theWickedWebDev/8-bit-computer

# Trilobyte CPU

## User Manual V0.1

2022 Trilobyte Computer // Stephen Young stephenyoung7267@gmail.com

# Special Thanks

# Objective

# Table of Contents

# Block Diagrams

# Hardware

## Architectural Overview

# Instruction

## Current Instruction

This is an 8-bit register who gets its value from either memory data bus, or the data 8-bit bus.  It stores the current instruction that was loaded. Unless an interrupt routine is actively running, this is the instruction currently being executed.

The input of the instruction register has three potential values: a hardwired `reset vector`, an `interrupt request vector` or from the `memory data bus`.

The `reset` or `interrupt request` addresses can only be asserted when there is an active `interrupt` or the `reset` line was pulled low, in addition to the `step zero flag (sz)` is set. Otherwise, the instruction register will load its value from the `memory data bus`.

### Reset & Interrupt Request Vectors

| **Vector Name** | **Address** | **Description** |
|---|---|---|
| Reset Handler | `0xFE` | Loaded into the instruction register which then runs the reset sequence. |
| Interrupt Request Handler | `0xFF` | Loaded into the instruction register which then runs the interrupt request handler sequence. |

## Step Counter

The step counter is a 4-bit counter register that counts up by 1 on every off-clock (low clock). It's value is asserted directly into the lower 4 bits of address space on the microcode eeproms. It is used to step through each operation of a given instruction. An instruction can have up to 16 different steps before looping back around again. The last step should trigger a valid reset step counter signal. This allows the CPU to move onto the next instruction without having to execute NOP's for the remaining unused steps.

### Step Zero Flag

This is active when the current value of the step counter equals 0x0. This is only used to allow one of the vector addresses to be conditionally loaded.

## Instruction Decoder

### Microcode

Microcode is stored on two separate EEPROMs. Both of them share the same address, like 8x2 or a 16-bit control word.

The following schema is how the micro-opcode address is constructed:

| **Microcode EEPROM Address** | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| INSTRUCTION (8-bits) | | | | | | | | STEP (4-bits) | | | |

## Microcode Decode Logic

### Micro-opcodes

The following control words describe operations that are performed while executing instructions. Each step in every instruction consists of a pair of control words. One from the read group and one from the write group. Together they form a 16-bit control word. There are also special control words that can be added onto their given group at the same time. For example, in the read group you can reset the step counter and assert the C register to the data bus at the same time.

| READs | | | WRITEs | | |
|---|---|---|---|---|---|
| | Description | μ | | Description | μ |
| NOP | No operation | 0x0 | NOP | No operation | 0x0 |
| CO | Assert C register to data bus | 0x1 | CI | Load C register from data bus | 0x1 |
| DO | Assert D register to data bus | 0x2 | DI | Load D register from data bus | 0x2 |
| S1O | Assert Scratch1 register to data bus | 0x3 | S1I | Load Scratch1 register from data bus | 0x3 |
| S2O | Assert Scratch2 register to data bus | 0x4 | S2I | Load Scratch2 register from data bus | 0x4 |
| S1S2O | Assert Scratch1 to MSB of address bus and Scratch2 to LSB of address bus | 0x5 | FI | Load F register from address bus | 0x5 |
| FO | Assert F register to the address bus | 0x6 | S3I | Load Scratch3 register from address bus | 0x6 |
| S3O | Assert Scratch3 to the address bus | 0x7 | AI | Load Accumulator from data bus | 0x10 |
| S3LO | Assert LSB of Scratch3 to the data bus | 0x8 | BI | Load Operand from data bus | 0x11 |
| S3HO | Assert MSB of Scratch3 to the data bus | 0x9 | MARI | Load memory address register from address bus | 0x12 |
| nc | Not connected | 0xA | CSI | Load code segment register from lower 4-bits of the data bus | 0x13 |
| nc | Not connected | 0xB | DSI | Load data segment register from the lower 4-bits of the data bus | 0x14 |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| `nc` | Not connected | 0xC |  | `SSI` | Load stack segment register from the lower 4-bits of the data bus | 0x15 |
| `nc` | Not connected | 0xD |  | `MDI` | Write data to memory from the data bus | 0x16 |
| `nc` | Not connected | 0xE |  | `MSI` | Write to the stack from the data bus | 0x17 |
| `nc` | Not connected | 0xF |  | `FLGI` | Load flags register from the ALU | 0x18 |
| `MCO` | Assert code memory to data bus | 0x10 |  | `RFLG` | Loads flags register from the lower 4-bits of the data bus | 0x19 |
| `MDO` | Assert data memory to data bus | 0x11 |  | `IQPRI` | Loads the interrupt priority register from the data bus | 0x1a |
| `MSO` | Assert stack memory to data bus | 0x12 |  | `IQMKI` | Loads the interrupt mask register from the data bus | 0x1b |
| `CSO` | Assert code segment to lower 5 bits of the data bus | 0x13 |  | `IOI` | Write a byte of data to an input device | 0x1c |
| `DSO` | Assert data segment to lower 5 bits of the data bus | 0x14 |  | `IOMI` | Write a byte of memory to a cartridge | 0x1d |
| `SSO` | Assert stack segment to lower 5 bits of the data bus | 0x15 |  | `nc` | Not connected | 0x1e |
| `PCO` | Assert program counter to the address bus | 0x16 |  | `nc` | Not connected | 0x1f |
| `SPO` | Assert stack pointer to the address bus | 0x17 |  | `IRI` | Loads instruction register with the value presented on the data bus | 0x20 |
| `FLGO` | Assert flags to the lower 4-bits of the data bus | 0x18 |  | `PCI` | Loads the program counter from the address bus | 0x21 |
| `IOO` | Read and assert a byte of data from an IO Device to the data bus | 0x19 |  | `DECPC` | Decrements the program counter | 0x22 |
| `IOMO` | Read and assert a byte of data from a cartridge onto the data bus | 0x1a |  | `SPI` | Loads the stack pointer from the address bus | 0x23 |
| `IRQO` | Reads IRQ port to the lower 7-bits of the data bus | 0x1b |  | `DECSP` | Decrements the stack pointer | 0x24 |
| `S4O` | Asserts Scratch4 register onto the address bus | 0x1c |  | `INCSP` | Increments the stack pointer | 0x25 |
| `S4L` | Asserts the LSB of the Scratch4 register onto the address bus | 0x1d |  | `S4I` | Loads Scratch4 register from the address bus | 0x26 |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| S4H | Asserts the MSB of the Scratch4 register onto the address bus | 0x1e |  | DISPI | Loads displacement register from the address bus | 0x27 |
| nc | Not connected | 0x1f |  | nc | Not connected | 0x28 |
| AO | Asserts the accumulator onto the data bus | 0x20 |  | nc | Not connected | 0x29 |
| ADD | Performs an add with the accumulator and operand, and then asserts result onto the data bus | 0x21 |  | nc | Not connected | 0x2a |
| ADC | Performs an add with carry with the accumulator and operand, and then asserts result onto the data bus | 0x22 |  | nc | Not connected | 0x2b |
| SUB | Performs a subtract with the accumulator and operand, and then asserts result onto the data bus | 0x23 |  | nc | Not connected | 0x2c |
| SBB | Performs a subtract with borrow with the accumulator and operand, and then asserts result onto the data bus | 0x24 |  | nc | Not connected | 0x2d |
| INC | Increment the accumulator and asserts result onto the data bus | 0x25 |  | nc | Not connected | 0x2e |
| DEC | Decrement the accumulator and asserts result onto the data bus | 0x26 |  | nc | Not connected | 0x2f |
| AND | ANDs the accumulator with the operand register and asserts the result onto the data bus | 0x27 |  | JP | Jump | 0x30 |
| OR | ORs the accumulator with the operand register and asserts the result onto the data bus | 0x28 |  | JLE | Jump if less than or equal to | 0x31 |
| XOR | XORs the accumulator with the operand register and asserts the result onto the data bus | 0x29 |  | JNG | Jump if not greater than | 0x31 |
| SHL | Shifts the accumulator left on the accumulator by one place and asserts the result onto the data bus | 0x2a |  | JG | Jump if greater than | 0x32 |
| SHR | Shifts the accumulator right on the accumulator by | 0x2b |  | JNLE | Jump if not less than or equal to | 0x32 |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
|  | one place and asserts the result onto the data bus |  |  |  |  |  |
| ASL | Performs an arithmetic shift left on the accumulator by one place and asserts result onto the data bus | 0x2c |  | JGE | Jump if greater than or equal | 0x33 |
| ASR | Performs an arithmetic shift right on the accumulator by one place and asserts result onto the data bus | 0x2d |  | JNL | Jump if not less than | 0x33 |
| ROR | Rotates the accumulator right by one place and asserts the result onto the data bus | 0x2e |  | JL | Jump if less than | 0x34 |
| ROL | Rotates the accumulator left by one place and asserts the result onto the data bus | 0x2f |  | JNGE | Jump if not greater than or equal to | 0x34 |
| NOT | Inverts the accumulator and asserts value onto the data bus | 0x30 |  | JA | Jump if above | 0x35 |
| NADD | Asserts an inverted add result onto the data bus | 0x31 |  | JNBE | Jump if not below or equal to | 0x35 |
| NADC | Asserts an inverted add with carry result onto the data bus | 0x32 |  | JBE | Jump if below or equal to | 0x36 |
| NSUB | Asserts an inverted subtract result onto the data bus | 0x33 |  | JNA | Jump if not above | 0x36 |
| NSBB | Asserts an inverted subtract with borrow result onto the data bus | 0x34 |  | JNB | Jump if not below | 0x37 |
| NINC | Increments the accumulator and asserts an inverted result onto the data bus | 0x35 |  | JAE | Jump if above or equal to | 0x37 |
| NDEC | Decrements the accumulator and asserts an inverted result onto the data bus | 0x36 |  | JNC | Jump if not carry | 0x37 |
| NAND | NANDs the accumulator with the operand register and asserts the result onto the data bus | 0x37 |  | JB | Jump if below | 0x38 |
| NOR | NORs the accumulator with the operand register and asserts the result onto the data bus | 0x38 |  | JNAE | Jump if not above or equal to | 0x38 |

| | | | | | | |
|---|---|---|---|---|---|---|
| XNOR | XNORs the accumulator with the operand register and asserts the result onto the data bus | 0x39 | | JC | Jump if carry | 0x38 |
| NSHL | Shifts the accumulator left on the accumulator by one place and asserts the inverted result onto the data bus | 0x3a | | JNE | Jump if not equal to | 0x39 |
| NSHR | Shifts the accumulator right on the accumulator by one place and asserts the inverted result onto the data bus | 0x3b | | JNZ | Jump if not zero | 0x39 |
| NASL | Performs an arithmetic shift left on the accumulator by one place and asserts an inverted result onto the data bus | 0x3c | | JE | Jump if equal to | 0x3a |
| NASR | Performs an arithmetic shift right on the accumulator by one place and asserts an inverted result onto the data bus | 0x3d | | JZ | Jump if zero | 0x3a |
| NROR | Rotates the accumulator right by one place and asserts an inverted result onto the data bus | 0x3e | | JNS | Jump if not sign | 0x3b |
| NROL | Rotates the accumulator left by one place and asserts an inverted result onto the data bus | 0x3f | | JS | Jump if sign | 0x3c |
| | | | | JNO | Jump if not overflow | 0x3d |
| | | | | JO | Jump if overflow | 0x3e |
| | | | | nc | Not connected | 0x3f |
| **Special Function** | | | | **Special Function** | | |
| OFST | Enable offset | 0x40 | | INCPC | Increment program counter | 0x40 |
| RSTSP | Reset Step Counter | 0x80 | | nc | Not connected | 0x80 |
| nc | Not connected | 0xc0 | | RST | Software reset | 0xc0 |

