import UserInterface  
from instruction\_set\_65C02.instructions import instructions  
from instruction\_set\_65C02.operations import Operations  
from instruction\_set\_65C02.addressing\_modes import AddressingModes  
from processor import Processor  
from memory import Memory, SpecificMemory, RandomAccessMemory, ReadOnlyMemory  
from additional\_hardware import PowerSupply, Clock, QuadNANDGate, Button, Resistor  
from assembler import Assembler  
from instruction\_set import InstructionSet, AddressingMode, Operation  
from component import Component, Node, Connection, Pin, Wire  
from general import int\_to\_bool, bytes\_to\_tuple, slice\_to\_tuple, BinaryElectric  
import unittest  
import random  
  
class TestComponent(Component):  
 def response(self):  
 print(f"TestComponent.response({self})")  
  
class Component\_test(unittest.TestCase):  
 pass  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

1. <https://www.peterhigginson.co.uk/AQA/> [↑](#footnote-ref-2)
2. <http://peterhigginson.co.uk/LMC/> [↑](#footnote-ref-3)
3. <https://www.peterhigginson.co.uk/ARMlite/> [↑](#footnote-ref-4)
4. <http://www.visual6502.org/JSSim/index.html>/ [↑](#footnote-ref-5)
5. <http://emulator101.com/> [↑](#footnote-ref-6)
6. <https://eater.net/6502/> [↑](#footnote-ref-7)
7. <https://en.wikipedia.org/wiki/MOS_Technology_6502>, and <http://archive.6502.org/datasheets/mos_6501-6505_mpu_preliminary_aug_1975.pdf> [↑](#footnote-ref-8)
8. <http://6502.org/tutorials/vflag.html> [↑](#footnote-ref-9)
9. <https://en.wikipedia.org/wiki/WDC_65C02> and <http://archive.6502.org/datasheets/wdc_w65c02s_oct_8_2018.pdf> [↑](#footnote-ref-10)
10. <https://www.jameco.com/Jameco/Products/ProdDS/27861.pdf> [↑](#footnote-ref-11)
11. <https://eater.net/datasheets/74hc00.pdf> [↑](#footnote-ref-12)
12. <https://eater.net/datasheets/28c256.pdf> [↑](#footnote-ref-13)
13. <https://eater.net/datasheets/hm62256b.pdf> [↑](#footnote-ref-14)
14. <https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-wanted>/ and <https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages>/ [↑](#footnote-ref-15)
15. <https://github.com/niklas-heer/speed-comparison>, <http://www.hildstrom.com/projects/langcomp/index.html>/,

    <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/csharp.html>/, and <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/javascript.html>/ [↑](#footnote-ref-16)