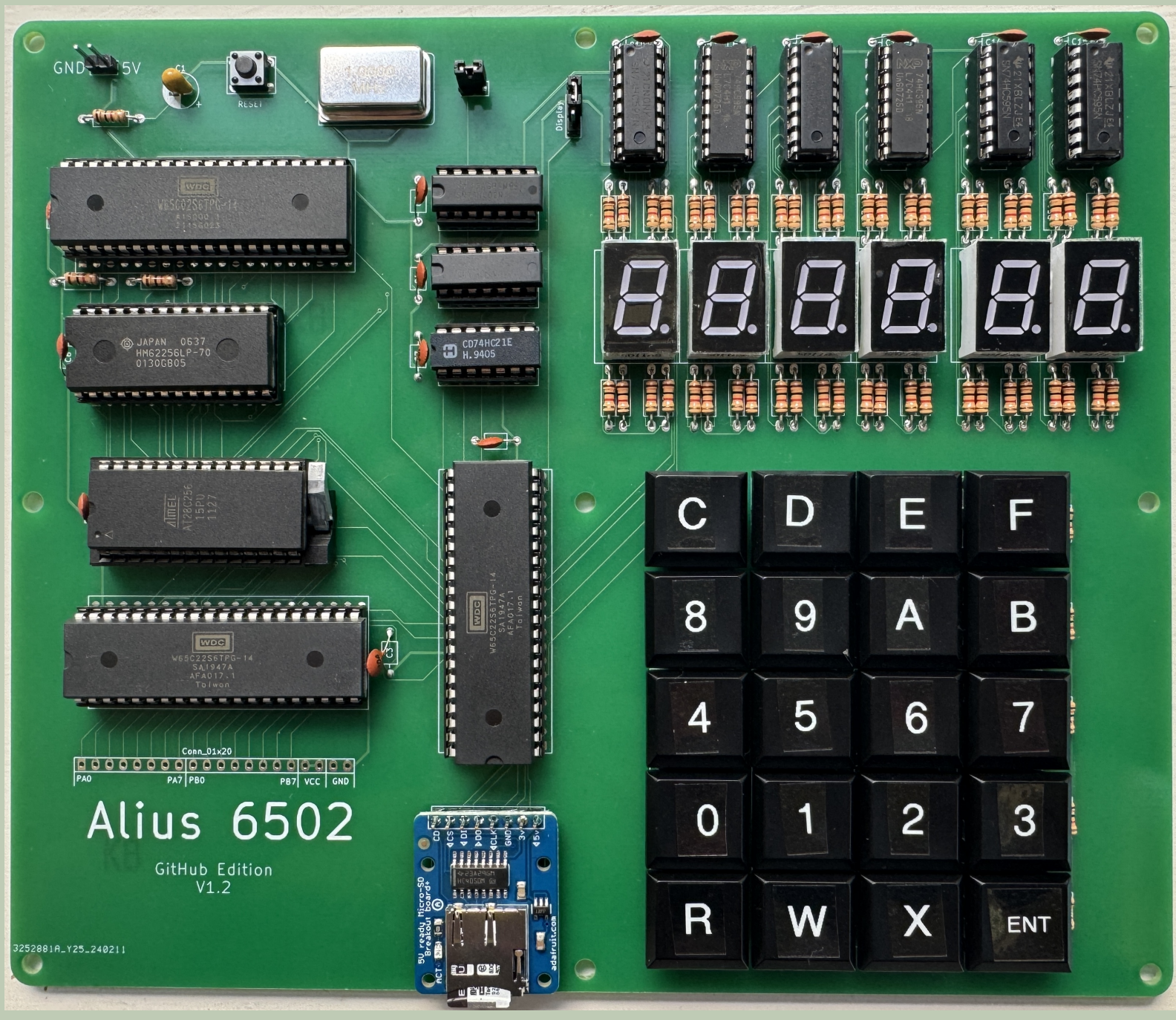
Alius 6502

User Guide

Derek Robson



Contents

1 Introduction

This book is a guide to the design, construction, and use of the Alius 6502 kit-set computer.

This project is aimed at anyone who wants to get a better understanding of basic digital electronics and computers. Throughout this project you will learn what all the chips are used for, how they interconnect and how you can program and expand the Alius 6502.

Many people who design a simple computer start with either a Z80 CPU or a 6502 CPU. Both CPUs are from the mid 1970's and both are very common in early home computers. I selected the 6502 due to my background with the Commodore 64 system in the late 80's.

There are many dozens of 6502 based kit-set computers and this is just another such system. In Latin the word *alius* means another or different, and so the Alius 6502 is another 6502 based system.

The Alius 6502 computer is a simple demonstration computer in the style of a *Homebrew Computer* and can be used by students to learn the basics of how computers work. You should find that only a basic understanding of math and electronics will be required to follow along.

1.1 History

The home computer revolution started in California around 1975 when a number of like-minded electronics enthusiasts started a group called The Homebrew Computer Club.

The Bay Area, which would later become Silicon Valley, was already well known for innovation and home to a growing community of electronics enthusiasts. The formation of the club coincided with the release of the Altair 8800, one of the first commercially successful personal computers, which sparked widespread interest among hobbyists.

One of the club's most famous members was Steve Wozniak, who attended his first meeting in 1975. Inspired by the discussions and the exchange of ideas at the club, Wozniak went on to design and build the Apple I computer.

During the mid 1970s people could build a whole computer from a bunch of chips and write all their own software. With a choice of three or four CPUs many of the designs had a lot in common. At The Homebrew Computer Club ideas and software were shared freely.

The original 6502 CPU was designed by MOS Technologies in 1975 and sold for a fraction of the cost of comparable CPUs. From 1975 through to about 1989 the 6502 was a good choice for home computers and gaming consoles, for example the Apple II and the original Nintendo Entertainment System (NES) used a 6502 and the Commodore 64 used a custom version of the 6502.

In 1981 Western Design Center created a version of the 6502 which has a lower power usage and resolved several bugs from the original design.
The Western Design Center still make and sell the WDC65C02 CPU.

Western Design Center also sell I/O support chips that are compatible with the 6502.

2 System Overview

2.1 Hardware

The Alius 6502 is based around the 65xx family of chips from WDC (Western Design Center).

The WDC65C02 is the CPU for the Alius 6502 computer, and the WDC65C22 is used for input and output.

Alius 6502 Specifications:

- 65C02 CPU running at 1MHz (Design proven at 4MHz)
- 32KB of RAM
- 16KB of ROM
- 2 x 65C22 I/O chips (one for the system and one for experiments)
- SD Card interface

2.2 Software

The Alius 6502 comes with an open source ROM that handles the keypad, seven segment display, and the SD Card functions. Software can be entered directly via the keypad or loaded via the SD Card.

The project GitHub includes a few demo programs. This is an open source project so new software is being added and changed by members of the community.

Open source code for this project can be found online at https://github.com/robsonde/Alius6502_ROM

3 Programming

A large part of this project is learning to program the 6502 CPU. In this section we will cover the basics of how to enter programs on the keypad and how to write your own programs.

Programming sits between and art and a science, programming is the process of creating instructions that a computer can follow to perform specific tasks. The computer will always do what it is told even if that thing is wrong or bad, the errors come from the programmer telling the computer to do the wrong thing or making an assumption about how the computer works.

Most modern programming is done in a high level language like python or PHP, but working in assembly will get you to understand what is really going on in the computer.

Before we can look at programming the computer we will need to understand a bit of math and some of how the CPU works.

3.1 Number Bases

A number base is the number of unique symbols used in the counting system.
Most of us are familiar with base 10 in which we have ten symbols (0-9)
In computers we often use base 2 (binary) or base 16 (hexadecimal).

3.1.1 Base 10

Lets start with a base 10 number.
Each digit is 10 times bigger than the digit to the right of it.

10^3	10^2	10^1	10^0
1000	100	10	1

The number 4632 is four thousand six hundred and thirty two.
4632 (base 10) = (4*1000)+(6*100)+(3*10)+(2*1)

3.1.2 Base 2

2^3	2^2	2^1	2^0
8	4	2	1

1011 (base 2) = (1*8)+(0*4)+(1*2)+(1*1) = 11 (base 10)

3.1.3 Base 16

16^3	16^2	16^1	16^0
4096	256	16	1

$2831 (base 16) = (2*4096)+(8*256)+(3*16)+(1*1) = 10289 (base 10)
$FA10 (base 16) = (15*4096)+(10*256)+(1*16)+(0*1) = 64016 (base 10)

3.2 Hexadecimal Notation

Internally the CPU uses binary on and off, or 1 and 0, but for programming it is more common to use hexadecimal notation.

The decimal system uses 10 symbols of 0 -> 9.
The binary system uses 2 symbols of 0 and 1.
The hexadecimal system uses 16 symbols which are 0 -> 9 and A -> F.

A number in hexadecimal notation is often prefixed with "0x" or "$". For example, "0x6E" or "$6E".

In hexadecimal notation, each digit represents a power of 16. For example, the right most digit in a hexadecimal number represents 16^0 (which is equal to 1), the next digit to the left represents 16^1, the next digit represents 16^2, and so on.

Decimal	Hexadecimal	Binary
0	$0	0000 0000
1	$1	0000 0001
2	$2	0000 0010
3	$3	0000 0011
4	$4	0000 0100
5	$5	0000 0101
6	$6	0000 0110
7	$7	0000 0111
8	$8	0000 1000
9	$9	0000 1001
10	$A	0000 1010
11	$B	0000 1011
12	$C	0000 1100
13	$D	0000 1101
14	$E	0000 1110
15	$F	0000 1111
16	$10	0001 0000

3.3 Use of the Monitor ROM

When the computer starts up it will try and run code from the SD Card, if there is nothing to run then it will default to running the Monitor ROM which allows you to edit memory directly from the keypad.

The seven segment display is split into the left four digits showing a memory address, the right two digits showing the contents of that memory address.

The main keypad is hexadecimal and the bottom row of four keys are control keys.

R - This key puts the system into memory read mode.
Any memory address entered on the keypad will display memory at that location. The ENT key will move the address up by one and allow you to view the next memory location.

W - This key puts the system into memory write mode.
Once an address is entered the memory is displayed, the ENT key allows you to change the contents of that memory location. Repeated use of the ENT key allows you to enter data at the next memory location. This can be used to type in small programs.

X - This key puts the system into execute mode.
Any memory address can be entered and the ENT key will trigger execution to start from that address.

ENT - This is the enter key for the system.

Example Read:
To view memory at $0200 you would type R, 0, 2, 0, 0.
Then if you push ENT it will show memory at $0201.

Example Write:
To write $FA to memory at $1000 you would type W, 1, 0, 0, 0, ENT, F, A, ENT.
Then it will move the address up to $1001 and you can enter more data by just typing the data and ENT.

3.4 CPU Registers

Within any CPU there are a number of registers. A register can be thought of as RAM inside the CPU that can be accessed very quickly.

Within the 6502 CPU there are three general purpose registers and three special registers.

The Accumulator - 8 bits
The Accumulator is often just called the A register. The Accumulator is an 8 bit register and this is why the 6502 is referred to as an 8 bit CPU. It is the only register that can be used for math and logic operations.

The X register - 8 bits
The X register is often used as part of memory addressing modes in which the value of X is added to the base address of a load or store instruction. The X register is also often used as a counter for a loop or other temporary storage.

The Y register - 8 bits
The Y register is often used as part of memory addressing modes in which the value of Y is added to the base address of a load or store instruction. The Y register is also often used as a counter for a loop or other temporary storage.

The Stack Pointer (SP) - 8 bits
The Stack Pointer is a special register that you can't interact with directly. It is used to point to the next memory address on the stack. The stack pointer can be moved to or from the X register which can allow for advanced memory access.

The Status register - 8 bits
Often called Flags, the Status register is a special register that has 8 bits to show the result of the last instruction.

The Program Counter (PC) - 16 bits
This special register holds the memory address of the current instruction being executed. The Program Counter can be changed by a branch or jump instruction.

3.5 Your First Program

Programming of the Alius 6502 is done via either assembly language or machine code.

Example of assembly language:

```
.org $1000
  LDA #$2E
  ADC #$AB
  STA $10FF
  BRK
```

Example of machine code:

```
1000: A9 2E 69 AB 8D FF 10 00
```

The two examples above are the exact same program but displayed in different ways, the assembly language version is what you would write, and the machine code is what the compiler will create for loading into the memory of the computer.

Looking at the assembly language line by line:

```
.org $1000
```

We set the start location to $1000 using the statement .org $1000

```
  LDA #$2E
```

Next we Load the Accumulator with $2E

```
  ADC #$AB
```

Then we have ADC $AB which will add $AB to the Accumulator

```
  STA $10FF
```

We then store the Accumulator to memory address $10FF

```
  BRK
```

Last we run the Brake instruction, which will return control to the monitor ROM.