## General Purpose Registers

The Trilobyte CPU contains two 8-bit general-purpose registers and one 16-bit general-purpose counter register that are available for user control. The GPR module itself contains an additional two 8-bit scratch registers and one 16-bit scratch register.

| Name | Size | Visibility | Mnemonic |
|---|---|---|---|
| **C** | 8-bit | Public | r, r8 |
| **D** | 8-bit | Public | r, r8 |
| **F** | 16-bit | Public | r16 |
| **S1** | 8-bit | Private | r, r8 |
| **S2** | 8-bit | Private | r, r8 |
| **S3** | 16-bit | Private | r16 |

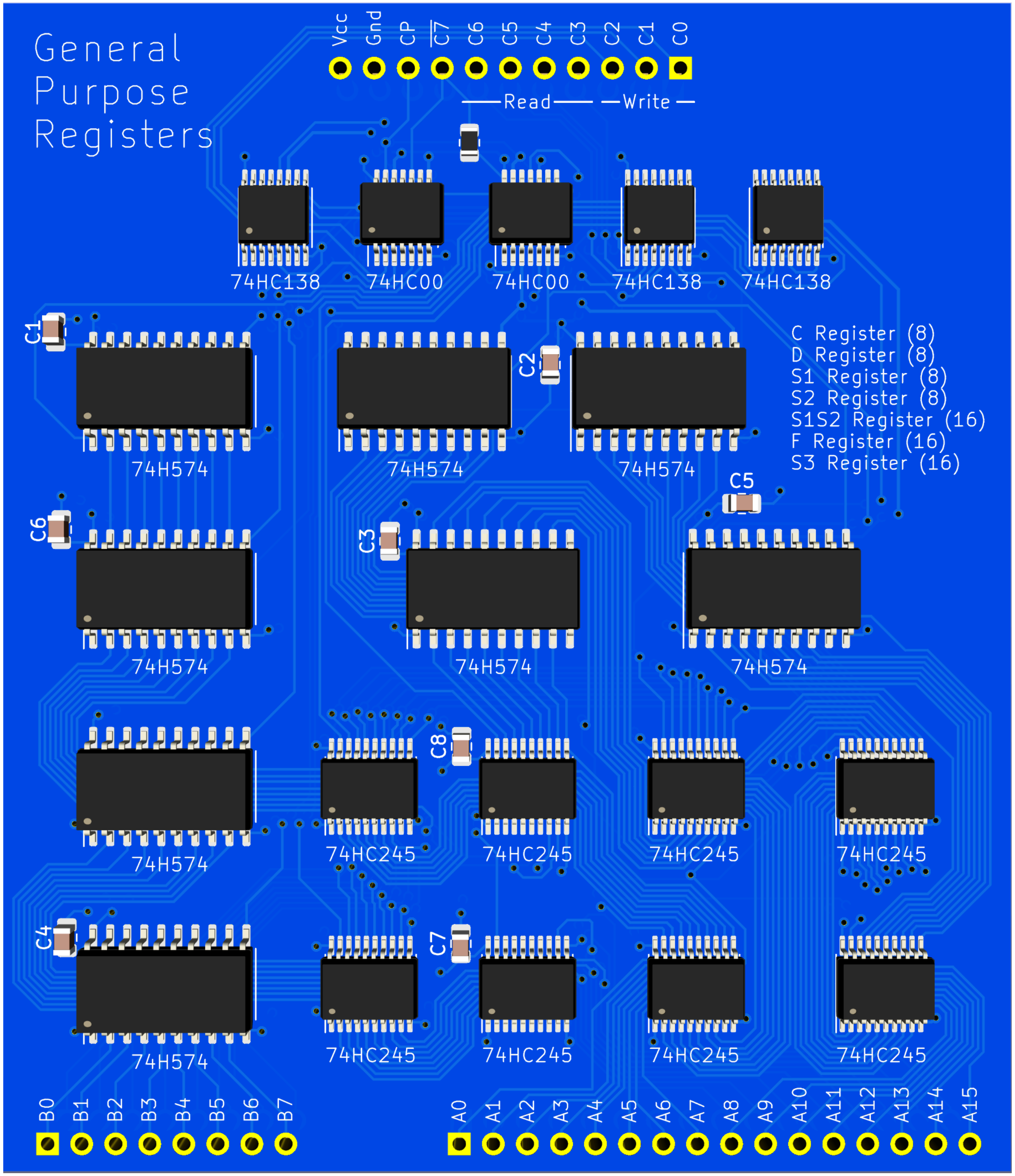
- **C, D**: Loads and asserts data from/to the 8-bit data bus
- **F**: Loads and asserts data from/to the 16-bit data bus
- **S1**: Loads and asserts data from/to the 8-bit data bus, as well as, asserts its data to the LSB of the 16-bit address bus.**S2**: Loads and asserts dat
- a from/to the 8-bit data bus, as well as, asserts its data to the MSB of the 16-bit address bus.
- **S3**: Loads and asserts data from/to the 16-bit data bus, as well as, asserts its MSB, or LSB to the 8-bit data bus.

# GPR Load & Assert Controls

| Description | [C0..C6] (Read_Write) | C7 | Opcode |
|---|---|---|---|
| Reset / Load ALI | 0000_000 | 0 | 0x0 |
| Load C | 0000_001 | 1 | 0x81 |
| Load D | 0000_010 | 1 | 0x82 |
| Load S1 | 0000_011 | 1 | 0x83 |
| Load S2 | 0000_100 | 1 | 0x84 |
| Load F | 0000_101 | 1 | 0x85 |
| Load S3 | 0000_110 | 1 | 0x86 |
| Assert C | 0001_000 | 1 | 0x88 |
| Assert D | 0010_000 | 1 | 0x90 |
| Assert S1 | 0011_000 | 1 | 0x98 |
| Assert S2 | 0100_000 | 1 | 0xa0 |
| Assert S1S2 | 0101_000 | 1 | 0xa8 |
| Assert F | 0110_000 | 1 | 0xb0 |

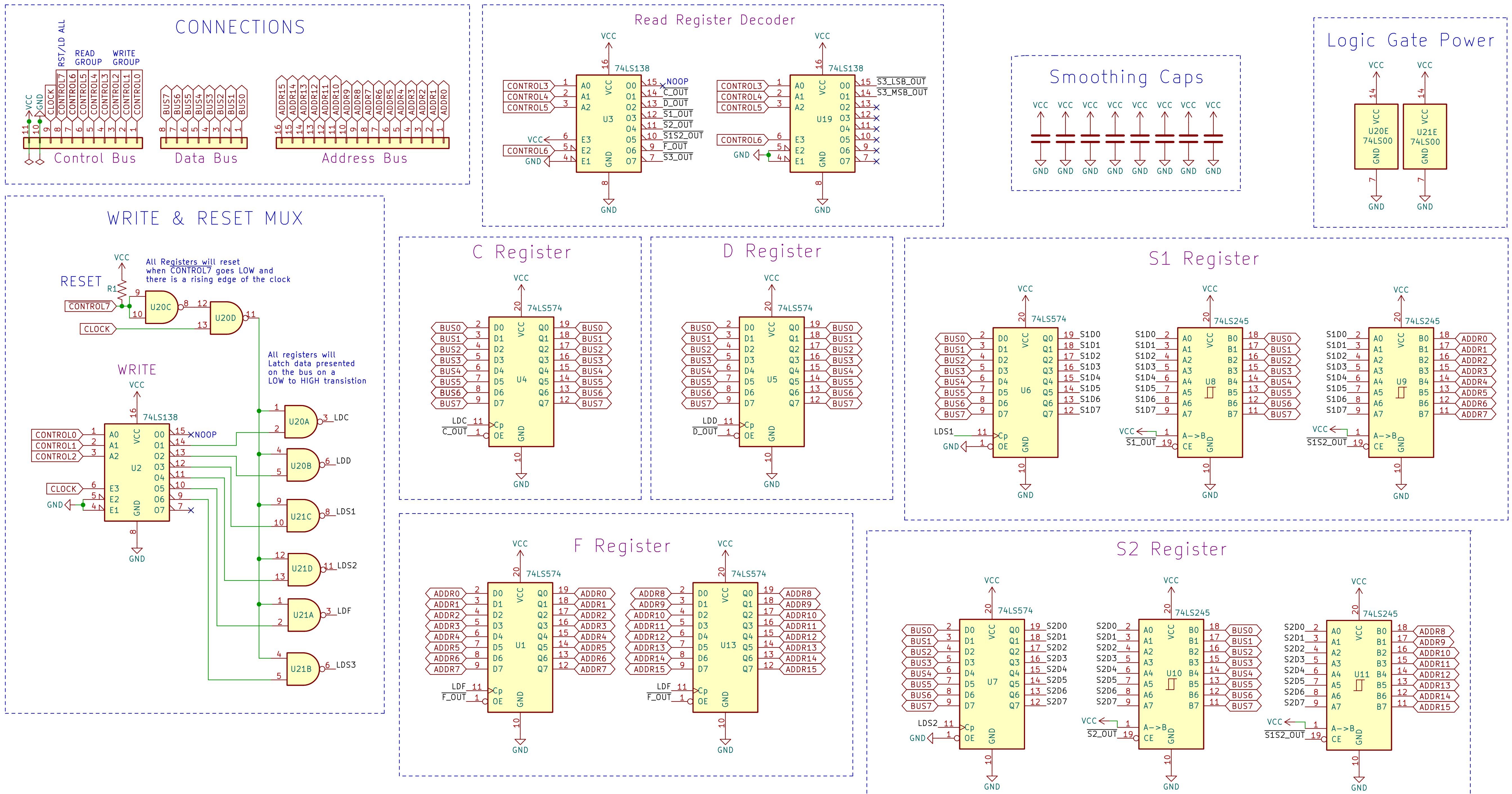
| | | | |
|---|---|---|---|
| Assert S3 | 0111_000 | 1 | 0xb8 |
| Assert S3L | 1000_000 | 1 | 0xc0 |
| Assert S3H | 1001_000 | 1 | 0xc8 |
| MOV C, D | 0010_001 | 1 | 0x91 |
| MOV C, S1 | 0011_001 | 1 | 0X99 |
| MOV C, S2 | 0100_001 | 1 | 0xa1 |
| MOV C, S3L | 1000_001 | 1 | 0xc1 |
| MOV C, S3H | 1001_001 | 1 | 0xc9 |
| MOV D, C | 0001_010 | 1 | 0x8a |
| MOV D, S1 | 0011_010 | 1 | 0x9a |
| MOV D, S2 | 0100_010 | 1 | 0xa2 |
| MOV D, S3L | 1000_010 | 1 | 0xc2 |
| MOV D, S3H | 1001_010 | 1 | 0xca |
| MOV S1, C | 0001_011 | 1 | 0x8b |
| MOV S1, D | 0010_011 | 1 | 0x9a |
| MOV S1, S2 | 0100_011 | 1 | 0x93 |

| | | | |
|---|---|---|---|
| MOV S1, S3L | 1000_011 | 1 | 0xc3 |
| MOV S1, S3H | 1001_011 | 1 | 0xcb |
| MOV S2, C | 0001_100 | 1 | 0x8c |
| MOV S2, D | 0010_100 | 1 | 0x94 |
| MOV S2, S1 | 0011_100 | 1 | 0x9c |
| MOV S2, S3L | 1000_100 | 1 | 0xc4 |
| MOV S2, S3H | 1001_100 | 1 | 0xcc |
| MOV F, S3 | 01111_101 | 1 | 0xbd |
| MOV F, S1S2 | 0101_101 | 1 | 0xad |
| MOV S3, F | 0110_110 | 1 | 0xb6 |
| MOV S3, S1S2 | 0101_110 | 1 | 0xae |


General
Purpose
Registers
Vcc
Gnd
CP
C7
C6
C5
C4
C3
C2
C1
C0
Read
Write
74HC138
74HC00
74HC00
74HC138
74HC138
C1
C2
74H574
74H574
74H574
C Register (8)
D Register (8)
S1 Register (8)
S2 Register (8)
S1S2 Register (16)
F Register (16)
S3 Register (16)
C5
C6
C3
74H574
74H574
74H574
C8
74H574
74HC245
74HC245
74HC245
74HC245
C4
C7
74H574
74HC245
74HC245
74HC245
74HC245
B0
B1
B2
B3
B4
B5
B6
B7
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
A13
A14
A15

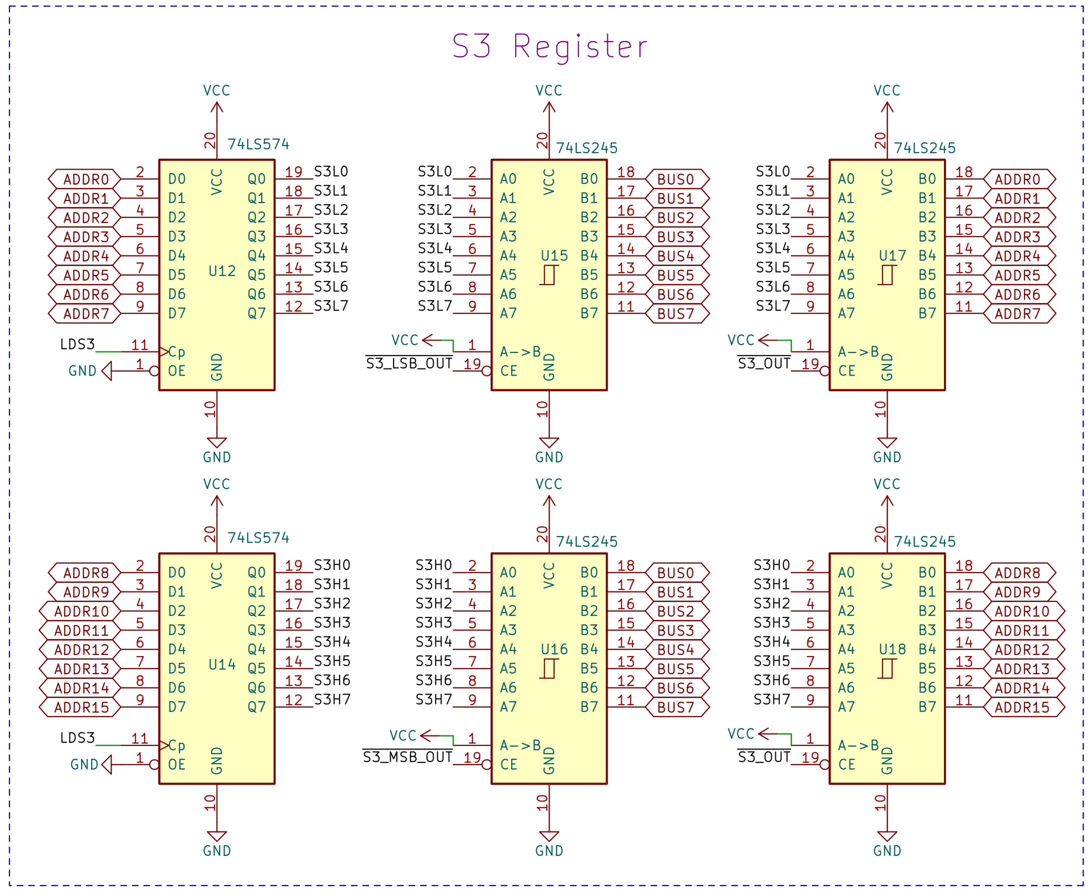
# GENERAL PURPOSE REGISTER ARRAY

## 8bit & 16bit C$^8$, D$^8$, S1$^8$, S2$^8$, S1S2$^{16}$, F$^{16}$, S3$^{16}$



All Registers will reset when CONTROL7 goes LOW and there is a rising edge of the clock

All registers will Latch data presented on the bus on a LOW to HIGH transition



## Control Word Truth Table

To just ASSERT or LOAD a value to the data bus to be used with external modules, just select a single value from either READ or WRITE

To do a GPR to GPR transfer just select a READ and WRITE control word and then OR them together.

To do a RESET, $\overline{\text{CONTROL7}}$ is set LOW and all data buses are set to 0x0, which causes each register to load the value presented on their data bus.

## *GPR <-> External*

| MOV R, * | MOV *, R |
|---|---|
| 0x81: LDC | 0x80: NOOP |
| 0x82: LDD | 0x88: CO |
| 0x83: LDS1 | 0x90: DO |
| 0x84: LDS2 | 0x98: S1OB |
| 0x85: LDF | 0xa0: S2OB |
| 0x86: LDS3 | 0xa8: S1S2OA |
| | 0xb0: FO |
| | 0xb8: S3O |
| | 0xc0: S3LOB |
| | 0xc8: S3HOB |

## *GPR to GPR Control Words*

| | | |
|---|---|---|
| MOV C, D | 0x91 | (0b1_0010_001) |
| MOV C, S1 | 0x99 | (0b1_0011_001) |
| MOV C, S2 | 0xa1 | (0b1_0100_001) |
| MOV C, S3LSB | 0xc1 | (0b1_1000_001) |
| MOV C, S3MSB | 0xc9 | (0b1_1001_001) |
| MOV D, C | 0x8a | (0b1_0001_010) |
| MOV D, S1 | 0x9a | (0b1_0011_010) |
| MOV D, S2 | 0xa2 | (0b1_0100_010) |
| MOV D, S3LSB | 0xc2 | (0b1_1000_010) |
| MOV D, S3MSB | 0xca | (0b1_1001_010) |
| MOV S1, C | 0x8b | (0b1_0001_011) |
| MOV S1, D | 0x9a | (0b1_0010_011) |
| MOV S1, S2 | 0x93 | (0b1_0100_011) |
| MOV S1, S3LSB | 0xc3 | (0b1_1000_011) |
| MOV S1, S3MSB | 0xcb | (0b1_1001_011) |
| MOV S2, C | 0x8c | (0b1_0001_100) |
| MOV S2, D | 0x94 | (0b1_0010_100) |
| MOV S2, S1 | 0x9c | (0b1_0011_100) |
| MOV S2, S3LSB | 0xc4 | (0b1_1000_100) |
| MOV S2, S3MSB | 0xcc | (0b1_1001_100) |
| MOV F, S3 | 0xbd | (0b1_0111_101) |
| MOV F, S1S2 | 0xad | (0b1_0101_101) |
| MOV S3, F | 0xb6 | (0b1_0110_110) |
| MOV S3, S1S2 | 0xae | (0b1_0101_110) |

**theWickedWebDev/8-bit-computer**
Sheet: /
File: Minified-Gpr.kicad_sch

**Title: General Purpose Register Array**

Size: User  Date: 2022-02-10  **Rev: v2**
KiCad E.D.A. kicad (6.0.0-0)  Id: 1/1

# Special Purpose Registers

Program Counter

Stack Pointer

Memory Segment Registers

**Code Segment Register (CS)**

**Data Segment Register (DS)**

**Stack Segment Register (SS)**

# Flags Register

## Bit Order on Data Bus

| BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|
| Zero (ZF) | Overflow (OF) | Sign (SF) | Carry (CF) |

- **Flags Out** – Asserts the flags register onto the data bus.
- **Flags In** – Latches the flag's register with calculated values based on the ALU function's result. This is active for all ALU functions except CMP and TST
- **Restore Flags** – Used in conjunction with Flags In, this will latch the register with values asserted on the data bus. This is normally used when returning from an interrupt and popping the flags off of the stack.

### Flags Register Truth Table

| Flags Out | Flags In | Restore Flags | Description |
|---|---|---|---|
| 1 | 1 | n.c. | NOP |
| 0 | 1 | n.c. | Assert Flags to Data Bus |
| 1 | 0 | 0 | Latch flags from ALU. Used with mostly all ALU functions. |
| 1 | 0 | 1 | Latches/Restores flag data from the 8-bit data bus. Normally used while returning from an interrupt. |

# Interrupt Registers

## Interrupt Mask Register

## Interrupt Priority Register

# Offset/Displacement Register

# Timer Reload Registers

# Arithmetic Logic Unit

This portion of the CPU is responsible for all of the arithmetic, logic, and boolean functionality. It also includes a flags register as well as a conditional jump logic module. Internally it uses the A and B data buses to move local data around. It is also connected to the main 8-bit data bus to transfer values to the accumulator and operand registers as well as output the result of the given function. It is also used to restore the flags register during an interrupt return instruction

## Functions

| Function | Description | Flags Affected | Control |
|---|---|---|---|
| `PASS A` | Outputs the accumulator(A) to the data bus | | `0x20` |
| `INC` | Increment, A + 1 | CF, SF, OF, ZF | `0x25` |
| `DEC` | Decrement, A – 1 | CF, SF, OF, ZF | `0x26` |
| `ADD` | Addition, A + B | CF, SF, OF, ZF | `0x21` |
| `ADC` | Addition w/ Carry, A + B + Carry Flag | CF, SF, OF, ZF | `0x22` |
| `SUB` | Subtract, A – B | CF, SF, OF, ZF | `0x23` |
| `SBB` | Subtract w/ Carry, A – B – Carry Flag | CF, SF, OF, ZF | `0x24` |
| `NOT` | Invert A, ~A | | `0x30` |
| `AND` | Boolean AND, A & B | SF, ZF, OF: Cleared, CF: Cleared | `0x27` |
| `NAND` | Boolean NAND, ~(A & B) | SF, ZF, OF: Cleared, CF: Cleared | `0x35` |

| | | | |
|---|---|---|---|
| OR | Boolean OR, A \| B | SF, ZF, OF: Cleared, CF: Cleared | 0x28 |
| NOR | Boolean NOR, ~(A \| B) | SF, ZF, OF: Cleared, CF: Cleared | 0x38 |
| XOR | Boolean XOR, A ^ B | SF, ZF, OF: Cleared, CF: Cleared | 0x29 |
| XNOR | Boolean XNOR, ~(A ^ B) | SF, ZF, OF: Cleared, CF: Cleared | 0x39 |
| SHL | Shift Left, A << 1 | CF: Bit shifted out, SF, ZF, OF | 0x2a |
| SHR | Shift Right, A >> 1 | CF: Bit shifted out, SF, ZF, OF | 0x2b |
| ASL | Signed Arithmetic Shift Left, A * 2 | CF: Bit shifted out, SF, ZF, OF | 0x2c |
| ASR | Signed Arithmetic Shift Riht, A / 2 | CF: Bit shifted out, SF, ZF, OF | 0x2d |
| ROR | Rotate Right | CF: Bit shifted out, SF, ZF, OF | 0x2e |
| ROL | Rotate Left | CF: Bit shifted out, SF, ZF, OF | 0x2f |
| CMP | Compare, same as SUB but does not latch Flags | | 0x23* |
| TST | Test, same as AND but does not latch Flags | | 0x27* |

## Accumulator and Operand Registers

- **Latch Accumulator** – Loads the value present on the data bus into the Accumulator (A) Register to be used by the various ALU functions.
- **Latch Operand** – Loads the value present on the data bus into the Operand (B) Register to be used by the various ALU functions.

## Arithmetic Module

This module is built with two 4-bit Full Adders to perform addition on two 8-bit values. It uses two's-complement for Subtraction by XOR'ing the operand (B) with 1 and adding 1. It also has the ability to increment or decrement the accumulator.