3.6 Memory Map

The memory of the 6502 system is split up into areas for different uses.

The upper byte of a memory address is often called the page number. Memory is split up into 256 pages of 256 bytes per page.

Hex Address	Page	Description
$0000 - $00FF	Page 0	Zero Page Addressing
$0100 - $01FF	Page 1	Stack
$0200 - $02FF	Page 2	System variables and pointers
$0300 - $04FF	Page 3-4	File System buffer
$0500 - $7FFF	Page 5-127	User Space RAM
$8000 - $80FF	Page 128	I/O (65C22)
$8100 - $9FFF	Page 129-159	DO NOT USE
$A000 - $FEFF	Page 160-254	ROM Code
$FF00 - $FFFF	Page 255	ROM Jump tables

The space from $8100 - $9FFF is not mapped to real RAM and so use of that space will incorrectly trigger I/O functions in unpredictable ways and so should not be used.

Zero Page (ZP) is special as it can be accessed faster than other memory in the system. Some of Zero Page is already used by the system and so you should avoid overwriting these memory locations.

Zero Page (ZP) Memory

Hex Address	Description
$0000-$00CF	Free for use.
$00D0-$00D2	Digits to display on seven segment display.
$00D3-$00D8	Bit-mask for the display.
$00F6-$00F7	A 16 bit variable, used by FAT32 code.
$00F8-$00F9	A 16 bit pointer, use by FAT32 code.
$00FA-$00FB	A 16 bit pointer, points to FAT32 buffer.
$00FC-$00FD	A 16 bit counter byte by assorted ROM code.
$00FE	A counter byte by assorted ROM code.
$00FF	A temporary byte used by assorted ROM code.

Memory at Page 2 is mostly used for the system to hold data related to reading files from the SD Card and FAT32 support.

Memory Page 2 - System variables and pointers

Hex Address	Description
$0200	Random number generator byte.
$0201 - $0202	Random number generator seed.
$0203 - $0205	Temporary bytes used during reading keypad.
$0206	Error return code from ROM functions.
$0207 - $0208	Monitor ROM state.
$0209 - $0224	Assorted storage for FAT32 support.
$0250	File number for loading a file by number.
$0251 - $025C	Filename padded to 11 bytes.
$025D - $025E	16 bit variable, file size.
$025F	Number of sectors to load with FAT32.
$0260 - $0261	16 bit pointer to load address, $0260(LSB),$0261(MSB).
$0270	Debug Accumulator.
$0271	Debug X Register.
$0272	Debug Y Register.
$0273	Debug CPU Status Register.
$0274	Debug Program Counter Low byte.
$0275	Debug Program Counter High byte.
$0276 - $02FD	Future expansion use.
$02FE - $02FF	16 bit pointer to IRQ handler, $02FE(LSB),$02FF(MSB).

The space from $0276 - $02FD is currently not used by the system but future versions of the ROM may use this space and so user programs should avoid using this space.

Memory Page 128 - I/O (65C22)

Hex Address	Name	Description
$8000 - $800F	–	DO NOT USE
$8010	ORB/IRB	Input / Output B
$8011	ORA/IRA	Input / Output A
$8012	DDRB	Data Direction Register B
$8013	DDRA	Data Direction Register A
$8014	T1C-L	Timer T1 LSB counter
$8015	T1C-H	Timer T1 MSB counter
$8016	T1L-L	Timer T1 LSB latches
$8017	T1L-H	Timer T1 MSB latches
$8018	T2C-L	Timer T2 LSB counter
$8019	T2C-H	Timer T2 MSB counter
$801A	SR	Shift Register
$801B	ACR	Auxiliary Control Register
$801C	PCR	Peripheral Control Register
$801D	IFR	Interrupt Flag Register
$801E	IER	Interrupt Enable Register
$801F	ORA/IRA	Same as $8011 except no "Handshake"

The I/O from $8010 - $801F is for system use. Do not use for experiments as any mistakes could damage hardware.

Memory Page 128 - I/O (65C22)

Hex Address	Name	Description
$8020	ORB/IRB	Input / Output B
$8021	ORA/IRA	Input / Output A
$8022	DDRB	Data Direction Register B
$8023	DDRA	Data Direction Register A
$8024	T1C-L	Timer T1 LSB counter
$8025	T1C-H	Timer T1 MSB counter
$8026	T1L-L	Timer T1 LSB latches
$8027	T1L-H	Timer T1 MSB latches
$8028	T2C-L	Timer T2 LSB counter
$8029	T2C-H	Timer T2 MSB counter
$802A	SR	Shift Register
$802B	ACR	Auxiliary Control Register
$802C	PCR	Peripheral Control Register
$802D	IFR	Interrupt Flag Register
$802E	IER	Interrupt Enable Register
$802F	ORA/IRA	Same as $8021 except no "Handshake"
$8030 - $80FF	–	DO NOT USE

3.7 Memory Addressing

The 6502 has several ways to access memory. The three most basic addressing modes are Immediate, Absolute and Zero Page.

Immediate: In this mode the data is directly after the instruction. This is indicated by a hash symbol # before the data.
Example: LDA #$FE
The Accumulator will be set to $FE

Absolute: In this mode the address of the data is directly after the instruction.
Example: LDA $AF3F
The Accumulator will be set to the data stored at memory location $AF3F

Zero Page: In this mode the first byte of the address is assumed to be $00 and so the data can be accessed faster and with less code.
Example: LDA $AF
The Accumulator will be set to the data at address $00AF

More advanced addressing modes allow for access to arrays and tables of data with pointers.

Absolute Indexed: In this mode the X or Y register is added to an absolute address.
Example: LDA $1000,X
If X was set to $0F then the Accumulator will be set to the data at the final address of $100F

Zero-Page Indexed: This is just Absolute Indexed mode as above but using Zero Page as the base.
Example: LDA $A3,X
If X was set to $01 then the Accumulator will be set to the data at the memory location of $00A4

Indexed Indirect: This allows for use of pointers. A pointer is where two bytes of memory hold a final address.
Example: LDA ($E0,X)
In this example we will assume that X register is $07, memory at $00E7 is set to $23 and memory at $00E8 is set to $5E
The CPU will add X to $E0 to get $E7 and the bytes at $E7 and $E8 are used as a pointer to a final address of $5E23

Indirect Indexed: This mode is similar to the above mode but the Y register is added to the pointer at the end.
Example: LDA ($E0),Y
In this example we will assume that Y register is $07, memory at $00E0 is set to $23 and memory at $00E1 is set to $5E
The CPU will load the address of $5E23 from $00E0 and $00E1 and then add $07 to get a final address of $5E2A

3.8 The Stack

The Stack is a special section of memory found from $0100 to $01FF.

If you think of the stack like a stack of plates, you can add one to the top of the stack or take one from the top of the stack. Pulling a plate from the middle of the stack will make a mess.

To push a byte onto the stack you can use the PHA instruction and to pop a byte off the stack you can use PLA.

If you need to store other registers to the stack then you can move them to the A register first.

```
PHA ; Push A to stack
TXA ; Transfer X to A
PHA ; Push A to stack.
TYA ; Transfer Y to A
PHA ; Push A to stack.
```

On the 6502 the stack is limited to only 256 bytes, if you push too many things the stack will wrap around and overwrite data on the stack.

3.9 Flow Control

Any non trivial program will have the need for flow control. Flow control is only running a section of code if a condition is met and means that a block of code will only be run sometimes.

In a high level language like Python you will have IF/THEN blocks. In assembly we use a compare and conditional branch.

```
.org $1000
  LDA #$03
  LDX #$00
LOOP:
  ADC #$03
  INX
  CPX #$07
  BNE LOOP
```

In this example it is the last two instructions that we are discussing. CPX is a "Compare X with" instruction and we are comparing X with $07
then we have BNE which is "Branch if Not Equal".

The whole example will run around the loop adding $03 to the Accumulator and then incrementing X until X = $07.

A branch instruction can only branch to an address of +127 or -128 bytes from the current location. This is called relative addressing.

3.9.1 CPU Flags

The CPU status register (often called flags) is really just 7 one bit registers. Many instructions will change one or more of the bits in the status register.

Bit 0 - C - Carry flag - This is the arithmetic carry out of an ADC instruction and the carry into an ADC.
Bit 1 - Z - Zero flag - This flag is set if the last instruction caused any register to contain zero. Also set or unset by doing a compare.
Bit 2 - I - Interrupt flag - This enables or disables interrupts, when this is set then interrupts are disabled.
Bit 3 - D - Decimal flag - Controls if ADC and SBC use normal binary mode or special decimal mode.
Bit 4 - B - Break flag - Only set if the BRK instruction has caused an interrupt.
Bit 5 - - not used.
Bit 6 - V - Overflow flag - Only relevant to signed arithmetic and shows if there is a twos complement overflow.
Bit 7 - N - Negative flag - This means that Bit 7 of the accumulator is set.

3.9.2 More Flow Control

You can branch on several conditions other than just the outcome of a compare:

Instruction	Description
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on result Zero or Equal
BMI	Branch on result Minus
BNE	Branch on result Not Zero or Not Equal
BPL	Branch on result Positive
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set

3.10 I/O (Input / Output)

The Alius 6502 system has two 65C22 Input and Output chips that can be used to connect to devices outside the basic RAM/ROM. One of the two chips is used for interfacing with the Keypad, Display and the SD Card, the other chip is free for student experiments.

The 65C22 has two 8-bit bi-directional I/O ports, and two 16-bit programmable Timers/Counters.
The first data port is called A and is located at $8021, the second port called B is located at $8020.

The order of the two ports is the reverse of what you might expect, with port B the lower address in memory.

Any bit of either port can be used for Input or Output. This is selected via the Data Direction Register (DDR). DDRB is at $8022 and DDRA is at $8023.

Example code to turn on all bits of port B

```
PORTB = $8020
PORTA = $8021
DDRB = $8022
DDRA = $8023

.org $1000
 LDA #$FF
 STA DDRB ; Setting a bit high marks that bit as output.
 STA PORTB ; Set all the bits high / on for port B
 BRK ; Return to monitor ROM
```

Example code to blink a LED on bit 0 of port B

```
PORTB = $8020
PORTA = $8021
DDRB = $8022
DDRA = $8023

.org $1000
 LDA #$FF
 STA DDRB ; Setting a bit high marks that bit as output.
LOOP:
 LDA #$01 ; This is only bit 0 high / on
 STA PORTB ; Sets port B as per Accumulator
 JSR $FF30 ; Calls a ROM function to sleep for 0.5 seconds
 LDA #$00 ; Whole byte is low / off
 STA PORTB ; Sets port B as per Accumulator
 JSR $FF30 ; Calls a ROM function to sleep for 0.5 seconds
 JMP LOOP ; Jumps back to LOOP
```

Example code to wait for an input on port A

```
PORTB = $8020
PORTA = $8021
DDRB = $8022
DDRA = $8023

.org $1000
 LDA #$00
 STA DDRA ; Setting a bit low marks that bit as input.
LOOP:
 LDA PORTA ; Read in all the bits for port A
 CMP #$00
 BEQ LOOP
```

There is another 65C22 chip at $8010 -> $801F and this runs the keypad, screen and SD Card. Do NOT use this unless you are working to improve the ROM as any mistakes could damage hardware.

3.11 IRQ (Interrupt Request)

An IRQ is a way that devices can request attention from the operating system. When a device wants to send data to the CPU or perform some other action, it sends an IRQ to the CPU. The CPU stops what it is currently doing and services the request from the device. Once the request has been serviced, the CPU returns to its previous task. IRQs are used to ensure that the CPU can efficiently manage the various devices that are connected to the system.

When an IRQ is triggered the CPU will push the current Program Counter and the CPU status flags to the stack and then jump to a special address to run code called an interrupt handler. The last instruction in an interrupt handler is RTI and that will cause the CPU to pop the CPU status flags and the Program Counter from the stack and resume from exactly where it left off. It should be noted that no other CPU registers are saved by default, so many interrupt handlers will push the A,X,Y registers at the start and pop them at the end.

You can use the 65C22 chip to create interrupts based on a timer.

In the Alius 6502 design there is a pointer at $02FE/$02FF which points to the interrupt handler code. On power up this pointer is set to a "do nothing" function in the ROM.

Within IRQ code be sure to save and restore any registers you use so as not to interfere with code running outside the IRQ.

3.11.1 IRQ - Timers

One way we can cause an interrupt is to use the timer from the 65C22.

With a CPU clock of 1MHz it seems logical to setup a counter at 1 million and get an IRQ once a second. The timers on the 65C22 only go as high as 65,535 and so we can't setup to get an interrupt once a second. With a clock running at 1MHz we can setup the counter for 50,000 ticks of the timer and trigger an IRQ 20 times a second.

In this example we will setup a timer to trigger an IRQ 20 times a second, and then by use of incrementing counters we can create a basic digital clock.

We start by setting up the IRQ vector to point to our interrupt handler code.

```
IRQ_Vec = $02FE ; IRQ vector in ram.
  LDA #<IRQ ; Get the low byte of address of IRQ code
  STA IRQ_Vec ; Store to IRQ Vector
  LDA #>IRQ ; Get the high byte of address of IRQ code
  STA IRQ_Vec+1 ; Store to IRQ Vector
```

We then setup the T1 counter to 50,000 or $C350. the counter will count down to zero and then trigger an interrupt.

```
  LDA #$50 ; Put $C350 (50,000) in the timer 1 counter.
  STA T1CL ; Write counter low byte.
  LDA #$C3 ; $C3 is the high byte of $C350
  STA T1CH ; Write high byte.
```

Last step is to setup the 65C22 to trigger an interrupt.

```
  LDA #01000000B ; Set the bit that tells T1 to
     automatically
  STA ACR ; Produce an interrupt at every time-out

  LDA #11000000B
  STA IER ; Enable the T1 interrupt in the VIA.
  CLI ; Enable interrupts
```

This next section is the interrupt handler and is called when an IRQ happens. The first thing we do is push the Accumulator to the stack so that we don't impact that register being used in the main code.

Next we increment "Tick" which is our counter of 20ths of a second. If that is not 20 ($14) then we are done, else we increment the Seconds counter and reset the Tick counter.

```
IRQ:
  PHA ; Push A to stack
  BIT T1CL ; Clear the IRQ state by reading from T1
     counter low byte
  INC Tick ; Increment the sub second counter in Zero page
  LDA #$14 ; Check if we have counted 20 sub seconds
  CMP Tick ; Check sub seconds counter in Zero page
  BNE ExitIRQ ; If we have not made 20 sub seconds then
     exit IRQ code
  INC Seconds
  LDA #$00
  STA Tick
```

The last section is to pop the Accumulator and the RTI instruction returns from the IRQ.

```
ExitIRQ:
  PLA ; Pop A from stack before exiting from IRQ
  RTI ; Exit from IRQ
```

3.12 Math

3.12.1 8 bit add

As the 6502 is an 8-bit CPU, so an 8-bit add is simple.

```
Add:
  CLC ; We clear the Carry flag before the add.
  LDA #$47 ; Load Accumulator with $47 (71 decimal)
  ADC #$1A ; Add with Carry - add $1A (26 decimal)
  STA $01 ; Store result in Zero page at $0001
```

Should the add go over $ff (255 decimal) then the Carry flag will be set.

3.12.2 8 bit subtract

```
Sub:
  SEC ; We set the carry flag before a subtract.
  LDA #$53 ; Load A with $53 (83 decimal)
  SBC #$10 ; Subtract with carry $10 (10 decimal)
  STA $01 ; Store result in Zero page at $0001
```

3.12.3 16 bit add

```
Add16:
  CLC
  LDA num1_low
  ADC num2_low
  STA result_low
  LDA num1_high
  ADC num2_high
  STA result_high
```

3.12.4 16 bit subtract

```
Sub16:
  SEC
  LDA num1_low
  SBC num2_low
  STA result_low
  LDA num1_high
  SBC num2_high
  STA result_high
```

3.12.5 Multiply

Lets look at how you would multiply 423 and 25 in decimal.
As discussed earlier in number bases we can break down each number.
So we can add (3 * 25) + (2 * 250) + (4 * 2500).
The lowest digit in 423 is 3 and we multiply it by 25, then we move to the next digit in 423 and multiply 25 by 10 (the base)

If we do this in binary the multiply by the base becomes a shift and then we either add or not, as each binary digit is 1 or 0.

```
; The starting factors in FactorA and FactorB
  LDA #0 ; set A to Zero.
  LDX #$8 ; We need to go around 8 times
  LSR FactorA ; The shift will leave one bit in the carry
     flag.
loop:
  BCC no_add ; If no carry flag then no add.
  CLC ; clear carry before add.
  ADC FactorB ; add factorB if carry was set from factorA.
no_add:
  ROR ; multply result by 2
  ROR FactorA ; rotate and get another bit into carry
  DEX ; decrement our counter
  BNE loop ; if we have not done 8 bits then go around
     again.
  STA FactorB ; Store the answer
; The high byte of the result is in FactorB, low byte is
   in FactorA
```

3.12.6 Negative numbers

Memory just holds bits, if they are positive or negative is up to how you want to interpret the bits.

Basic math logic says that -1 + 1 = 0.
11111111 can be -1 or 255.

```
00000000 = 0
00000001 = +1
...
01111101 = +125
01111110 = +126
01111111 = +127
...
10000000 = -128
10000001 = -127
10000010 = -126
...
11111111 = -1
00000000 = 0
```

3.13 Debugging Code

Every programmer makes mistakes, this creates bugs in the code.

Finding and fixing such mistakes is called debugging and the Alius 6502 has a trick or two to help you find and fix the bugs in your code.

The BRK instruction will cause the system to stop running your code and return to the Monitor ROM.

Once back in the Monitor ROM you can use a few memory locations to examine the CPU registers. You can also restart the program from just after the BRK.

Debug memory locations:
$0270 - Accumulator
$0271 - X Register
$0272 - Y Register
$0273 - CPU Status Register
$0274 - Program Counter Low byte
$0275 - Program Counter High byte.

To return to execution you can just execute $FFAA by typing "X, F, F, A, A, ENT"

The CPU will ignore the byte directly after the BRK instruction, so it is required to have a padding byte after the BRK. Typically this is done with a NOP instruction.

3.14 Compiling Code

There are many 6502 compilers and if you have a small program you can even compile it by hand to create the machine code.

3.14.1 Compiling Code by hand

Lets look at a single line of code:

```
LDA $05FF
$
```

This is a Load Accumulator from an absolute address, so the opcode is "$AD", this is followed by the address low byte of "$FF" and then the address high byte of "$05"

The assembled code is:

```
$AD $FF $05
$
```

The 6502 CPU is a *Little-endian* design meaning that for a 16 bit address it is written with the "little end" or low byte first.

A larger example:

```
LDA #$42 ; Load A with $42
CLC ; Clear carry before add.
ADC $0900 ; Add value at $0900
STA $09FF ; Store result at $09FF
$
```

Load Accumulator from Immediate which is "$A9" and the data to load is "$42"
Clear Carry flag is "$18"
Add with Carry from absolute address is "$6D" then the address in Little-endian byte order.
Store Accumulator at absolute address is "$8D" and then the address in Little-endian byte order.

```
$A9 $42
$18
$6D $00 $09
$8D $FF $09
$
```

3.14.2 Compiling Code with a compiler

For any program larger than a few instructions it is easier to use a compiler.

It is typical to make your source file have a ".s" extension to show that it is a source file.

```
vasm6502_oldstyle -Fbin -dotdir example.s
```

This will create a file called a.out which is the final machine code file that you can write to an SD Card or flash to the ROM chip.

The VASM compiler used in this book can be found at http://sun.hasenbraten.de/vasm/ and has versions for both Windows and Linux.

3.15 Running Code from SD Card

The Alius 6502 can load files from an SD Card. This can make it easier to write larger programs and not have to type them into the keypad.

By default on boot up the Alius 6502 will check for an SD Card and then try to load a file called "00.BIN". If the 00.BIN is not found then the system will jump to the Monitor ROM and allow keypad entry.

When in the Monitor ROM you can load a different file by setting a file number in to memory location $0250 and then executing code at $FF90.

For example to load and run a file called F7.BIN from the monitor ROM: W, 0, 2, 5, 0, ENT, F ,7 ,ENT, X, F, F, 9, 0, ENT

The format of the SD Card is important. Files must be in the root directory, and the card must be formatted as a FAT32 file system.

3.16 Experiments

Here you will find a few simple programs that you can enter directly on the keypad.

3.16.1 Adding two numbers

In this simple example we will add two numbers together.

```
.org $1000 ; Set program start to memory location $1000
   LDA #$2F ; Load the Accumulator with $2F
   CLC ; Clear the Carry Flag to get to known state
   ADC #$42 ; Add $42 to the Accumulator
   STA $2000 ; Store the Accumulator at $2000
   BRK ; Return control to the Monitor ROM
```

Once compiled this becomes:

```
1000: A9 2F 18 69 42 8D 00 20
1008: 00
```

This can be typed into the keypad with the following key strokes.

W, 1, 0, 0, 0, ENT
A, 9, ENT
2, F, ENT
1, 8, ENT
6, 9, ENT
4, 2, ENT
8, D, ENT
0, 0, ENT
2, 0, ENT
0, 0, ENT

Then it can be run via X, 1, 0, 0, 0, ENT.

If all has gone well you can examine memory at $2000 and see if it holds $71.

3.16.2 Counting

```
  .org $1000

start:
  LDA #$00 ; Load A with zero.
  STA $D0 ; Store A into first Display buffer byte.
  STA $D1 ; Store A into next Display buffer byte.
  STA $D2 ; Store A into last Display buffer byte.

Count:
  JSR $ff10 ; Jump to ROM to update the 7 segment display.
  INC $D0 ; Increment the Display buffer.
  BNE Count ; If not 00 then jump back to top of loop.
  INC $D1 ; Increment the Display buffer.
  BNE Count ; If not 00 then jump back to top of loop.
  INC $D2 ; Increment the Display buffer.
  JMP Count ; jump back to top of loop.
```

Once complied this becomes:

```
1000: A9 00 85 D0 85 D1 85 D2
1008: 20 10 FF E6 D0 D0 F9 E6
1010: D1 D0 F5 E6 D2 4C 08 10
\
```

3.17 Coding Challenges

Here are a few small programming challenges to test your skills.

3.17.1 Store the number 100 in memory location $2000

Hint: It's just a load and a store, but check addressing mode.

3.17.2 Move data from memory location $2005 to location $2006

Hint: It's just a load and a store, but a different addressing mode.

3.17.3 Swap data from memory location $2008 to location $2009

Hint: Where do you stack it while you move it around?

3.17.4 Fill the memory from $2010 to location $2020 with $00

Hint: A loop and another addressing mode.

3.18 Advanced Programming Tricks

Here are a few ideas for how to make smaller or faster code by small tricks.

3.18.1 Loops

This example will loop around for $20 times.

```
   LDX #$00 ; set counter to zero
LOOP:
   NOP ; do nothing for example
   INX ; increment counter
   CPX #$20 ; compare X with $20
   BNE LOOP ; branch if not equal
```

However if you set the loop size at the top and count down then you get the compare for free as the decrement will leave a Status of Zero.

```
   LDX #$20 ; set counter to zero
LOOP:
   NOP ; do nothing for example
   DEX ; Decrement counter
   BNE LOOP ; branch if not equal
```

4 Programming References

4.1 Assembler Op Codes

ADC

Add to Accumulator with Carry

Mode	Syntax	HEX	Length	Time
Immediate	ADC #$44	$69	2	2
Zero Page	ADC $44	$65	2	3
Zero Page, X	ADC $44,X	$75	2	4
Absolute	ADC $4400	$6d	3	4
Absolute, X	ADC $4400,X	$7d	3	4 (+1 if page crossed)
Absolute, Y	ADC $4400,Y	$79	3	4 (+1 if page crossed)
(Indirect)	ADC ($4400)	$72	2	5
(Indirect, X)	ADC ($4400,X)	$61	2	6
(Indirect), Y	ADC ($4400),Y	$71	2	5 (+1 if page crossed)

FLAGS - N V - B D I Z C

AND

AND with Accumulator

Mode	Syntax	HEX	Length	Time
Immediate	AND #$44	$29	2	2
Zero Page	AND $44	$25	2	3
Zero Page, X	AND $44,X	$35	2	4
Absolute	AND $4400	$2d	3	4
Absolute, X	AND $4400,X	$3d	3	4 (+1 if page crossed)
Absolute, Y	AND $4400,Y	$39	3	4 (+1 if page crossed)
(Indirect)	AND ($4400)	$32	2	5
(Indirect, X)	AND ($4400,X)	$21	2	6
(Indirect), Y	AND ($4400),Y	$31	2	5 (+1 if page crossed)

FLAGS - N V - B D I Z C

ASL

Arithmetic Shift Left

Mode	Syntax	HEX	Length	Time
Implied	ASL	$0a	1	2
Zero Page	ASL $44	$06	2	5
Zero Page, X	ASL $44,X	$16	2	6
Absolute	ASL $4400	$0e	3	6
Absolute, X	ASL $4400,X	$1e	3	6 (+1 if page crossed)

FLAGS -

N	V	-	B	D	I	Z	C

BBR

Branch On Bit Reset

Mode	Syntax	HEX	Length	Time
Bit 0 - Zero Page	BBR0, $44	$0f	3	4
Bit 1 - Zero Page	BBR1, $44	$1f	3	4
Bit 2 - Zero Page	BBR2, $44	$2f	3	4
Bit 3 - Zero Page	BBR3, $44	$3f	3	4
Bit 4 - Zero Page	BBR4, $44	$4f	3	4
Bit 5 - Zero Page	BBR5, $44	$5f	3	4
Bit 6 - Zero Page	BBR6, $44	$6f	3	4
Bit 7 - Zero Page	BBR7, $44	$7f	3	4