Arithmetic Control Truth Table

| Instruction | Mux | Two's Comp | Carry In |
|---|---|---|---|
| `A + B` | 0 | 0 | 0 |
| `A - B` | 0 | 1 | 1 |
| `A + B + Carry` | 0 | 0 | Carry Flag Value |
| `A - B - Carry` | 0 | 1 | Carry Flag Value |
| `INC ( A + 1)` | 1 | 1 | 1 |
| `DEC ( A - 1)` | 1 | 0 | 0 |

Arithmetic Examples

| Function | A | B | CF | Result |
|---|---|---|---|---|
| `ADD` | 0b0011 | 0b0001 | n.c. | 0b0100 |
| `ADC` | 0b0011 | 0b0001 | 1 | 0b0101 |

| | | | | |
|---|---|---|---|---|
| SUB | 0b0011 | 0b0001 | n.c. | 0b0010 |
| SBB | 0b0011 | 0b0001 | 1 | 0b0001 |
| INC | 0b0011 | n.c. | n.c. | 0b0100 |
| DEC | 0b0011 | n.c. | n.c. | 0b0010 |

## Logic Gate Module

This module is responsible for executing logical bitwise operations. It contains only the AND, OR, and NOR functions. To acheive NAND, NOR, and XNOR the result is inverted at the ALU control unit.

Logic Gate Examples

| Function | A | B | Result |
|---|---|---|---|
| AND | 0b0101 | 0b0011 | 0b0001 |
| NAND* | 0b0101 | 0b0011 | 0b1110 |
| OR | 0b0101 | 0b0011 | 0b0111 |
| NOR* | 0b0101 | 0b0011 | 0b1000 |
| XOR | 0b0101 | 0b0011 | 0b0110 |
| XNOR* | 0b0101 | 0b0011 | 0b1001 |

Logic Gate Examples

*1 – Uses same function, however, the invert control bit is set to active. ie. and => nand

## Shift & Rotate Module

This module provides the ability to shift bits to the left or right. Can be done logically or arithmetically. The logical shift is unsigned (ignores MSB), whereas Arithmetic Shift is signed, and maintains the most significant bit. This is the equivalent of multiplying/dividing by 2.

Shift & Rotate Truth Table

| **Function** | **Shift Right** | **Shift Left** | **Rotate** | **Arithmetic** |
|---|---|---|---|---|
| `SHL` | 1 | 0 | 1 | 1 |
| `SHR` | 0 | 1 | 1 | 1 |
| `ASL` | 1 | 0 | 1 | 0 |
| `ASR` | 0 | 1 | 1 | 0 |
| `ROR` | 0 | 1 | 0 | 1 |
| `ROL` | 1 | 0 | 0 | 1 |

Shift & Rotate Examples

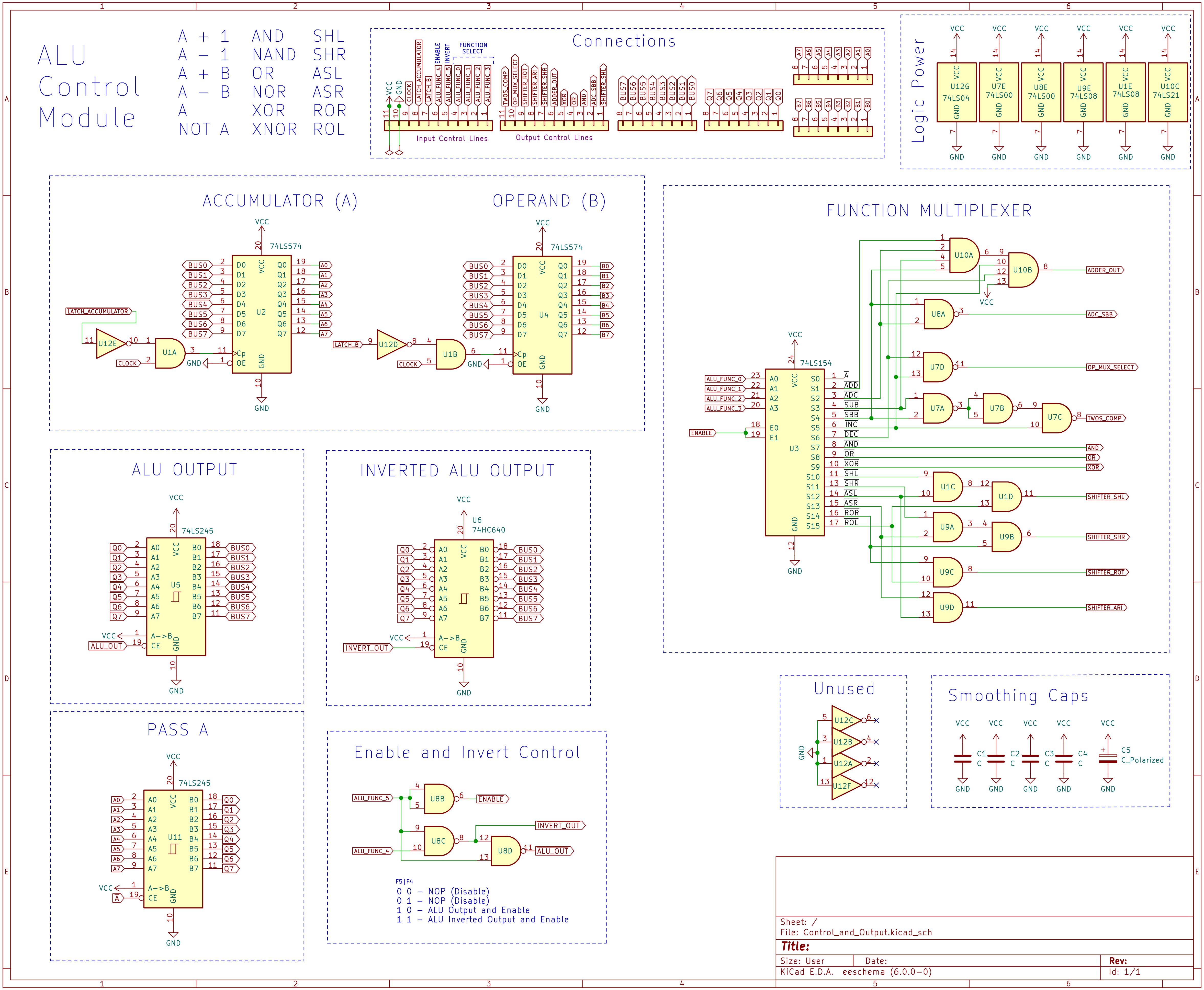
| Function | Operand | Result |
|---|---|---|
| SHL | 0b10001001 | 0b00010010 |
| SHR | 0b10001001 | 0b01000100 |
| ASL | 0b10001001 | 0b10010010 |
| ASR | 0b10001001 | 0b11000100 |
| ROR | 0b10001001 | 0b11000100 |
| ROL | 0b10001001 | 0b00010011 |

### Conditional Jump Logic

This module provides all of the conditions for calculating jumps based on ALU Flags. It uses a 4-bit control bus with enable, which is connected to the main CPU's microcode control logic, the ALU flags and outputs a single control line which is LOW if a jump is active.

| Mnemonic | Description | Flags | Control |
|---|---|---|---|
| jp | Jump | n/a | 0x0 |
| jle / jng | Jump if Less Than or Equal / Jump Not Greater | ZF = 1 or SF <> OF | 0x1 |
| jg / jnle | Jump if Greater / Jump if Not Less Than or Equal | ZF = 0 and SF = OF | 0x2 |
| jge / jnl | Jump if Greater Than or Equal / Jump if Not Lower | SF = OF | 0x3 |

| | | | |
|---|---|---|---|
| jl / jnge | Jump if Less Than / Jump if Not Greater Than or Equal | SF <> OF | 0x4 |
| ja / jnbe | Jump if Above / Jump if Not Below | CF = 0 and ZF = 0 | 0x5 |
| jbe / jna | Jump if Below / Jump if Not Above | CF = 1 or ZF = 1 | 0x6 |
| jnb / jae / jnc | Jump if not below / Jump if above or equal / Jump if not carry | CF = 0 | 0x7 |
| jb / jnae / jc | Jump if below / Jump if not above or equal / Jump if carry | CF = 1 | 0x8 |
| jne / jnz | Jump if not equal / Jump if not zero | ZF = 0 | 0x9 |
| je / jz | Jump if equal / Jump if zero | ZF = 1 | 0xa |
| jns | Jump if not sign | SF = 0 | 0xb |
| js | Jump if sign | SF = 1 | 0xc |
| jno | Jump if not overflow | OF = 0 | 0xd |
| jo | Jump if overflow | OF = 1 | 0xe |



# ALU Control Module

| | | |
|---|---|---|
| A + 1 | AND | SHL |
| A − 1 | NAND | SHR |
| A + B | OR | ASL |
| A − B | NOR | ASR |
| A | XOR | ROR |
| NOT A | XNOR | ROL |

F5|F4
0 0 − NOP (Disable)
0 1 − NOP (Disable)
1 0 − ALU Output and Enable
1 1 − ALU Inverted Output and Enable

Sheet: /
File: Control_and_Output.kicad_sch

**Title:**

Size: User Date: **Rev:**
KiCad E.D.A. eeschema (6.0.0−0) Id: 1/1

# Arithmetic Module

8bit Arithmetic Module provides
ADD, ADC, SUB, SBB, INC, DEC

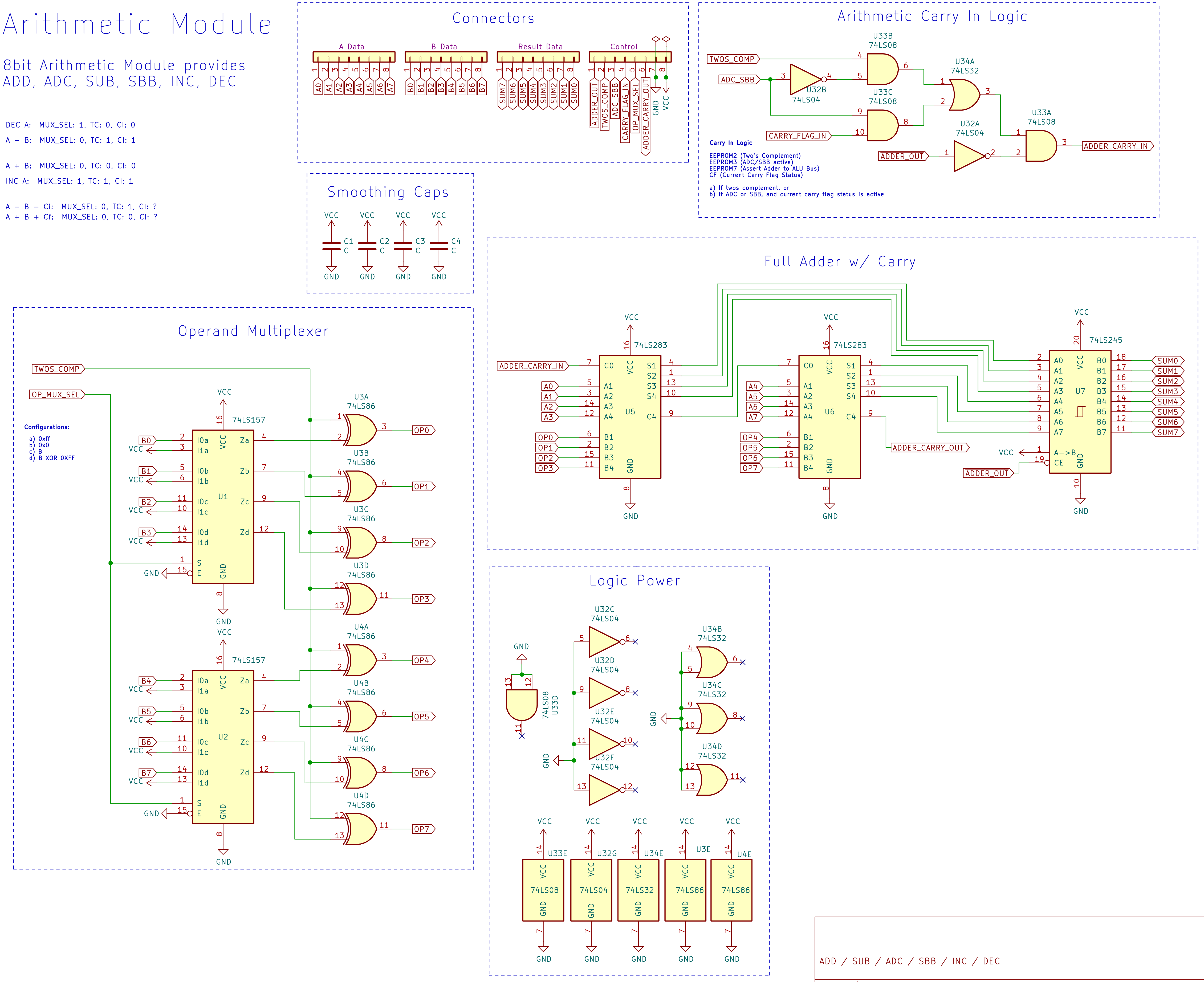
DEC A:  MUX_SEL: 1, TC: 0, CI: 0

A – B:  MUX_SEL: 0, TC: 1, CI: 1

A + B:  MUX_SEL: 0, TC: 0, CI: 0

INC A:  MUX_SEL: 1, TC: 1, CI: 1

A – B – Ci:  MUX_SEL: 0, TC: 1, CI: ?
A + B + Cf:  MUX_SEL: 0, TC: 0, CI: ?



Connectors
A Data
A0 A1 A2 A3 A4 A5 A6 A7
B Data
B0 B1 B2 B3 B4 B5 B6 B7
Result Data
SUM7 SUM6 SUM5 SUM4 SUM3 SUM2 SUM1 SUM0
Control
ADDER_OUT TWOS_COMP ADC_SBB CARRY_FLAG_IN OP_MUX_SEL ADDER_CARRY_OUT
GND VCC
Arithmetic Carry In Logic
TWOS_COMP
ADC_SBB
CARRY_FLAG_IN
ADDER_OUT
ADDER_CARRY_IN
U33B 74LS08
U32B 74LS04
U33C 74LS08
U34A 74LS32
U32A 74LS04
U33A 74LS08
Smoothing Caps
C1 C C2 C C3 C C4 C
VCC GND
Operand Multiplexer
TWOS_COMP
OP_MUX_SEL
U1 74LS157
U2 74LS157
I0a I1a I0b I1b I0c I1c I0d I1d S E Za Zb Zc Zd
B0 B1 B2 B3 B4 B5 B6 B7
U3A 74LS86 U3B 74LS86 U3C 74LS86 U3D 74LS86
U4A 74LS86 U4B 74LS86 U4C 74LS86 U4D 74LS86
OP0 OP1 OP2 OP3 OP4 OP5 OP6 OP7
Full Adder w/ Carry
ADDER_CARRY_IN
U5 74LS283
U6 74LS283
C0 A1 A2 A3 A4 B1 B2 B3 B4 S1 S2 S3 S4 C4
A0 A1 A2 A3 OP0 OP1 OP2 OP3
A4 A5 A6 A7 OP4 OP5 OP6 OP7
ADDER_CARRY_OUT
U7 74LS245
A0 A1 A2 A3 A4 A5 A6 A7 B0 B1 B2 B3 B4 B5 B6 B7
A->B CE
ADDER_OUT
SUM0 SUM1 SUM2 SUM3 SUM4 SUM5 SUM6 SUM7
Logic Power
U32C 74LS04 U32D 74LS04 U32E 74LS04 U32F 74LS04
U34B 74LS32 U34C 74LS32 U34D 74LS32
U33D 74LS08
U33E 74LS08 U32G 74LS04 U34E 74LS32 U3E 74LS86 U4E 74LS86
VCC GND

Carry In Logic

EEPROM2 (Two's Complement)
EEPROM3 (ADC/SBB active)
EEPROM7 (Assert Adder to ALU Bus)
CF (Current Carry Flag Status)

a) If twos complement, or
b) if ADC or SBB, and current carry flag status is active

Configurations:

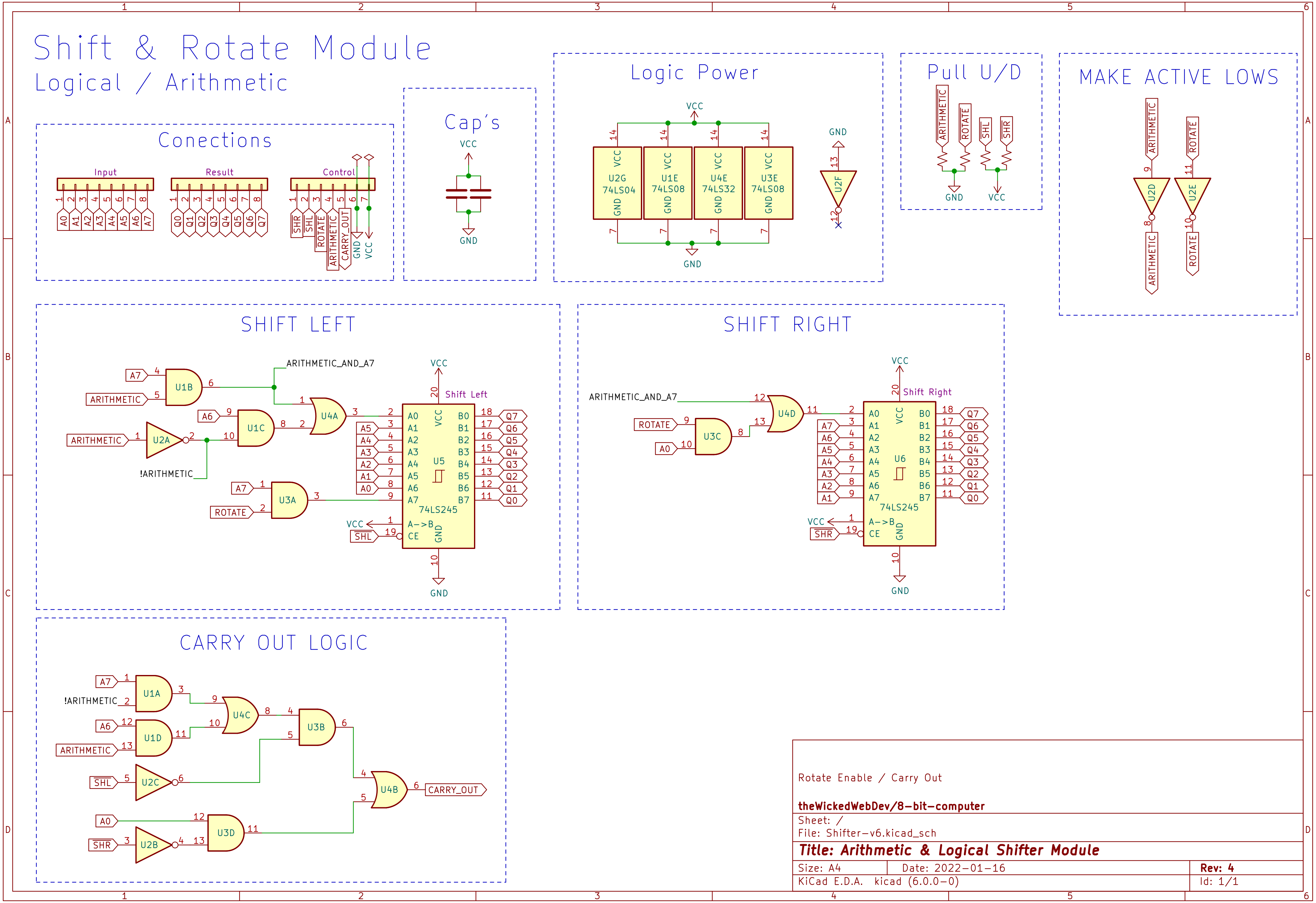
a) 0xff
b) 0x0
c) B
d) B XOR 0XFF

ADD / SUB / ADC / SBB / INC / DEC

Sheet: /
File: Arithmetic.kicad_sch

**Title: Arithmetic Module**

| | | |
|---|---|---|
| Size: User | Date: | Rev: 3 |
| KiCad E.D.A.  kicad (6.0.0-0) | | Id: 1/1 |



Rotate Enable / Carry Out

**theWickedWebDev/8-bit-computer**

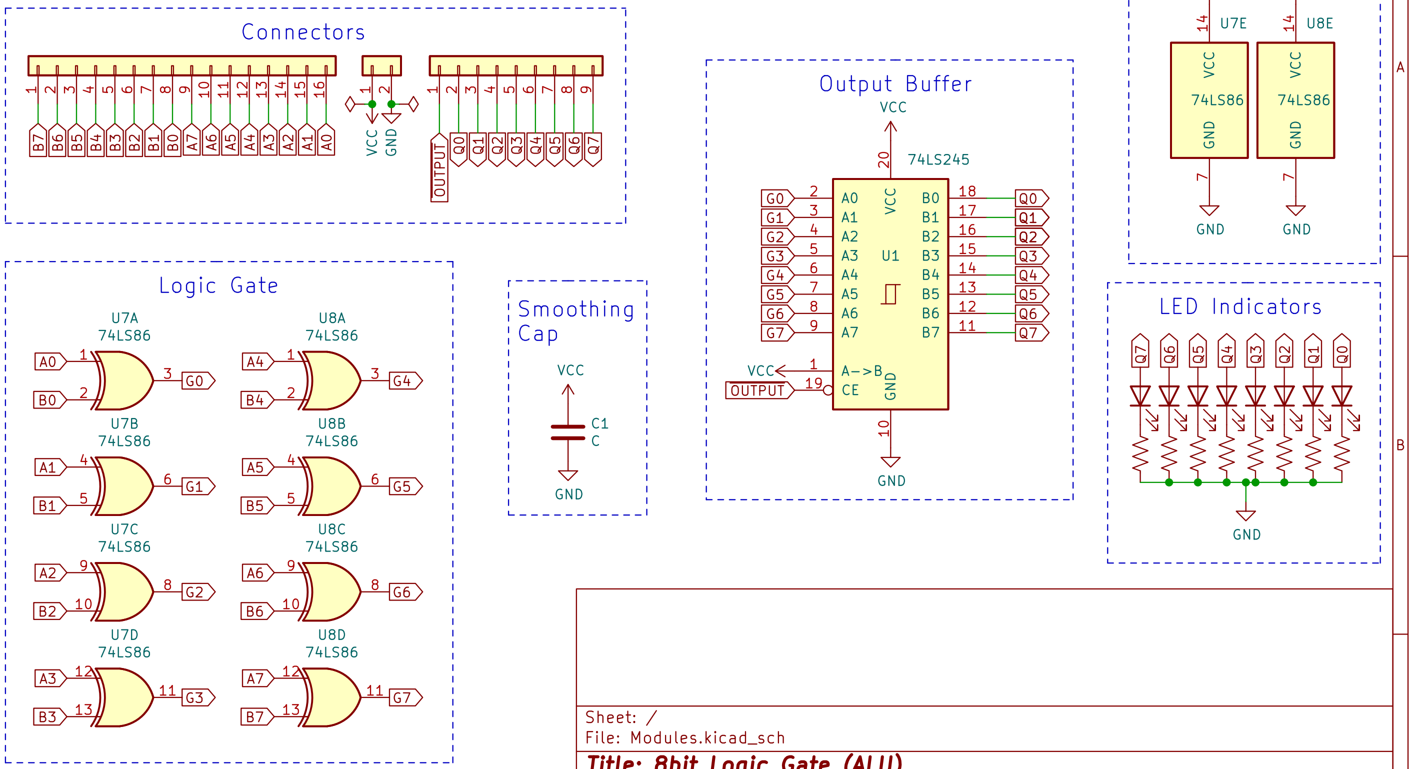
Sheet: /

File: Shifter-v6.kicad_sch

***Title: Arithmetic & Logical Shifter Module***

| | | |
|---|---|---|
| Size: A4 | Date: 2022-01-16 | **Rev: 4** |
| KiCad E.D.A. kicad (6.0.0-0) | | Id: 1/1 |