FLAGS -

N	V	-	B	D	I	Z	C

BBS

Branch On Bit Set

Mode	Syntax	HEX	Length	Time
Bit 0 - Zero Page	BBS0, $44	$8f	3	4
Bit 1 - Zero Page	BBS1, $44	$9f	3	4
Bit 2 - Zero Page	BBS2, $44	$af	3	4
Bit 3 - Zero Page	BBS3, $44	$bf	3	4
Bit 4 - Zero Page	BBS4, $44	$cf	3	4
Bit 5 - Zero Page	BBS5, $44	$df	3	4
Bit 6 - Zero Page	BBS6, $44	$ef	3	4
Bit 7 - Zero Page	BBS7, $44	$ff	3	4

FLAGS -

N	V	-	B	D	I	Z	C

BCC

Branch if Carry Clear

Mode	Syntax	HEX	Length	Time
Immediate	BCC #$44	$90	2	2 (+1 if branch taken, +2 if to a new page)

FLAGS - N | V | - | B | D | I | Z | C

BCS

Branch if Carry Set

Mode	Syntax	HEX	Length	Time
Immediate	BCS #$44	$b0	2	2 (+1 if branch taken, +2 if to a new page)

FLAGS - N | V | - | B | D | I | Z | C

BEQ

Branch if Result Zero

Mode	Syntax	HEX	Length	Time
Immediate	BEQ #$44	$f0	2	2 (+1 if branch taken, +2 if to a new page)

FLAGS - N | V | - | B | D | I | Z | C

BIT

Bit Test with Accumulator

Mode	Syntax	HEX	Length	Time
Immediate	BIT #$44	$89	2	2
Zero Page	BIT $44	$24	2	3
Zero Page, X	BIT $44,X	$34	2	4
Absolute	BIT $4400	$2c	3	4
Absolute, X	BIT $4400,X	$3c	3	4 (+1 if page crossed)

FLAGS - N | V | - | B | D | I | Z | C

BMI

Branch if Result Negative

Mode	Syntax	HEX	Length	Time
Immediate	BMI #$44	$30	2	2 (+1 if branch taken, +2 if to a new page)

FLAGS - N V - B D I Z C

BNE

Branch if Result Not Zero

Mode	Syntax	HEX	Length	Time
Immediate	BNE #$44	$d0	2	2 (+1 if branch taken, +2 if to a new page)

FLAGS - N V - B D I Z C

BPL

Branch if Result Positive

Mode	Syntax	HEX	Length	Time
Immediate	BPL #$44	$10	2	2 (+1 if branch taken, +2 if to a new page)

FLAGS - N V - B D I Z C

BRA

Branch Always

Mode	Syntax	HEX	Length	Time
Immediate	BRA #$44	$80	2	3 (+1 if page crossed)

FLAGS - N V - B D I Z C

BRK

Break / Interrupt

Mode	Syntax	HEX	Length	Time
Implied	BRK	$00	1	7

FLAGS - N V - B D I Z C

BVC

Branch if Overflow Clear

Mode	Syntax	HEX	Length	Time
Immediate	BVC #$44	$50	2	2 (+1 if branch taken, +2 if to a new page)

FLAGS - N V - B D I Z C

BVS

Branch if Overflow Set

Mode	Syntax	HEX	Length	Time
Immediate	BVS #$44	$70	2	2 (+1 if branch taken, +2 if to a new page)

FLAGS - N V - B D I Z C

CLC

Clear Carry

Mode	Syntax	HEX	Length	Time
Implied	CLC	$18	1	2

FLAGS - N V - B D I Z C

CLD

Clear Decimal Mode

Mode	Syntax	HEX	Length	Time
Implied	CLD	$d8	1	2

FLAGS - N V - B D I Z C

CLI

Clear Interrupt Disable

Mode	Syntax	HEX	Length	Time
Implied	CLI	$58	1	2

FLAGS - N V - B D I Z C

CLV

Clear Overflow

Mode	Syntax	HEX	Length	Time
Implied	CLV	$b8	1	2

FLAGS - N V - B D I Z C

CMP

Compare with Accumulator

Mode	Syntax	HEX	Length	Time
Immediate	CMP #$44	$c9	2	2
Zero Page	CMP $44	$c5	2	3
Zero Page, X	CMP $44,X	$d5	2	4
Absolute	CMP $4400	$cd	3	4
Absolute, X	CMP $4400,X	$dd	3	4 (+1 if page crossed)
Absolute, Y	CMP $4400,Y	$d9	3	4 (+1 if page crossed)
(Indirect)	CMP ($4400)	$d2	2	5
(Indirect, X)	CMP ($4400,X)	$c1	2	6
(Indirect), Y	CMP ($4400),Y	$d1	2	5 (+1 if page crossed)

FLAGS - N V - B D I Z C

CPX

Compare with X

Mode	Syntax	HEX	Length	Time
Immediate	CPX #$44	$e0	2	2
Zero Page	CPX $44	$e4	2	3
Absolute	CPX $4400	$ec	3	4

FLAGS - N V - B D I Z C

CPY

Compare with Y

Mode	Syntax	HEX	Length	Time
Immediate	CPY #$44	$c0	2	2
Zero Page	CPY $44	$c4	2	3
Absolute	CPY $4400	$cc	3	4

FLAGS - N V - B D I Z C

DEC

Decrement by one

Mode	Syntax	HEX	Length	Time
Implied	DEC	$3a	1	2
Zero Page	DEC $44	$c6	2	5
Zero Page, X	DEC $44,X	$d6	2	6
Absolute	DEC $4400	$ce	3	6
Absolute, X	DEC $4400,X	$de	3	7

FLAGS - N | V | - | B | D | I | Z | C

DEX

Decrement X by one

Mode	Syntax	HEX	Length	Time
Implied	DEX	$ca	1	2

FLAGS - N | V | - | B | D | I | Z | C

DEY

Decrement Y by one

Mode	Syntax	HEX	Length	Time
Implied	DEY	$88	1	2

FLAGS - N | V | - | B | D | I | Z | C

EOR

Exclusive OR with Accumulator

Mode	Syntax	HEX	Length	Time
Immediate	EOR #$44	$49	2	2
Zero Page	EOR $44	$45	2	3
Zero Page, X	EOR $44,X	$55	2	4
Absolute	EOR $4400	$4d	3	4
Absolute, X	EOR $4400,X	$5d	3	4 (+1 if page crossed)
Absolute, Y	EOR $4400,Y	$59	3	4 (+1 if page crossed)
(Indirect)	EOR ($4400)	$52	2	5
(Indirect, X)	EOR ($4400,X)	$41	2	6
(Indirect), Y	EOR ($4400),Y	$51	2	5 (+1 if page crossed)

FLAGS - N | V | - | B | D | I | Z | C

INC

Increment by one

Mode	Syntax	HEX	Length	Time
Implied	INC	$1a	1	2
Zero Page	INC $44	$e6	2	5
Zero Page, X	INC $44,X	$f6	2	6
Absolute	INC $4400	$ee	3	6
Absolute, X	INC $4400,X	$fe	3	7

FLAGS -	N	V	-	B	D	I	Z	C

INX

Increment X by one

Mode	Syntax	HEX	Length	Time
Implied	INX	$e8	1	2

FLAGS -	N	V	-	B	D	I	Z	C

INY

Increment Y by one

Mode	Syntax	HEX	Length	Time
Implied	INY	$c8	1	2

FLAGS -	N	V	-	B	D	I	Z	C

JMP

Jump

Mode	Syntax	HEX	Length	Time
Absolute	JMP $4400	$4c	3	3
(Indirect)	JMP ($4400)	$6c	3	6
Absolute, X	JMP $4400,X	$7c	3	6

FLAGS -	N	V	-	B	D	I	Z	C

JSR

Jump to Subroutine

Mode	Syntax	HEX	Length	Time
Absolute	JSR $4400	$20	3	6

FLAGS -	N	V	-	B	D	I	Z	C

LDA

Load Accumulator

Mode	Syntax	HEX	Length	Time
Immediate	LDA #$44	$a9	2	2
Zero Page	LDA $44	$a5	2	3
Zero Page, X	LDA $44,X	$b5	2	4
Absolute	LDA $4400	$ad	3	4
Absolute, X	LDA $4400,X	$bd	3	4 (+1 if page crossed)
Absolute, Y	LDA $4400,Y	$b9	3	4 (+1 if page crossed)
(Indirect)	LDA ($4400)	$b2	2	5
(Indirect, X)	LDA ($4400,X)	$a1	2	6
(Indirect), Y	LDA ($4400),Y	$b1	2	5 (+1 if page crossed)

FLAGS - N | V | - | B | D | I | Z | C

LDX

Load X

Mode	Syntax	HEX	Length	Time
Immediate	LDX #$44	$a2	2	2
Zero Page	LDX $44	$a6	2	3
Zero Page, Y	LDX $44,Y	$b6	2	4
Absolute	LDX $4400	$ae	3	4
Absolute, Y	LDX $4400,Y	$be	3	4 (+1 if page crossed)

FLAGS - N | V | - | B | D | I | Z | C

LDY

Load Y

Mode	Syntax	HEX	Length	Time
Immediate	LDY #$44	$a0	2	2
Zero Page	LDY $44	$a4	2	3
Zero Page, X	LDY $44,X	$b4	2	4
Absolute	LDY $4400	$ac	3	4
Absolute, X	LDY $4400,X	$bc	3	4 (+1 if page crossed)

FLAGS - N | V | - | B | D | I | Z | C

LSR

Logical Shift Right

Mode	Syntax	HEX	Length	Time
Implied	LSR	$4a	1	2
Zero Page	LSR $44	$46	2	5
Zero Page, X	LSR $44,X	$56	2	6
Absolute	LSR $4400	$4e	3	6
Absolute, X	LSR $4400,X	$5e	3	6 (+1 if page crossed)

FLAGS - N | V | - | B | D | I | Z | C

NOP

No Operation

Mode	Syntax	HEX	Length	Time
Implied	NOP	$ea	1	2

FLAGS - N | V | - | B | D | I | Z | C

ORA

OR with Accumulator

Mode	Syntax	HEX	Length	Time
Immediate	ORA #$44	$09	2	2
Zero Page	ORA $44	$05	2	3
Zero Page, X	ORA $44,X	$15	2	4
Absolute	ORA $4400	$0d	3	4
Absolute, X	ORA $4400,X	$1d	3	4 (+1 if page crossed)
Absolute, Y	ORA $4400,Y	$19	3	4 (+1 if page crossed)
(Indirect)	ORA ($4400)	$12	2	5
(Indirect, X)	ORA ($4400,X)	$01	2	6
(Indirect), Y	ORA ($4400),Y	$11	2	5 (+1 if page crossed)

FLAGS - N | V | - | B | D | I | Z | C

PHA

Push Accumulator to Stack

Mode	Syntax	HEX	Length	Time
Implied	PHA	$48	1	3

FLAGS - N | V | - | B | D | I | Z | C

PHP

Push Status Register to stack

Mode	Syntax	HEX	Length	Time
Implied	PHP	$08	1	3

FLAGS -	N	V	-	B	D	I	Z	C

PHX

Push X to Stack

Mode	Syntax	HEX	Length	Time
Implied	PHX	$da	1	3

FLAGS -	N	V	-	B	D	I	Z	C

PHY

Push Y to Stack

Mode	Syntax	HEX	Length	Time
Implied	PHY	$5a	1	3

FLAGS -	N	V	-	B	D	I	Z	C

PLA

Pull Accumulator from Stack

Mode	Syntax	HEX	Length	Time
Implied	PLA	$68	1	4

FLAGS -	N	V	-	B	D	I	Z	C

PLP

Pull Status Register from stack

Mode	Syntax	HEX	Length	Time
Implied	PLP	$28	1	4

FLAGS -	N	V	-	B	D	I	Z	C

PLX

Pull X from Stack

Mode	Syntax	HEX	Length	Time
Implied	PLX	$fa	1	4

FLAGS -	N	V	-	B	D	I	Z	C

PLY

Pull Y from Stack

Mode	Syntax	HEX	Length	Time
Implied	PLY	$7a	1	4

FLAGS - N V - B D I Z C

RMB

Reset Memory Bit

Mode	Syntax	HEX	Length	Time
Bit 0 - Zero Page	RMB0, $44	$07	2	5
Bit 1 - Zero Page	RMB1, $44	$17	2	5
Bit 2 - Zero Page	RMB2, $44	$27	2	5
Bit 3 - Zero Page	RMB3, $44	$37	2	5
Bit 4 - Zero Page	RMB4, $44	$47	2	5
Bit 5 - Zero Page	RMB5, $44	$57	2	5
Bit 6 - Zero Page	RMB6, $44	$67	2	5
Bit 7 - Zero Page	RMB7, $44	$77	2	5