# 8bit Logic Gate

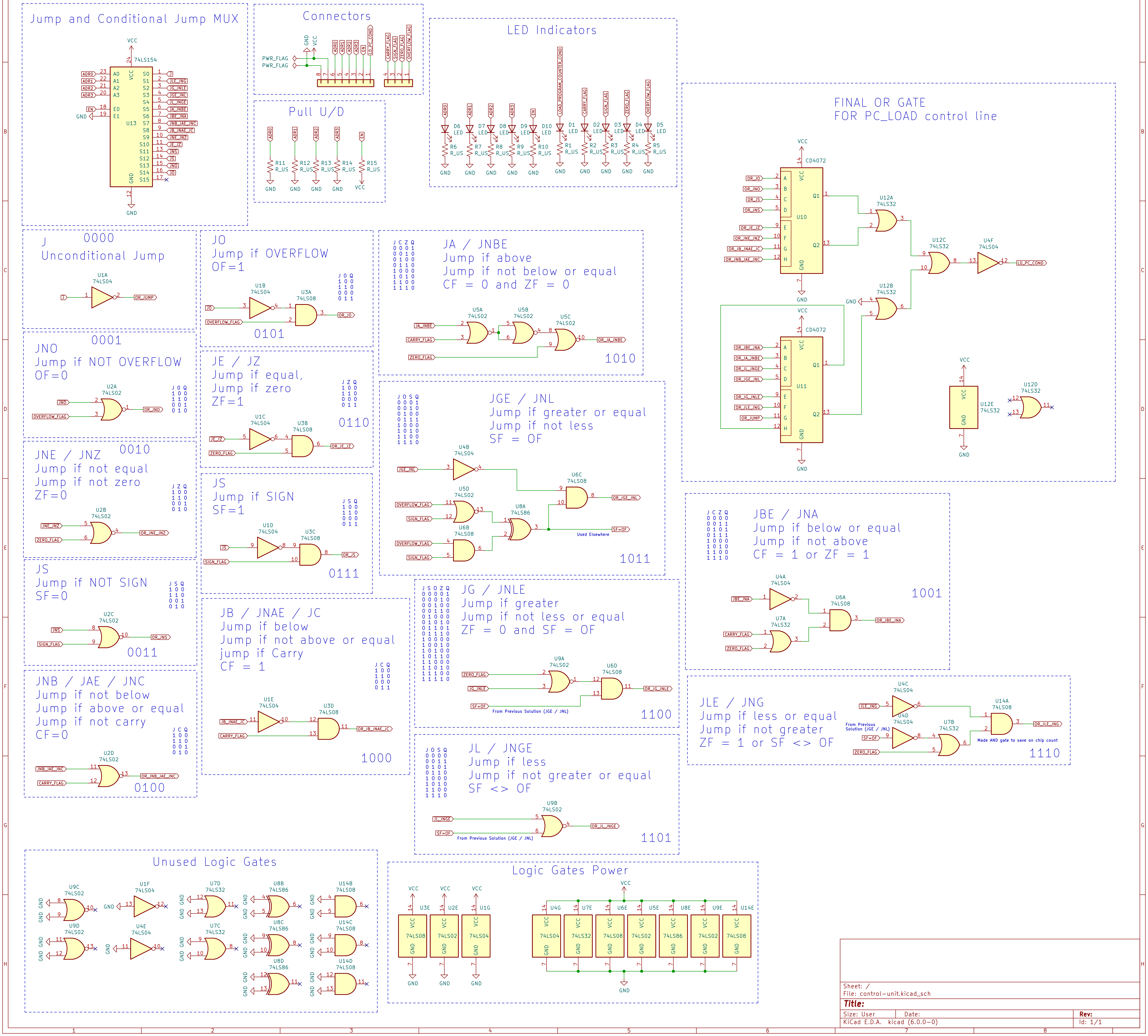


Sheet: /
File: Modules.kicad_sch

**Title: 8bit Logic Gate (ALU)**

| | | |
|---|---|---|
| Size: A5 | Date: | **Rev: 3** |
| KiCad E.D.A.  kicad (6.0.0-0) | | Id: 1/1 |

# ALU Conditional Jump Logic



Sheet: /
File: control-unit.kicad_sch

**Title:**

Size: User Date: Rev:
KiCad E.D.A. kicad (6.0.0-0) Id: 1/1

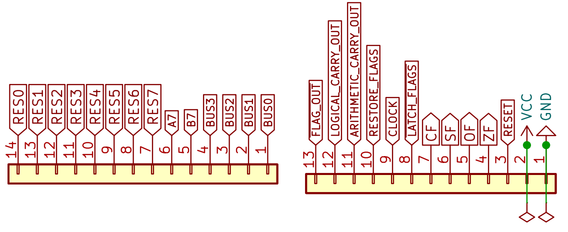
# FLAGS REGISTER

LATCH_FLAGS – A LOW signal will store the data asserted from the multiplexer into the Flags Register (FR)

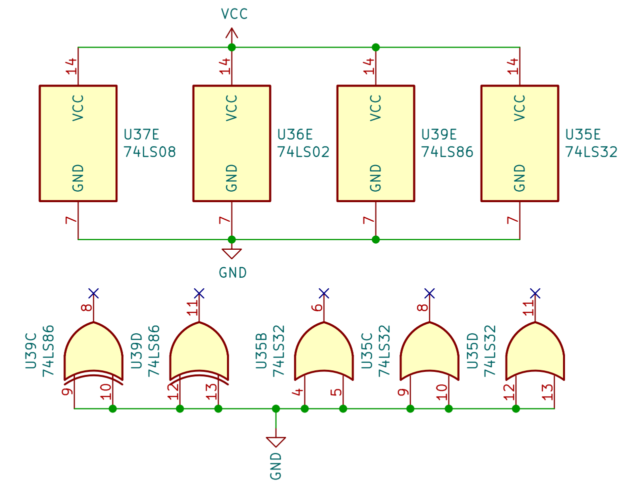
- RESTORE: LOW, uses signals from ALU
- RESTORE: HIGH, uses signal asserted on data bus

FLAG_OUT – Assers the current flags statuses onto the Data bus, typically used to push it onto the stack to handle an ISR

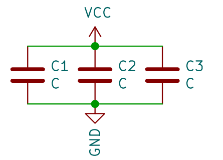
## Connections



## Logic Gate Power

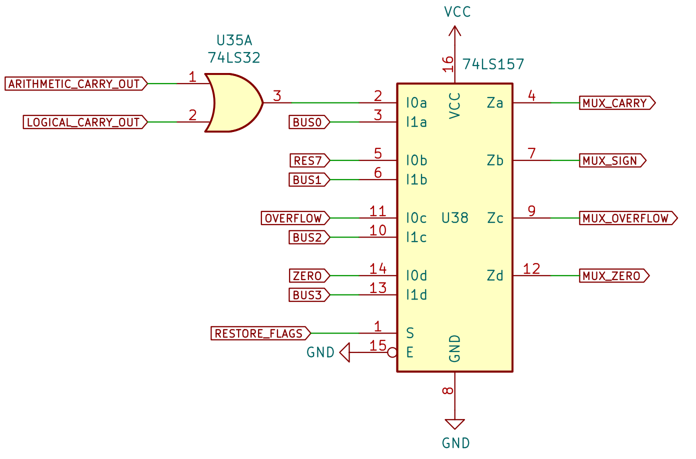


## Smoothing Caps

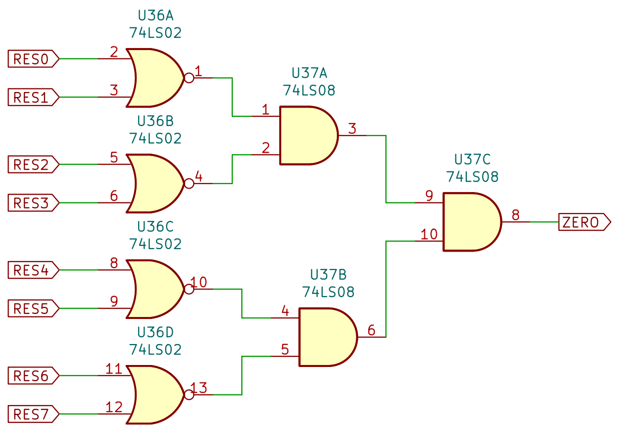


## Source Multiplexer

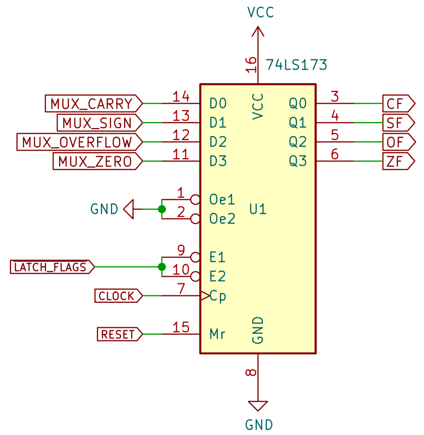
Flags come directly from ALU, or, from the flag/data bus to restore flags from the stack or another location



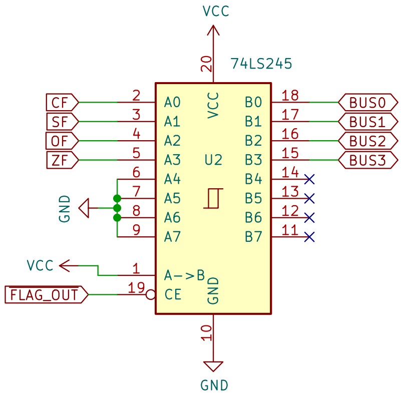
## Zero



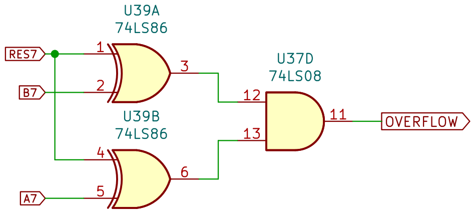
## REGISTER



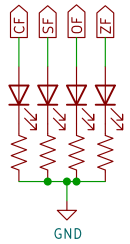
## Bus Connection



## OVERFLOW



## Leds



For storing and asserting current flag statuses from ALU
**theWickedWebDev/8-Bit-Computer**
Sheet: /
File: Flags Register.kicad_sch

**Title: Flags Register**

Size: User Date: 2022-01-03 **Rev: 3**
KiCad E.D.A. kicad (6.0.0-0) Id: 1/1

# Memory

The module provides the CPU with 2 megabytes of volatile memory and 32k of persistent EEPROM memory.
Note: The lower 32k of address space is not available to RAM, it is unreachable.

## EEPROM (32k) - $0000:$7fff

This is the memory that is used on startup to start running the program.

### Reserved Address Space

- $0:0: Initial program code instruction called after reset/boot

Interrupt Vector Table (IVT)

- $0:f0: IRQ0
- $0:f1: IRQ1
- $0:f2: IRQ2
- $0:f3: IRQ3
- $0:f4: IRQ4
- $0:f5: IRQ5
- $0:f6: IRQ6
- $0:f7: IRQ7
- $0:ff: Interrupt Service Routine (ISR)
- $0:fe: Reset

# RAM (2Mb) - $8000:$20000

## Segments / Paged

Each segment register is 5-bits wide and allows the user to access the entire range of space in memory. 16-bits from the address bus and the additional 3-bits (+2 bits for decoding chip enables) to make a 19-bit address.
Each segment can latch data from the lower 5-bits of the data bus as well as assert its contents out.

### Code Segment (CS)

This is active when fetching the next instruction. It is only changed by a far `jmp` instruction and cannot be manipulated by `mov`

### Data Segment (DS)

This is active during any reads or writes to Memory (excluding program code). This can be set using `mov ds, 0xf`

### Stack Segment (SS)

This is active when executing a `PUSH` or `POP` instruction to and from the stack. This can be set using `mov ss, 0xf`

# MEMORY MODULE w/segments

## 32K ROM | 2M RAM

## ROM $0000 – $7FFF
## RAM $8000 – $200000

Contents of this module include the following:

- 16-bit Memory Address Register (MAR)
- Memory Segments: Data, Code, Stack segments (DS, CS, SS)
- EEPROM & RAM

ROM address space is 15bits wide
RAM address space is 19bits wide

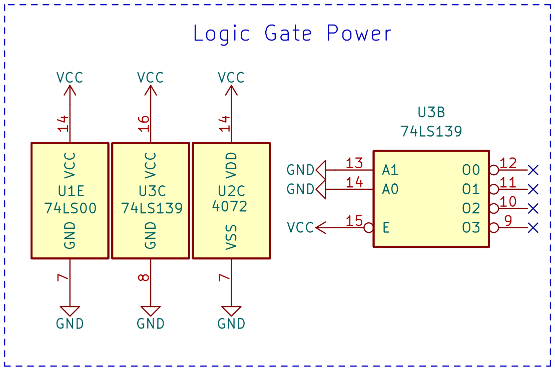
Since there is only 16bits in the MAR for addressing, the following logic explains how chip are selected:

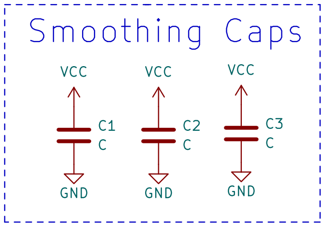
- If any segment is HIGH, or A15 is HIGH then EEPROM is disabled
- Otherwise, EEPROM is active and uses READ only
- The SEGMENT[0..2]'s are used to fill address space 16, 17, & 18
- SEGMENT[3,4] is decoded into 4 values and is used to target different RAM chips

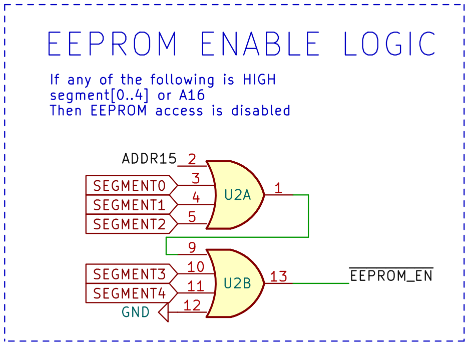
Segment registers are loaded from the shared 8bit bus and can also be asserted back onto the BUS. A reason for asserting is so you can {push ds} onto the stack to store them during interrupts

Addressing Key:
S: SEGMENT[3,4]
s: SEGMENT[0..2]
a: MAR[0..15]

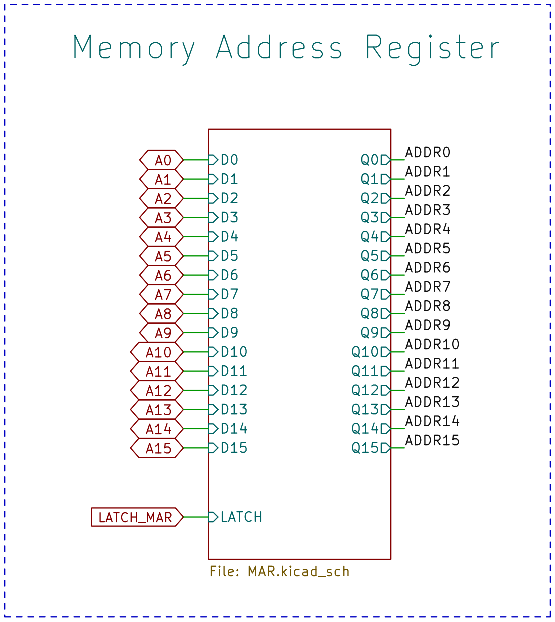
0b_SS_sss_aaaa_aaaa_aaaa_aaaa

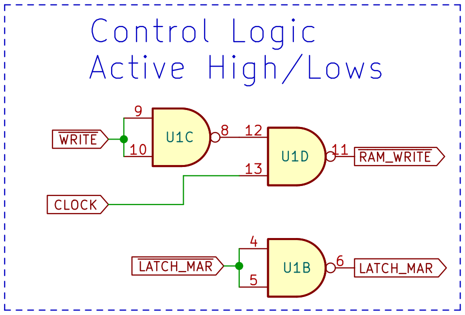


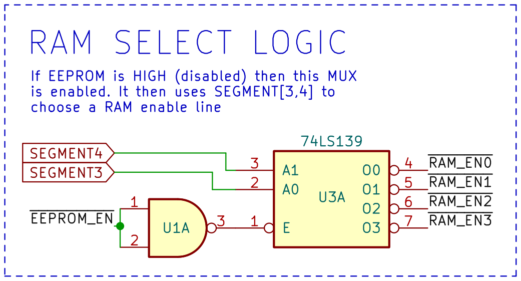




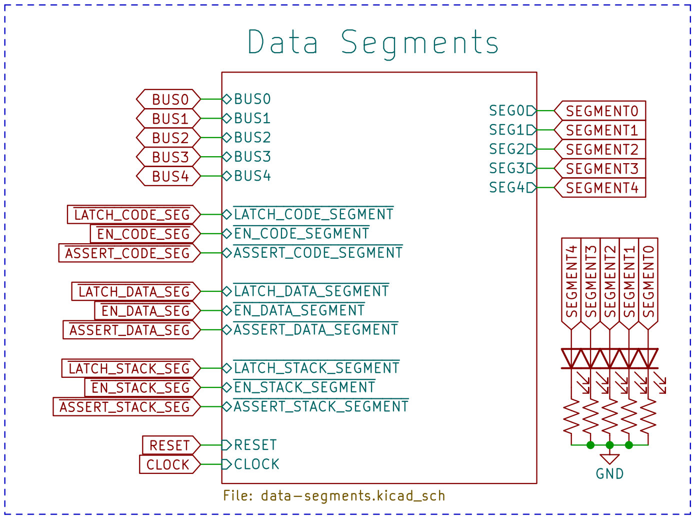
If any of the following is HIGH
segment[0..4] or A16
Then EEPROM access is disabled

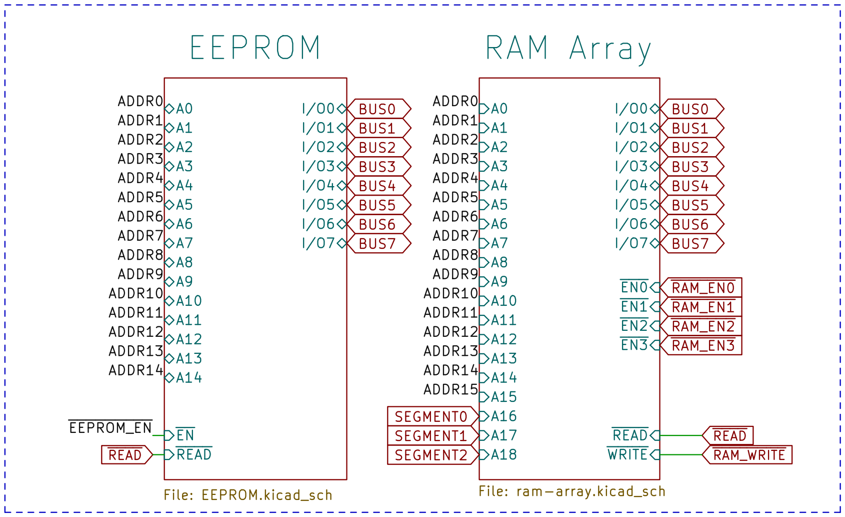


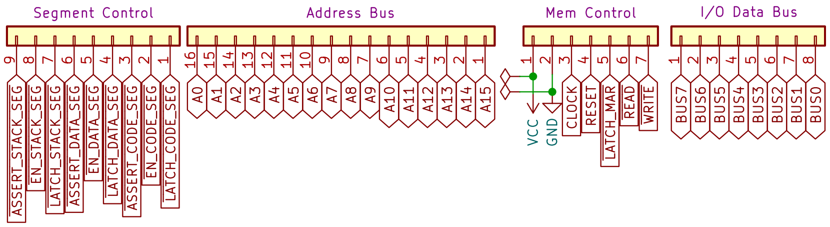




If EEPROM is HIGH (disabled) then this MUX
is enabled. It then uses SEGMENT[3,4] to
choose a RAM enable line







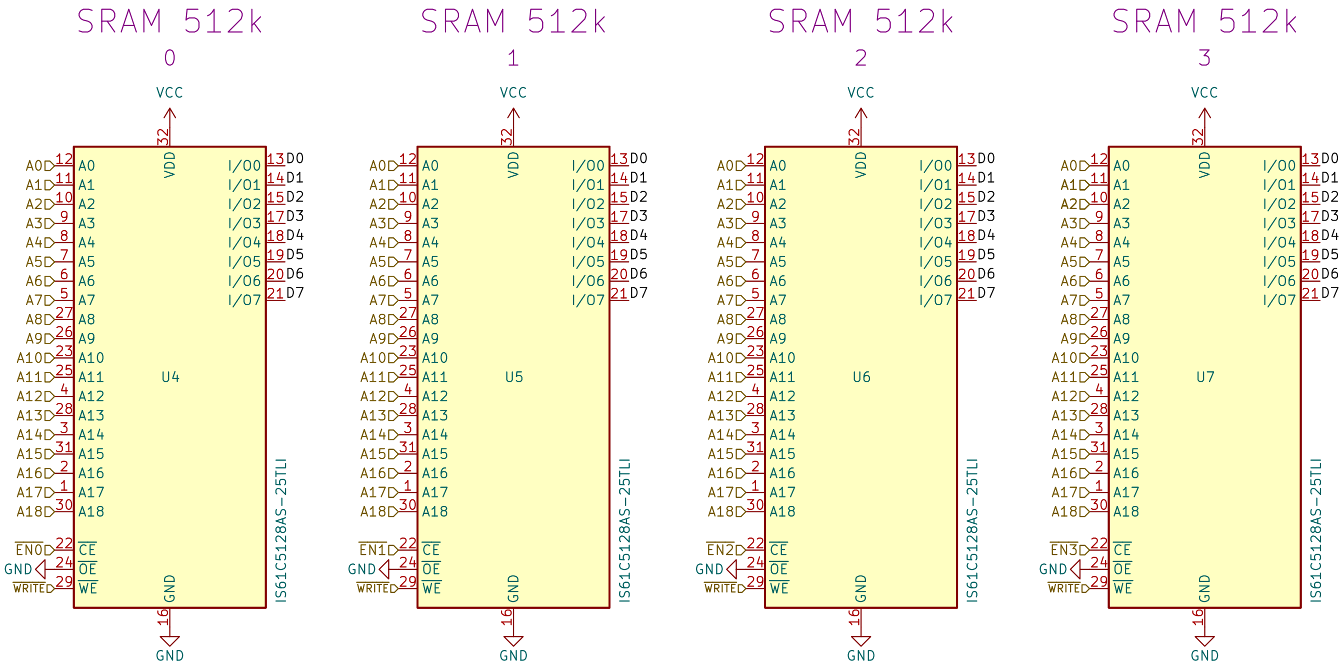
RAM $8000 – $20000
ROM $0000 – $7fff
32K ROM | 2M RAM
**Trilobyte – theWickedWebDev/8-bit-computer**

Sheet: /
File: memory.kicad_sch

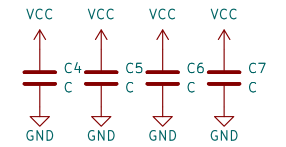
***Title: Memory Module***

| Size: User | Date: 2022-03-11 | **Rev: 1.6** |
|---|---|---|
| KiCad E.D.A. kicad (6.0.0-0) | | Id: 1/8 |