FLAGS - N V - B D I Z C

ROL

Rotate Left

Mode	Syntax	HEX	Length	Time
Implied	ROL	$2a	1	2
Zero Page	ROL $44	$26	2	5
Zero Page, X	ROL $44,X	$36	2	6
Absolute	ROL $4400	$2e	3	6
Absolute, X	ROL $4400,X	$3e	3	6 (+1 if page crossed)

FLAGS - N V - B D I Z C

ROR

Rotate Right

Mode	Syntax	HEX	Length	Time
Implied	ROR	$6a	1	2
Zero Page	ROR $44	$66	2	5
Zero Page, X	ROR $44,X	$76	2	6
Absolute	ROR $4400	$6e	3	6
Absolute, X	ROR $4400,X	$7e	3	6 (+1 if page crossed)

FLAGS - N V - B D I Z C

RTI

Return from Interrupt

Mode	Syntax	HEX	Length	Time
Implied	RTI	$40	1	6

FLAGS - N V - B D I Z C

RTS

Return from Subroutine

Mode	Syntax	HEX	Length	Time
Implied	RTS	$60	1	6

FLAGS - N V - B D I Z C

SBC

Subtract from Accumulator with Borrow

Mode	Syntax	HEX	Length	Time
Immediate	SBC #$44	$e9	2	2
Zero Page	SBC $44	$e5	2	3
Zero Page, X	SBC $44,X	$f5	2	4
Absolute	SBC $4400	$ed	3	4
Absolute, X	SBC $4400,X	$fd	3	4 (+1 if page crossed)
Absolute, Y	SBC $4400,Y	$f9	3	4 (+1 if page crossed)
(Indirect)	SBC ($4400)	$f2	2	5
(Indirect, X)	SBC ($4400,X)	$e1	2	6
(Indirect), Y	SBC ($4400),Y	$f1	2	5 (+1 if page crossed)

FLAGS - N V - B D I Z C

SEC

Set Carry

Mode	Syntax	HEX	Length	Time
Implied	SEC	$38	1	2

FLAGS - N V - B D I Z **C**

SED

Set Decimal Mode

Mode	Syntax	HEX	Length	Time
Implied	SED	$f8	1	2

FLAGS - N V - B **D** I Z C

SEI

Set Interrupt Disable

Mode	Syntax	HEX	Length	Time
Implied	SEI	$78	1	2

FLAGS - N V - B D **I** Z C

SMB

Set Memory Bit

Mode	Syntax	HEX	Length	Time
Bit 0 - Zero Page	SMB0, $44	$87	2	5
Bit 1 - Zero Page	SMB1, $44	$97	2	5
Bit 2 - Zero Page	SMB2, $44	$a7	2	5
Bit 3 - Zero Page	SMB3, $44	$b7	2	5
Bit 4 - Zero Page	SMB4, $44	$c7	2	5
Bit 5 - Zero Page	SMB5, $44	$d7	2	5
Bit 6 - Zero Page	SMB6, $44	$e7	2	5
Bit 7 - Zero Page	SMB7, $44	$f7	2	5

FLAGS - N V - B D I Z C

STA

Store Accumulator

Mode	Syntax	HEX	Length	Time
Zero Page	STA $44	$85	2	4
Zero Page, X	STA $44,X	$95	2	5
Absolute	STA $4400	$8d	3	5
Absolute, X	STA $4400,X	$9d	3	6
Absolute, Y	STA $4400,Y	$99	3	6
(Indirect)	STA ($4400)	$92	2	6
(Indirect, X)	STA ($4400,X)	$81	2	7
(Indirect), Y	STA ($4400),Y	$91	2	7

FLAGS - N V - B D I Z C

STP

Stop

Mode	Syntax	HEX	Length	Time
Implied	STP	$db	1	2

FLAGS - N V - B D I Z C

STX

Store X

Mode	Syntax	HEX	Length	Time
Zero Page	STX $44	$86	2	4
Zero Page, Y	STX $44,Y	$96	2	5
Absolute	STX $4400	$8e	3	5

FLAGS - N V - B D I Z C

STY

Store Y

Mode	Syntax	HEX	Length	Time
Zero Page	STY $44	$84	2	4
Zero Page, X	STY $44,X	$94	2	5
Absolute	STY $4400	$8c	3	5

FLAGS - N V - B D I Z C

STZ

Store Zero

Mode	Syntax	HEX	Length	Time
Zero Page	STZ $44	$64	2	4
Zero Page, X	STZ $44,X	$74	2	5
Absolute	STZ $4400	$9c	3	5
Absolute, X	STZ $4400,X	$9e	3	6

FLAGS - N V - B D I Z C

TAX

Transfer Accumulator to X

Mode	Syntax	HEX	Length	Time
Implied	TAX	$aa	1	2

FLAGS - N V - B D I Z C

TAY

Transfer Accumulator to Y

Mode	Syntax	HEX	Length	Time
Implied	TAY	$a8	1	2

FLAGS - N V - B D I Z C

TRB

Test and Reset Bits

Mode	Syntax	HEX	Length	Time
Zero Page	TRB $44	$14	2	5
Absolute	TRB $4400	$1c	3	6

FLAGS - N V - B D I Z C

TSB

Test and Set Bits

Mode	Syntax	HEX	Length	Time
Zero Page	TSB $44	$04	2	5
Absolute	TSB $4400	$0c	3	6

FLAGS - N V - B D I Z C

TSX

Transfer Stack Pointer to X

Mode	Syntax	HEX	Length	Time
Implied	TSX	$ba	1	2

FLAGS -	N	V	-	B	D	I	Z	C

TXA

Transfer X to Accumulator

Mode	Syntax	HEX	Length	Time
Implied	TXA	$8a	1	2

FLAGS -	N	V	-	B	D	I	Z	C

TXS

Transfer X to Stack Pointer

Mode	Syntax	HEX	Length	Time
Implied	TXS	$9a	1	2

FLAGS -	N	V	-	B	D	I	Z	C

TYA

Transfer Y to Accumulator

Mode	Syntax	HEX	Length	Time
Implied	TYA	$98	1	2

FLAGS -	N	V	-	B	D	I	Z	C

WAI

Wait for Interrupt

Mode	Syntax	HEX	Length	Time
Implied	WAI	$cb	1	5

FLAGS -	N	V	-	B	D	I	Z	C

4.2 ROM functions

The ROM file found on GitHub has many standard functions that may be of use when creating your own programs.

ROM functions are called via JSR $XXXX.

4.2.1 Read_Chunk

Purpose: Reads a 512 byte chunk of a large file
Call address: $FEF4
Communication via: Memory addresses
Error return: None
Stack requirements:
Registers affected: A, X, Y
Description: This reads a 512 byte chunk of memory to $0800
Example:

```
JSR $FEF4 ; Call Write_Chunk
```

4.2.2 Write_Chunk

Purpose: Writes a 512 byte chunk of a large file
Call address: $FEF8
Communication via: Memory addresses
Error return: None
Stack requirements:
Registers affected: A, X, Y
Description: This writes a 512 byte chunk of memory from $0800
Example:

```
JSR $FEF8 ; Call Write_Chunk
```

4.2.3 InitBlob

Purpose: Creates a large file
Call address: $FEFC
Communication via: Memory addresses
Error return: None
Stack requirements:
Registers affected: A, X, Y
Description: This creates a file on the SDcard based on the Filename and Filesize set in memory
Example: Creates a example.bin of 1MB

```
JSR $FEFC ; Call InitBlob
```

4.2.4 UpdateDisplay

Purpose: Updates the Seven Segment Display
Call address: $FF10
Communication via: Memory addresses
Error return: None
Stack requirements: 4 bytes
Registers affected: A, X, Y
Description: This routine takes the three bytes at $D0, $D1, $D2 and displays them on the Seven Segment Display
Example: Update the display to read "010203"

```
LDA #$01
STA $D0
LDA #$02
STA $D1
LDA #$03
STA $D2
JSR $FF10 ; Call UpdateDisplay
```

4.2.5 ReadKeypad

Purpose: Reads the keypad, with debounce
Call address: $FF14
Communication via: A Register
Error return: None
Stack requirements: Unknown
Registers affected: A, X, Y
Description: Returns the keycode of any key pressed. Keycode is returned in A register. This will hold execution until key is released
Example: Loop until the "5" key is pressed and released

```
Loop:
   JSR $FF14 ; Call ReadKeypad
   CMP #$05 ; Compare A with 05
   BNE Loop ; Branch if not equal
```

4.2.6 ScanKeypad

Purpose: Returns current key pressed
Call address: $FF18
Communication via: A Register
Error return: None
Stack requirements: Unknown
Registers affected: A, X, Y
Description: Returns the keycode on any key pressed, will return immediately and so can miss a short key press.
Example: Loop until the "5" key is pressed

```
Loop:
   JSR $FF18 ; Call ScanKeypad
   CMP #$05 ; Compare A with 05
   BNE Loop ; Branch if not equal
```

4.2.7 GetRandomByte

Purpose: Generates a random byte
Call address: $FF20
Communication via: A Register
Error return: None
Stack requirements: Unknown
Registers affected: A, X, Y
Description: This routine uses a linear congruential generator system to create a mostly random byte.
Example: Get random byte and display it

```
Loop:
   JSR $FF20 ; Call GetRandomByte
   STA $D0 ; Store in display buffer
   JSR $FF10 ; Call UpdateDisplay
```

4.2.8 Sleep_Long

Purpose: Sleeps for about 0.5 seconds
Call address: $FF30
Communication via: None
Error return: None
Stack requirements: None
Registers affected: X, Y
Description: This returns after about 0.5 seconds

4.2.9 Sleep_Short

Purpose: Sleeps for about 50 milliseconds
Call address: $FF34
Communication via: None
Error return: None
Stack requirements: None
Registers affected: X
Description: This returns after about 50 milliseconds

4.2.10 SPI_Write_Byte

Purpose: Writes one byte to the SPI bus
Call address: $FF40
Communication via: A Register
Error return: None
Stack requirements: None
Registers affected: A
Description: This routine writes the byte in the A register to the SPI bus
Example: Send $FF to the display.
Note: This will light all segments of the left most digit and NOT display $FF

```
Loop:
   JSR $FF58 ; Call SPI_Select_7seg
   LDA #$FF ; Setup to send $FF
   JSR $FF40 ; Call SPI_Write_Byte
   JSR $FF5C ; Call SPI_Unselect_7seg
```

4.2.11 SPI_Read_Byte

Purpose: Reads one byte from the SPI bus
Call address: $FF44
Communication via: A Register
Error return: None
Stack requirements: None
Registers affected: A, X, Y
Description: This routine reads a byte from the SPI bus to the A register. It also writes a null byte to the SPI bus at the same time.

4.2.12 SPI_Select_SDcard

Purpose: Selects the SD Card as the active device on the SPI bus
Call address: $FF50
Communication via: None
Error return: None
Stack requirements: None
Registers affected: A
Description: Sets the Chip Select line low for the SD Card
Note: The Chip Select line is Active low

4.2.13 SPI_Unselect_SDcard

Purpose: Unselects the SD Card as the active device on the SPI bus
Call address: $FF54
Communication via: None
Error return: None
Stack requirements: None
Registers affected: A
Description: Sets the Chip Select line high for the SD Card
Note: The Chip Select line is Active low

4.2.14 SPI_Select_7seg

Purpose: Selects the Seven Segment Display as the active device on the SPI bus
Call address: $FF58
Communication via: None
Error return: None
Stack requirements: None
Registers affected: A
Description: Sets the Chip Select line low for the Seven Segment Display
Note: The Chip Select line is Active low

4.2.15 SPI_Unselect_7seg

Purpose: Unselects the Seven Segment Display as the active device on the SPI bus
Call address: $FF5C
Communication via: None
Error return: None
Stack requirements: None
Registers affected: A
Description: Sets the Chip Select line high for the Seven Segment Display
Note: The Chip Select line is Active low

4.2.16 Init_SD_card

Purpose: Initialises the SD Card for use on SPI bus
Call address: $FF60
Communication via: None
Error return: Errors returned at "ErrorCode" ($0206)
Stack requirements: Unknown
Registers affected: A, X, Y
Description: This routine initialises the SD Card to work in SPI mode

4.2.17 SD_Card_Mount

Purpose: Gathers information about the file system ready for other file operations
Call address: $FF64
Communication via: None
Error return: Errors returned at "ErrorCode" ($0206)
Stack requirements: Unknown
Registers affected: A, X, Y
Description: This routine takes file system information from the SD Card and sets up the card for further file system operations

4.2.18 SD_Card_Read_Sector

Purpose: Reads a single sector from the SD Card.
Call address: $FF68
Communication via: Memory addresses.
Error return: Errors returned at "ErrorCode" ($0206)
Stack requirements: Unknown
Registers affected: A, X, Y
Description: This routine reads the sector of "CurrentSector" into memory pointed to by "ZP_SectorBufPTR". "ZP_SectorReadSize" is the number of bytes to read, 512 bytes is a full sector.
Example: Read sector $00000042 to location of $2000.

```
LDA #$00 ; Setup to read sector $00000042
STA CurrentSector ; MSB of sector
STA CurrentSector+1
STA CurrentSector+2
LDA #$42
STA CurrentSector+3 ; LSB of Sector
LDA #$00 ; Setup to read sector to $2000
STA ZP_SectorBufPTR ; Set buffer start pointer Low byte
LDA #$20 ; Setup to read sector to $2000
STA ZP_SectorBufPTR+1 ; Set buffer start pointer High byte
LDA #$00 ; Setup to read $200 bytes
STA ZP_SectorReadSize ; Set buffer end pointer Low byte
LDA #$02 ; Setup to read $200 bytes
STA ZP_SectorReadSize+1 ; Set buffer end pointer High byte
JSR $FF68 ; Read sector
```

4.2.19 SD_Card_Write_Sector

Purpose: Writes a sector to the SD Card
Call address: $FF6C
Communication via:
Error return: None
Stack requirements:
Registers affected:
Description: This routine writes a sector to the SD Card.
Example: TBA
Note: This function has not been written yet.