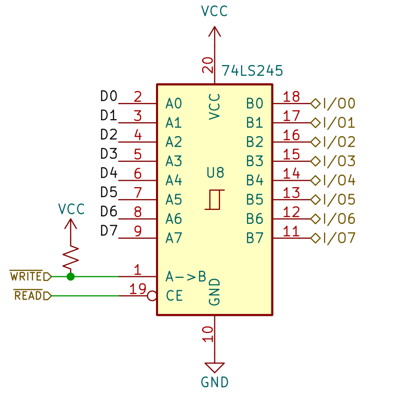
# RAM Array Module



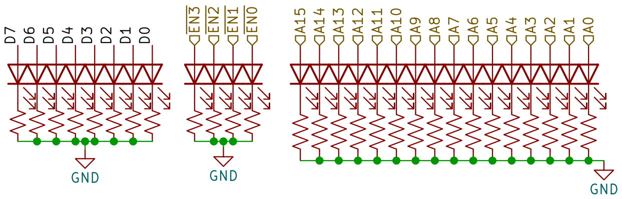
## Smoothing Caps



## Data BUS Tranceiver



## LED Indicators

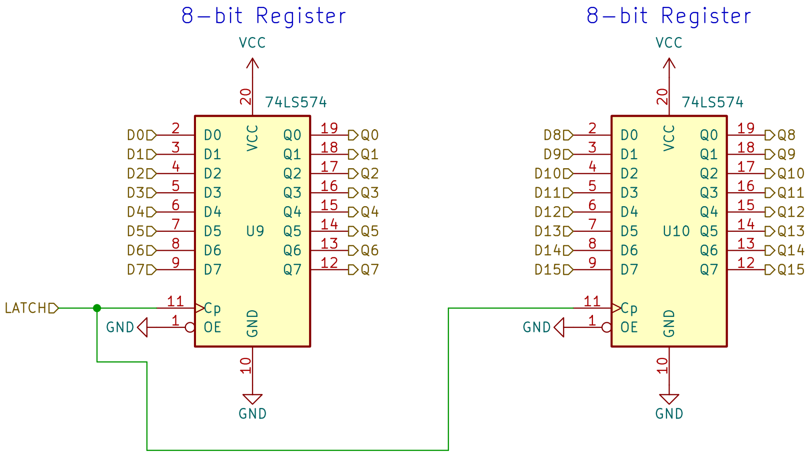


Sheet: /RAM Array/
File: ram-array.kicad_sch

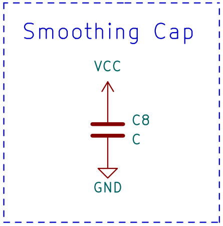
**Title:**

| Size: A4 | Date: | **Rev:** |
|---|---|---|
| KiCad E.D.A. kicad (6.0.0-0) | | Id: 2/8 |

# MEMORY ADDRESS REGISTER



ACTIVE LOW
Latches on the RISING EDGE of Clock

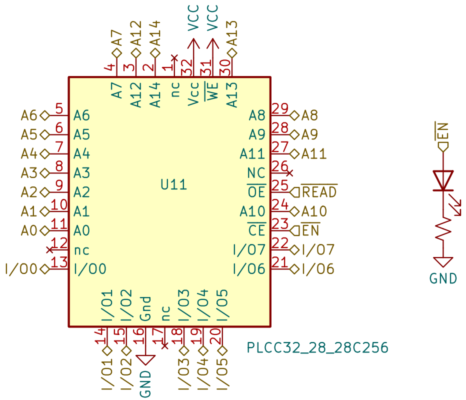

Smoothing Cap

Sheet: /Memory Address Register/
File: MAR.kicad_sch

**Title:**

| Size: A5 | Date: | **Rev:** |
|---|---|---|
| KiCad E.D.A. kicad (6.0.0-0) | | Id: 3/8 |

# EEPROM

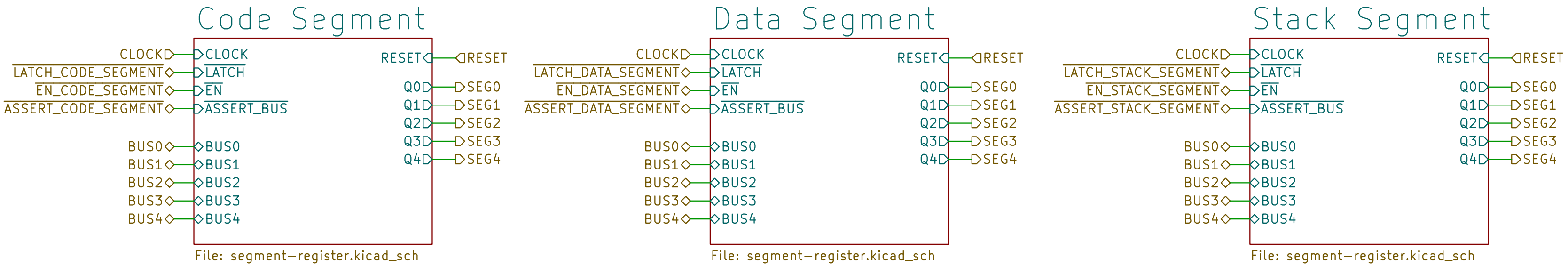


Sheet: /EEPROM/
File: EEPROM.kicad_sch

**Title:**

Size: User
Date:
**Rev:**
KiCad E.D.A.  kicad (6.0.0-0)
Id: 4/8

# Segments



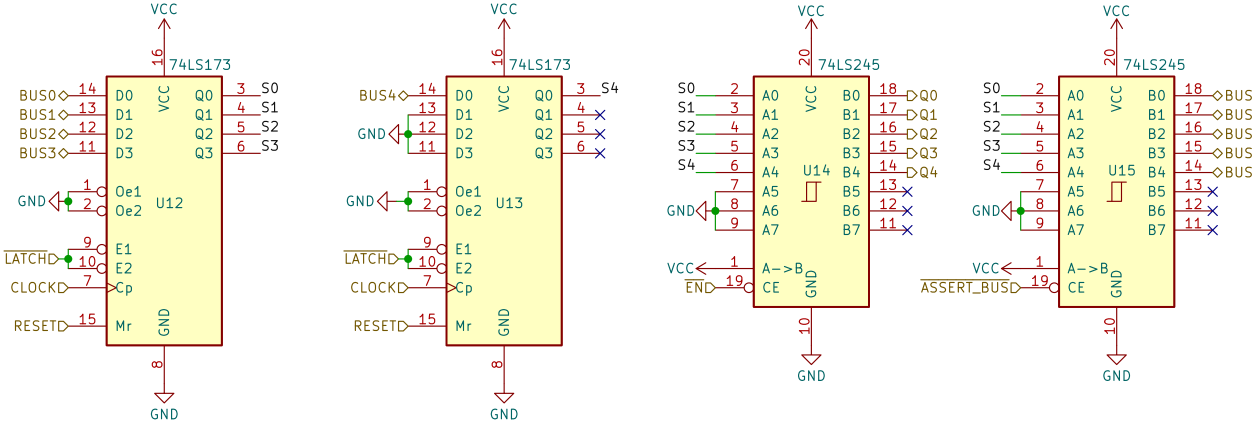
Sheet: /Data Segments/
File: data–segments.kicad_sch

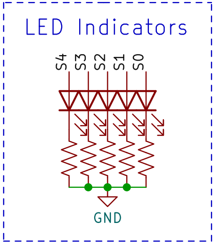
**Title:**

Size: User Date: **Rev:**

KiCad E.D.A. kicad (6.0.0–0) Id: 5/8

# Segment Registers: Data, Code & Stack





Sheet: /Data Segments/Data Segment/
File: segment-register.kicad_sch

**Title:**

Size: A5 Date: **Rev:**

KiCad E.D.A. kicad (6.0.0-0) Id: 6/8

# Internal Clock

# Interrupts

# Timers

# 16x2 Character LCD

# UART / Serial I/O

# VGA

# MIDI / Audio

# ADC - Analog to Digital Converter

# DAC - Digital to Analog Converter

# Software

# Instruction Set

## Opcode Table

## Address Modes

## Registers

## Instruction List

### Standard Instructions

### Jump Instructions

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | **NOP** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 | MOV A, C | MOV A, D | MOV A, (CD) | MOV A, imm | MOV A, mem | MOV F, AD | MOV F, AC | MOV F, CD | MOV F, imm$^{16}$ | MOV DS, imm | MOV DS, mem | MOV DS, A |  |  |  |  |
| 2 | MOV C, A | MOV C, D | MOV C, (AD) | MOV C, imm | MOV C, mem | MOV F, (AD) | MOV F, (AC) | MOV F, (CD) | MOV F, mem | MOV SS, imm | MOV SS, mem | MOV SS, A |  |  |  |  |
| 3 | MOV D A | MOV D, C | MOV D, (AC) | MOV D, imm | MOV D, mem | MOV SP, imm16 | MOV SP, F | MOV FLAGS, A | MOV FLAGS, imm | MOV FLAGS, (F) | MOV FLAGS, mem |  |  |  |  |  |
| 4 | PUSH A | PUSH C | PUSH D | PUSH F | PUSH FLAGS | PUSH DS | PUSH SS | PUSH imm | PUSH imm$^{16}$ |  |  |  |  |  |  |  |
| 5 | POP A | POP C | POP D | POP F | POP FLAGS | POP DS | POP SS |  |  |  |  |  |  |  |  |  |
| 6 | ADD A, C | ADD A, D | ADD A, imm | ADC A, C | ADC A, D | ADC A, imm | NOT A, A | SHL A | SHR A | CMP A, C | TEST A, C |  |  |  |  |  |
| 7 | SUB A, C | SUB A, D | SUB A, imm | SBB A, C | SBB A, D | SBB A, imm | NOT A, C | ASL A | ASR A | CMP A, D | TEST A, D |  |  |  |  |  |
| 8 | AND A, C | AND A, D | AND A, imm | NAND A, C | NAND A, D | NAND A, imm | NOT A, D | ROL A | ROR A | CMP A, imm | TEST A, imm |  |  |  |  |  |
| 9 | OR A, C | OR A, D | OR A, imm | NOR A, C | NOR A, D | NOR A, imm | NOT A, mem | INC A | DEC A | CMP A, mem | TEST A, mem |  |  |  |  |  |
| A | XOR A, C | XOR A, D | XOR A, imm | XNOR A, C | XNOR A, D | XNOR A, imm |  |  |  |  |  |  |  |  |  |  |
| B | JMP F | JMP imm16 | JMP (F) | JMP D:F | JMP D:(F) | JMP imm4:imm16 | JMP imm4:mem | CALL F | CALL (F) | CALL D:F | CALL D:(F) | CALL imm4: imm16 | CALL imm4:mem | RET |  |  |
| C | JNO F | JNE/JNZ F | JS F | JNB/JAE/JNC F | JO F | JE/JZ F | JS F | JB/JNAE/JC F | JA/JNBE F | JGE/JNL F | JG/JNLE F | JL/JNGE F | JBE/JNA F | JLE/JNG F |  |  |
| D | JNO (F) | JNE/JNZ, (F) | JS (F) | JNB/JAE/JNC (F) | JO (F) | JE/JZ (F) | JS (F) | JB/JNAE/JC (F) | JA/JNBE (F) | JGE/JNL (F) | JG/JNLE (F) | JL/JNGE (F) | JBE/JNA (F) | JLE/JNG (F) |  | CLI imm |
| E | IN A, imm | IN A, C | IN A, D | OUT imm, imm | OUT imm, A | OUT imm, C | OUT imm, D |  |  |  |  |  |  |  |  | RTI |
| F | CIN A, imm | CIN A, C | CIN A, D | COUT imm, imm | COUT imm, A | COUT imm, C | COUT imm, D |  |  |  |  |  |  | **HLT** | **RESET** | IRQ |

# Microinstructions