4.2.20 CreateFileName

Purpose: Creates a file name from a number
Call address: $FF70
Communication via: Memory addresses
Error return: None
Stack requirements:
Registers affected: A, X, Y
Description: This routine converts a single byte number into a ASCII string file name
Example: Converts $01 to "01.BIN", filename is stored at $0251

```
LDA #$01 ; Setup for file 01.BIN
STA $0250 ; Store 01 at FileNumber
JSR $FF70 ; Call CreateFileName
```

4.2.21 FindFile

Purpose: Finds the first cluster of a file for a file load operation
Call address: $FF74
Communication via: Memory addresses
Error return: Errors returned at "ErrorCode" ($0206)
Stack requirements:
Registers affected: A, X, Y
Description: This routine will search the directory of the SD Card for the file-name and return the first cluster number of the file. Returns cluster number via 4 bytes at $021B

4.2.22 LoadFile

Purpose: Loads a file into memory
Call address: $FF78
Communication via: Memory addresses
Error return: Errors returned at "ErrorCode" ($0206)
Stack requirements:
Registers affected: A, X, Y
Description: This routine loads "FileName" into memory pointed to by "LoadAddrPTR"
Example: Loads "01.BIN"

```
LDA #$01 ; Setup file number $01
STA $0250 ; FileNumber is stored at $0250
JSR $FF70 ; Convert FileNumber to FileName
JSR $FF60 ; Init the SD Card to SPI mode.
JSR $FF64 ; Mount SD Card to setup file system.
LDA #$00 ; Setup Load Address to $1000
STA $0260 ; Load address pointer (LSB) is stored at
   $0260
LDA #$10 ; Setup Load Address to $1000
STA $0261 ; Load Address pointer (MSB) is stored at
   $0261
JSR $FF78 ; Call LoadFile
```

4.2.23 LBA_Addr

Purpose: Calculates LBA sector from cluster number
Call address: $FF7C
Communication via: A, X, Y
Error return: None
Stack requirements:
Registers affected: A, X, Y
Description: This routine converts a cluster number to an LBA sector address as part of loading a file.
Note: Not typically used by end user

4.2.24 GetNextSector

Purpose: Finds the next sector in a file
Call address: $FF80
Communication via:
Error return: None
Stack requirements:
Registers affected: A, X, Y
Description: This routine finds the next sector in a file and is used as part of loading a file
Note: Not typically used by end user

4.2.25 FindNextCluster

Purpose: Finds the next cluster in a file
Call address: $FF84
Communication via:
Error return: None
Stack requirements:
Registers affected: A, X, Y
Description: This routine finds the next cluster in a file and is used as part of loading a file
Note: Not typically used by end user

4.2.26 BootStrap

Purpose: Checks the SD Card for a file and runs it or returns to the monitor ROM
Call address: $FF90
Communication via: Memory addresses
Error return: None
Stack requirements:
Registers affected: A, X, Y
Description: This routine is used as part of power up, it will check the SD Card for "FileNumber" and if it finds the file it will load it to $1000 and jump to $1000.

5 Detailed Design and Schematic

5.1 Catch the BUS

The word "Bus" is used to describe a number of wires used for a given purpose. In the Aluis 6502 we have 16 wires that make up the 16 bits of the Address Bus, we have 8 bits for the Data Bus, and we have 4 wires for the SPI Bus.

5.2 6502 - CPU

The heart of the Alius is the WDC65C02 CPU, which is the Western Design Center version of the classic MOS6502.
The CPU connects to the Address Bus and Data Bus, it has a Read/Write line and and IRQ line. The 1MHz Clock feeds into the CPU on pin 37.

5.3 RAM

RAM is the working memory of the computer. Software can change data in RAM at any time.
The two main types of RAM we could use are Dynamic RAM (DRAM) or Static RAM (SRAM).
DRAM requires a process to refresh the data many times per second, but it is cheap and high density.
SRAM holds the data as long as power is connected, but SRAM is more expensive.

The Alius 6502 system uses 32K of Static RAM in a single chip "62256". The choice of using SRAM was to keep the whole design simple.
The RAM connects to the Address and Data bus, but also has connections via the address decode logic chips. The address decoder chips feed to CS and OE lines which control if the RAM is active. last of all the RAM has a R/W line which controls if you are reading or writing RAM data.

5.4 ROM

The ROM is permanent storage for system software.
There are a few types of ROM but ROM that can be erased and re-written is the most useful.
In the early days of computers RAM was often "Mask ROM" in which the Data is part of the chip design and can never be changed.
UV Erasable ROM became common by the 1980's, with UV erasable ROM the chip can be made blank by exposure to UV light. You can tell that it's a UV erasable chip as they have a little windows on top (often covered by a sticker to stop accident erasure).
More modern ROM can be electrically erased and so lets you update the ROM software quickly and easily.
The Alius system uses a 32K of EEPROM in a AT28C256 chip.
Much like the RAM the ROM chip connects to the Address and Data bus and has connections via the address decode logic chips. Unlike the RAM the Read/Write line is tied to 5V as you can never write the the ROM during normal use.

The Alius 6502 is compatible with UV erasable ROM, but it is not as easy to use.

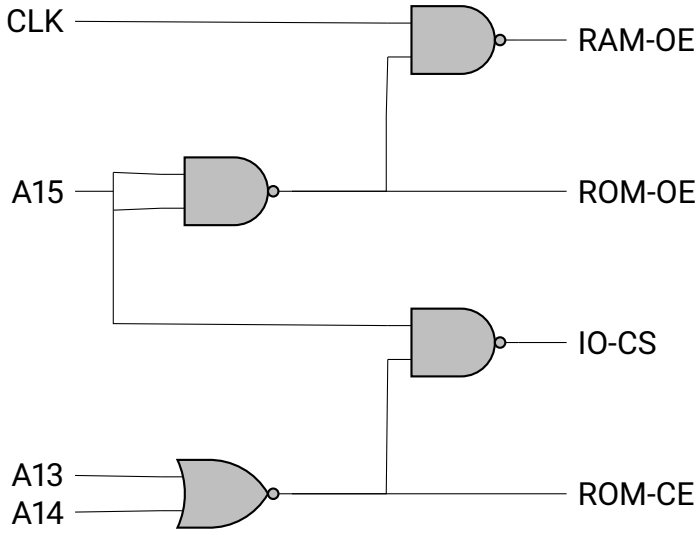
5.5 I/O - 65C22

5.6 Memory Address Decoding

An address decoding circuit is used by the system to “select” whether RAM, ROM or an I/O device is enabled when the processor is addressing a particular memory address, as specified by the system’s memory map.

All the chips connected to a Bus will have a Chip Select line to signal if that chip should be using the Bus at any given time.

A small logic circuit can use the Address Bus and create sevral Chip Select singals.
In the Alius 6502 design we have the 74HC00 and the 74HC02 chips which have four individual logic gates in each chip, which allows us to build the circuit shown below.



A15	A14	A13	
0	X	X	**RAM**
1	0	0	**I/O**
1	0	1	**ROM**
1	1	0	**ROM**
1	1	1	**ROM**

5.7 SPI bus

Serial Peripheral Interface (SPI) is a common way for low speed serial connections between devices.

5.7.1 SPI bus connections

The typical SPI bus needs 4 wires, 2 for data, one for clock and one for chip select. The lack of an official standard means that the names for the lines can be one of many ‘standards’

In most documentation you will see once device called ‘Master’ and the other as ‘Slave’. You will also see it documented as ‘Controller’ and ‘Peripheral’

The data from the Controller to the Peripheral can be marked as any one of:

* MOSI - Master Out Slave In.
* SIMO - Slave in Master out.
* SDI - Serial Data In.

The data from the Peripheral to the Controller can be marked as any one of:

* MISO - Master In Salve Out.
* SOMI - Slave Out Master In.
* SDO - Serial Data Out.

The clock line is mark as any of:

* CLK
* SCK
* SCLK
* SCL
* CLOCK

The Chip Select can be marked as any one of:

* SS
* CS
* CSN
* CE

The Chip Select line is active low, often written with a bar over the name.

5.7.2 SPI Modes

SPI can be used in one of four modes, each mode is just a difference in the clock phase and when data is latched by the SPI device.

Mode 0 the clock is idle low, and the data is latched on the rising edge of the clock.
Mode 1 the clock is idle low, and the data is latched on the falling edge of the clock.
Mode 2 the clock is idle high, and the data is latched on the falling edge of the clock.
Mode 3 the clock is idle high, and the data is latched on the rising edge of the clock.

5.8 Display

The Display is made up of six seven segment display modules, each module is driven by a 74HC595 shift register chip.

Each shift register chip will pass data along to the next shift register one bit at a time when the clock changes.

By selecting the display we can just sending 6 bytes on the SPI bus the display will be updated.

The ROM function will take a byte and "decode" it to turn on/off the right segments to display the byte.

5.9 SD Card

The SPI bus give access to the SD Card.
An SD Card runs in Mode 0.
An SD card runs at 3.3V and so the SD card board has a 5V to 3.3V level shiftier chip.

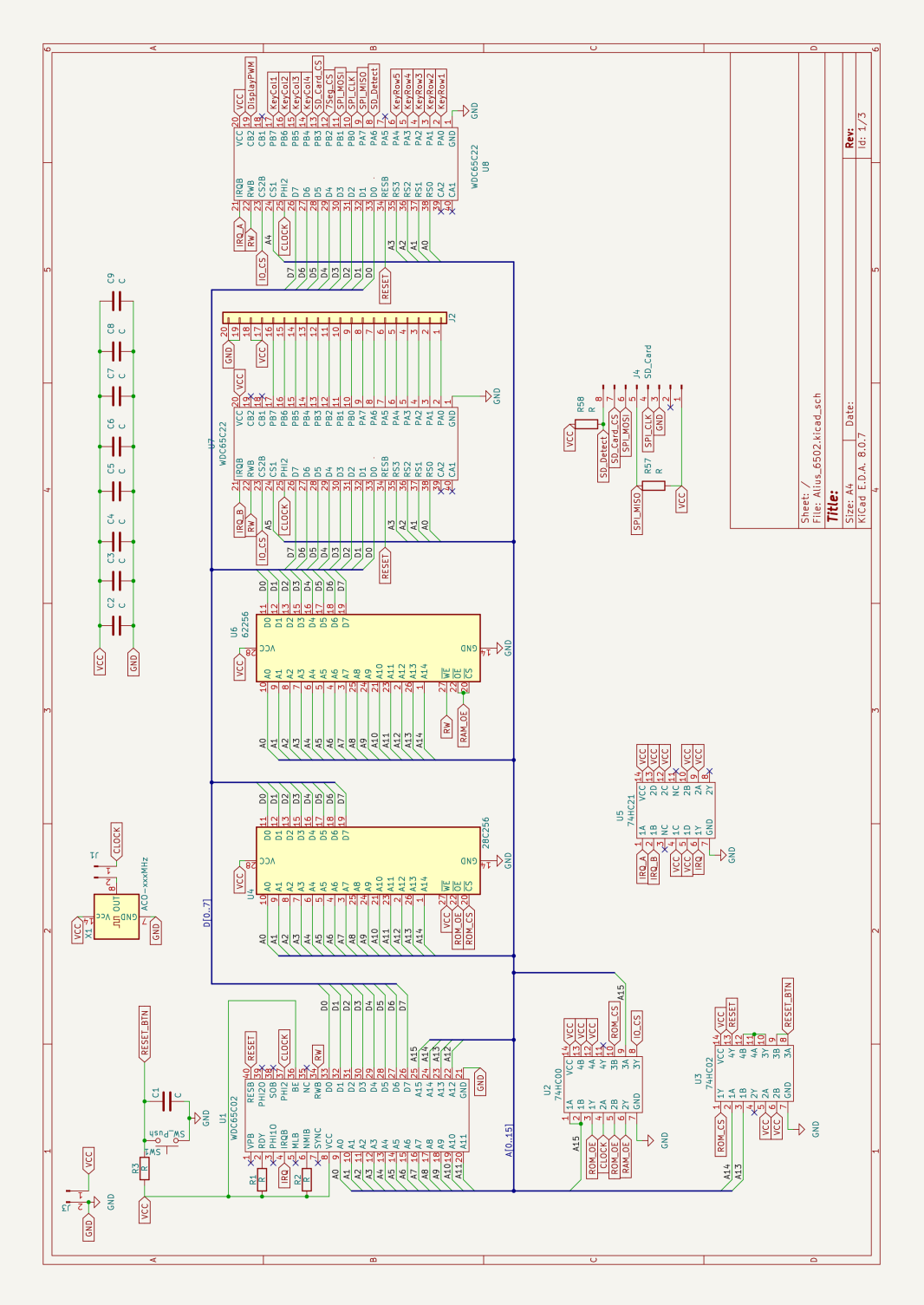
5.10 Keypad

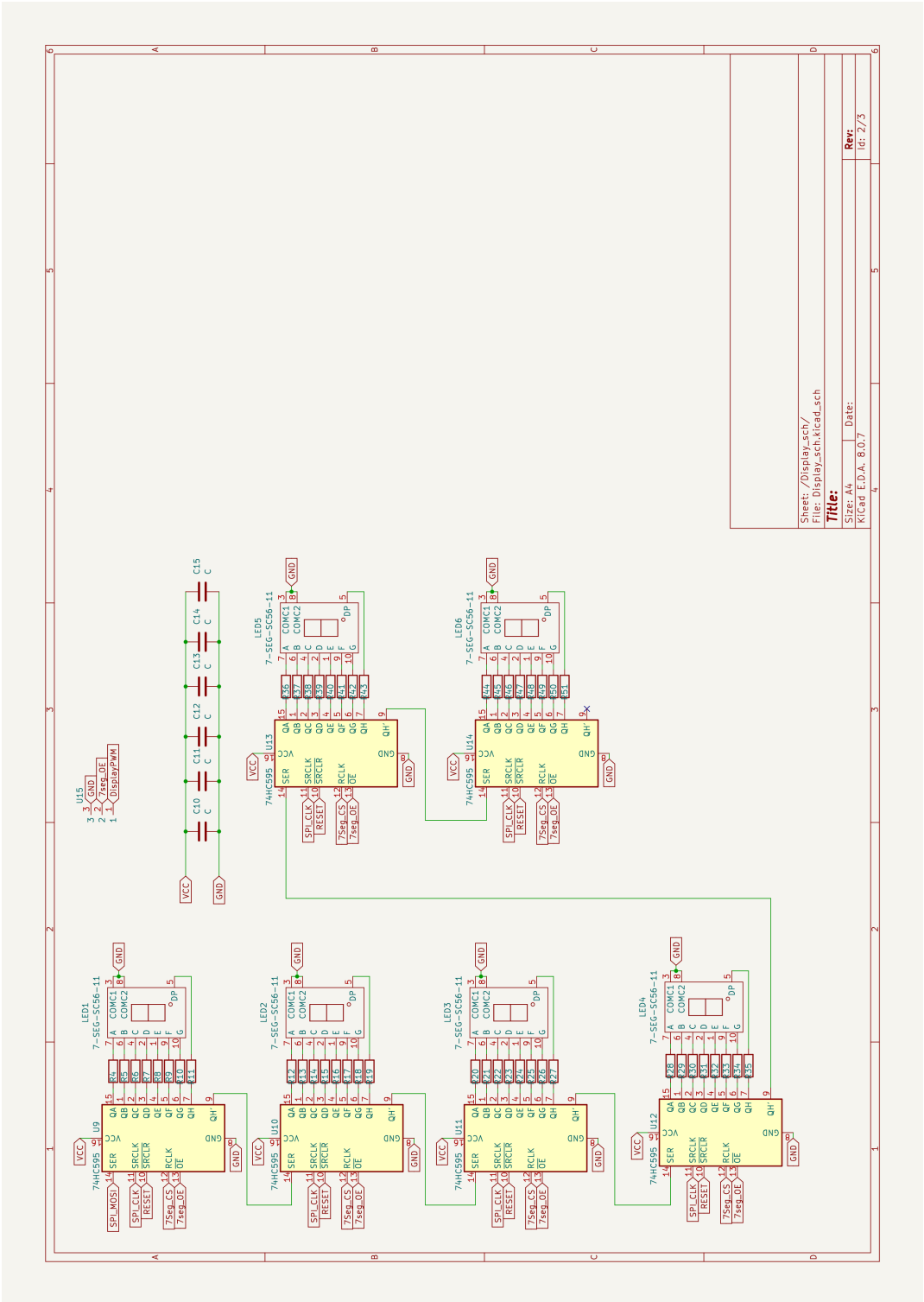
The keypad is setup as a row and column of wires, a push button at the intersection of each row and column.

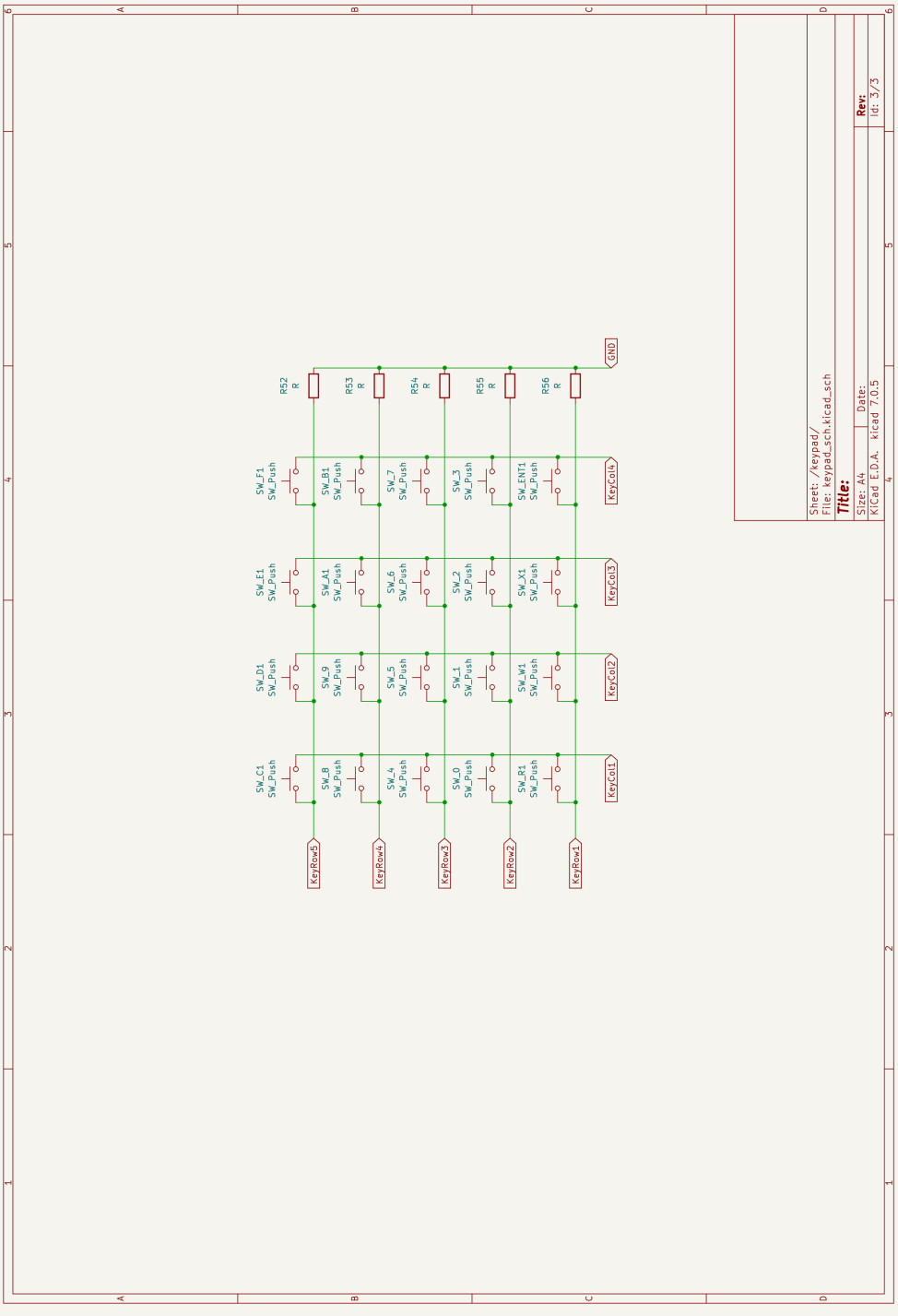
By setting a voltage on a given row we can then check if any assorted column is being pressed.

The Scan keypad function in the ROM will check every button in a fraction of a second.

5.11 Schematic



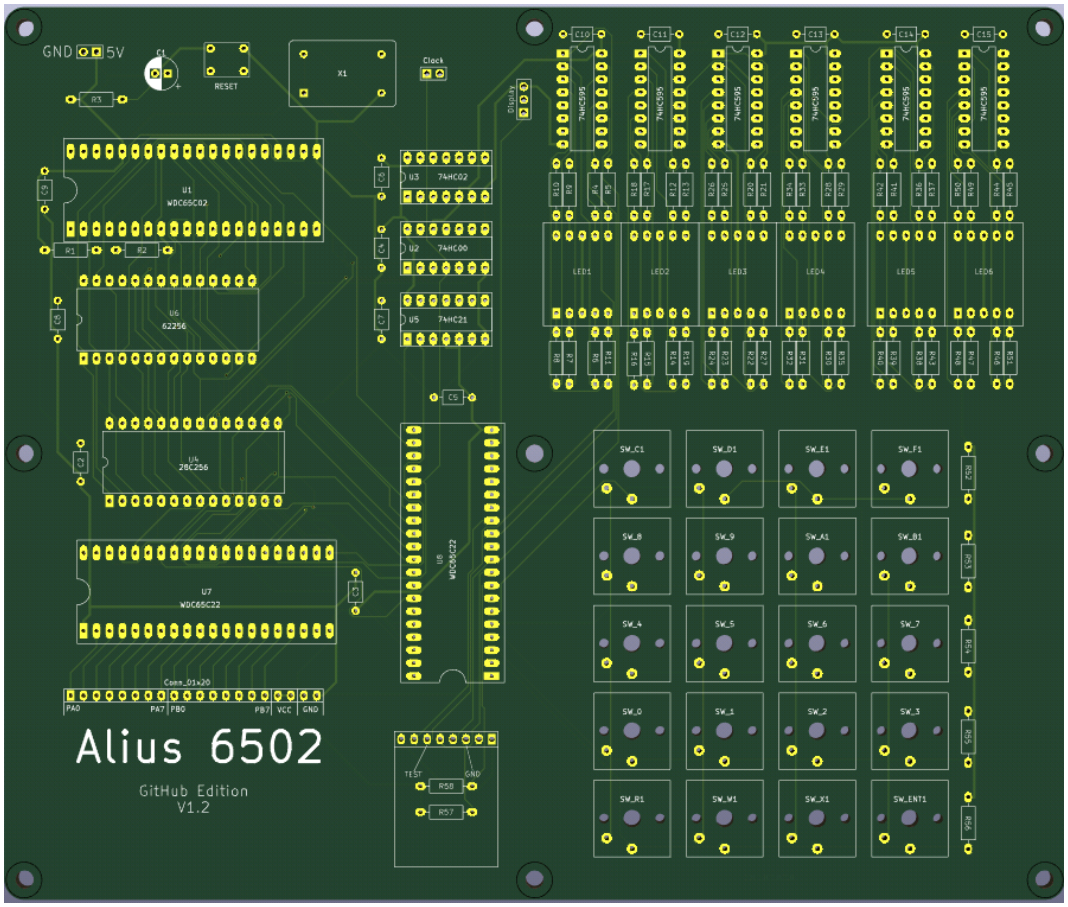

Sheet: /Display_sch/
File: Display_sch.kicad_sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. 8.0.7
Id: 2/3
LED1
LED2
LED3
LED4
LED5
LED6
74HC595
7-SEG-SC56-11
U9
U10
U11
U12
U13
U14
U15
C10
C11
C12
C13
C14
C15
VCC
GND
SPI_MOSI
SPI_CLK
RESET
7seg_CS
7seg_OE
DisplayPWM


GND
R52
R53
R54
R55
R56
SW_F1
SW_B1
SW_7
SW_3
SW_ENT1
SW_E1
SW_A1
SW_6
SW_2
SW_X1
SW_D1
SW_9
SW_5
SW_1
SW_W1
SW_C1
SW_8
SW_4
SW_0
SW_R1
SW_Push
KeyCol4
KeyCol3
KeyCol2
KeyCol1
KeyRow5
KeyRow4
KeyRow3
KeyRow2
KeyRow1
Sheet: /keypad/
File: keypad_sch.kicad_sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad 7.0.5
Id: 3/3

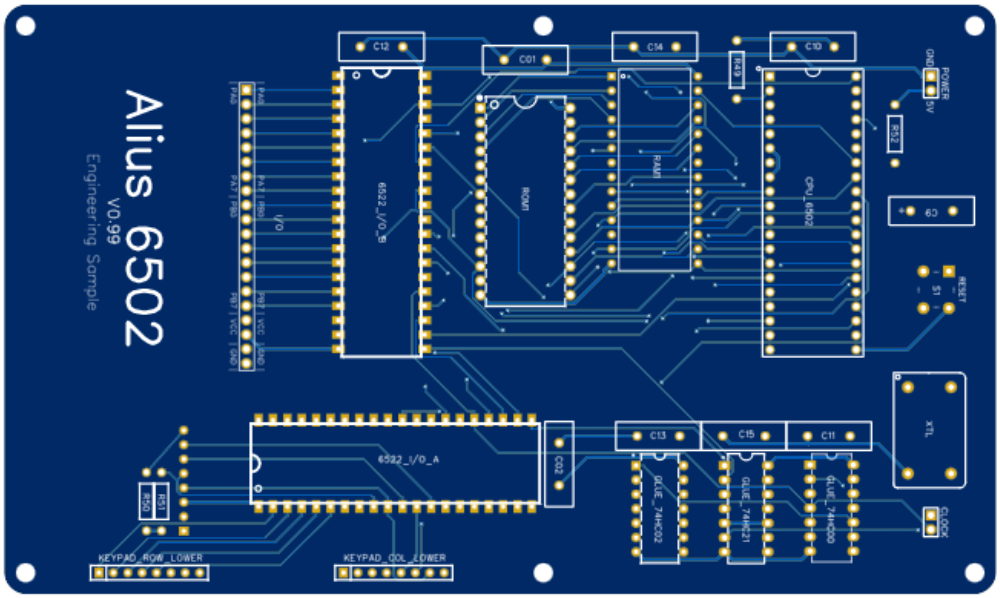
6 Building The System

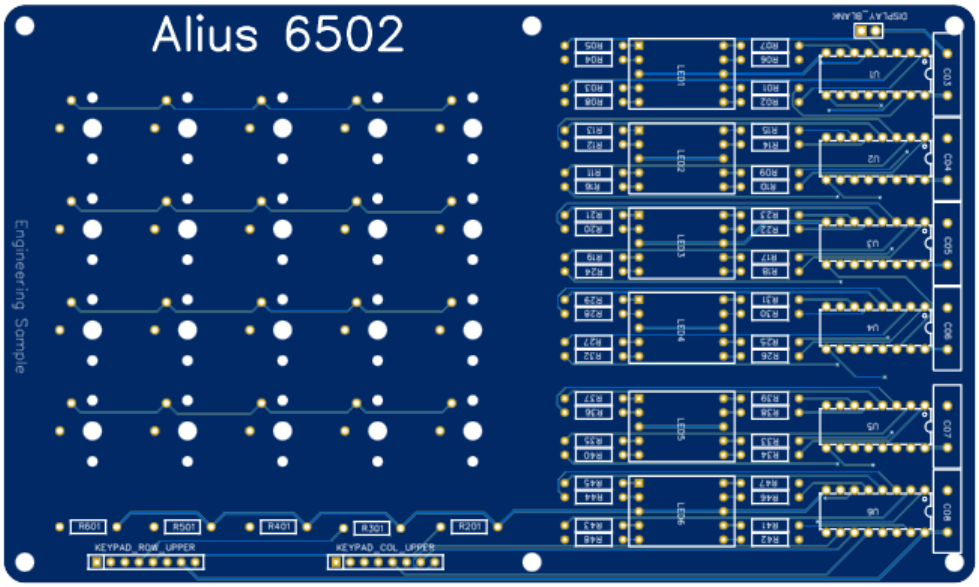
The Alius 6502 has two different form factors. The "Wide Board" is best for experiments and the "Stacked Board" makes a good base for a handheld system.

NOTE: The two form factors are fully software compatible.



Wide Board





Stacked Board

6.1 Parts List

Alius Parts List		
QTY	**Part**	**Description**
1 or 2	Printed Circuit Board(s)	Two boards for the stacked design
1	1MHz Crystal	
1	74HC02N	Quad NOR Gate
1	74HC00N	Quad NAND Gate
1	74HC21N	Dual 4-Input AND Gate
1	W65C02S	Main 6502 CPU
2	WDC65C22	6522 - VIA I/O
6	74HC595N	Shift register
6	SC56-11/5011AS	Seven segment red LED
1	AT28C256	32KB EEPROM
1	62256P	32KB static RAM
57	1KΩ resistor	
1	10μf capacitor	
14	0.1μf capacitor	
2	2 pin header	Used for power and clock
1	3 pin header	Used for display
1	8 pin header	for SD Card
1	SD Card module	Adafruit board
1	Reset switch	
1	28 pin ZIF socket	ZIF for ROM chip
20	Kailh Choc switch	
20	Kailh Choc key cap	
Extra parts for Stacked design		
2	8 pin header male	
2	8 pin header female	
6	M3 standoff	
12	M3 bolt	

Part numbers on chips will normally have letters or numbers before or after the listed part number. For example the "62256" might be marked as "MH62256LP-15"

6.2 Tools

Some tools are nice to have and other tools are a hard requirement, this list is in order of importance.

6.3 Soldering iron

6.4 Solder Remover

6.5 Side cutters

6.6 Screw Driver

6.7 Isopropyl Alcohol

6.8 Helping Hands

6.9 Multimeter

6.10 Magnifying Glass

6.11 Variable Power Supply

6.12 Oscilloscope

6.13 Construction

It is recommended that you start by checking you have all the parts listed above and ensure you can identify each part. Use of chip sockets will make any repair and changes easier.

The order of steps below has been tested as the best work flow, it also allows for testing during the construction.

For components with a large number of pins a good method of soldering them is to solder one pin at each corner to ensure correct fit before soldering all the pins.

Depending on your soldering skills and pace of work, the full construction will take 3 to 5 hours.

The project build is split into three sections and you can use this as a sensible way to break the project up over a few days.

Core Compute

In this section we will build and test the core of the computer.

Step 1
Install in 3 x 1KΩ resistors, marked on the board as R1, R2 and R3.

Step 2
Install reset switch S1.

Step 3
Install chip sockets for 65C02 CPU, 62256 RAM, 2 x 65C22 I/O chips, 74HC00, 74HC02, 74HC21

Step 4
Install the ZIF socket for the ROM. This should be mounted to have the lever on the right hand side of the socket.

> ⚠ Pin one of the chip is on the left side of the board, the lever is on thre right for easy access.

Step 5
Install capacitor C1

> ⚠ This is a 10μf tantalum capacitor, the polarity is marked on the board and on the part.

Step 6
Install 8 x 0.1μf capacitors, C2, C3, C4, C5, C6, C7, C8, C9.

Step 7
Install 1MHz crystal - marked as X1.
The polarity is marked by a dot on the board and one corner of the crystal can is not as rounded as the others.

Step 8
Install the 2 pin jumper for the clock.

Step 9
Install the 8 pin header for the SD Card.

Do not install the whole SD Card board at this point.

Step 10
Install a two pin header as power connector.

Step 11
Insert chips.
65C02 CPU, 62256 RAM, 2 x 65C22 I/O chips, 74HC02, 74HC00, 74HC21.

Step 12
Setup a test LED across SD Card I/O pins marked "Test" and "GND".

Step 13
Flash the ROM chip with test ROM.

Step 14
Power up, press reset button, and test.

If this stage has all worked correctly you should have the LED blinking about once per second.

If you do not have a blinking LED then read up on fault finding before proceeding.

The Display

In this section we will build and test the seven segment display.

Step 15
Install the 48 x 1KΩ resistors, R4 -> R51

Step 16
Install 6 x seven segment modules LED1 - > LED6.
Be careful to get the polarity correct, the decimal point on the display module is at the bottom right corner.

Step 17
Install sockets for chips U1 -> U6.

Step 18
Install 6 x 0.1μf capacitors C10 -> C15

Step 19
Install the 3 pin jumper for Display_Blank.
The jumper should be in the upper position.

If you have the "Wide" design then skip to step 23

Step 20 - "Stacked" design only
Install the 8 pin male headers on the lower board.

Step 21 - "Stacked" design only
Screw on the 6 x stand offs to the lower board.

Step 22 - "Stacked" design only
Stack the two boards and screw together to correctly position the female headers before soldering.

Step 23
Power on and test that all segments of the display light up one by one.

If you do not have each segment light up then read up on fault finding before proceeding.

The Keypad

In this section we will build the keypad and install the SDcard module.

Step 24
Install 5 x 1KΩ resistors, R52, R53, R54, R55, R56

Step 25
Install the 20 x key switches.

Step 26
Label the 20 x key caps.

Step 27
Install the 20 x key caps.

The SD Card

Step 28
Install 2 x 1kΩ resisters R57, R58

Step 29
Install SD Card board.

Step 30
Flash the ROM chip with Monitor ROM.

Step 31
Power up and test.

6.14 Flashing the ROM

TL866 II Plus EEPROM Programmer

6.15 Fault Finding

Any electronics project comes with the need for some fault finding. Having everything work first time is not guaranteed. In this section we will look at the most common things to check for issues.

6.15.1 Check Power

One of the more frustrating faults is when nothing at all happens.
Check for 5V of power on the board. A good test point is the right end of the expansion header. (VCC/GND)

If you don't detect 5V then check the power is working. Also check what amperage is being drawn by the board, under 200mA is expected.

A high power draw is a sign of a short circuit.

6.15.2 Check Soldering

The system has in the range of 500 solder joints and most of them are critical to having the system work. A visual inspection under good lighting will find many issues.

Look for joints that are missing, joints that have bridged to a next door connections.

Removing excessive solder is best done with solder wick, which lets you draw off some solder from the joint.

6.15.3 Check chips

It is important that you have the correct chips in the right places, but also inserted in the correct orientation.

Bent pins can also happen, you can often straighten pins with tweezers. Just be aware that the pin will never be as strong and might bend again.

6.15.4 Check jumpers

The clock jumper is essential to correct operation, and the Display_Blank jumper is needed to have the display work correctly.

7 Glossary

CMOS - Complementary Metal–Oxide–Semiconductor. This is a type of microchip construction.

CPU - Central Processing Unit. This is the main chip in a computer and it is where all the math and logic is processed.

EEPROM - Electrically Erasable Programmable Read-Only Memory. This is a chip that can hold data that is not lost when power is removed (non-volatile) but can be reprogrammed as required.

FAT32 - FAT stands for File Allocation Table, FAT32 is a version of the FAT file system created by Microsoft in 1995 and has become a standard for use with SD Cards.

I/O - Input/Output. This is the way to get data into and out of a computer.

IRQ - Interrupt Request. This is a system in which external hardware can request that the CPU stops running the current code and runs special interrupt handling code.

LBA - Logical Block Addressing. This is a common way of addressing which sector or block of data to access on a file system.

LED - Light Emitting Diode. A semiconductor device that emits light.

LSB - Least Significant Bit or Least Significant Byte. The LSB is sometimes referred to as the low-order bit or right-most bit.

MHz - Megahertz. Hertz is a measure of cycles per second, 1 Megahertz is one million cycles per second.

MOS - MOS is short for Metal Oxide Semiconductor, but also can relate to MOS Technology, Inc. who are most well known for designing the 6502 CPU.

MSB - Most Significant Bit or Most Significant Byte. The MSB is sometimes referred to as the high-order bit or left-most bit.

RAM - Random Access Memory. This is a type of memory that allows data to be accessed in any order and allows for the change of a single byte.

ROM - Read Only Memory. A type of memory that can't be changed and keeps the data if power is removed.

SD Card - Secure Digital Card. A removable card of memory storage that can be electrically erased and reprogrammed. Often used in place of a hard drive in portable electronics.

SPI - Serial Peripheral Interface. A communication interface specification used for simple data interfaces.

ZIF - Zero Insertion Force. A special type of chip socket that allows easy insertion and removal of a computer chip. Often used for a ROM chip to allow for easy upgrades.