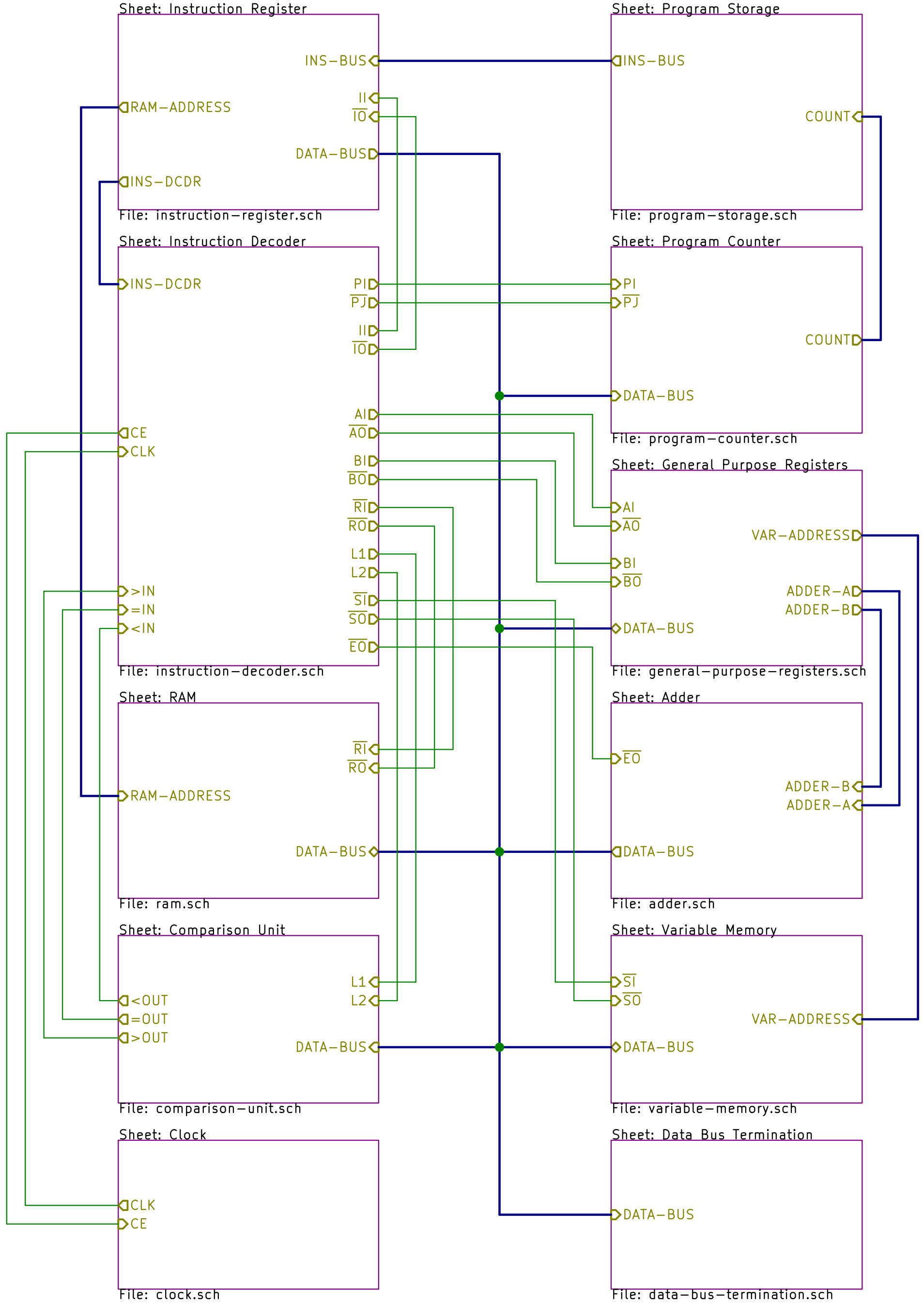


How To Build An 8-Bit CPU

This document is the full schematic (excluding a power supply) for a fully functional, Turing complete,
8-bit CPU (at least I really hope it is).
By that, I mean that it is capable of executing any algorithm, or computing anything
'computable' – an incredibly vague definition that you'd be much better served by Turing's original paper
on than by anything I could say here.
This is meant to be understandable to anyone with a basic understanding of digital logic
and nothing more, because that's really all it is. Most of the 'circuit diagrams' are just the
connections between integrated circuits, all of which are as well labeled as clearly as I could,
so hopefully they should be fairly simple to understand, however don't hesitate to ask any questions.

The first thing to explain is the fundamental process by which a 'stored program computer' – ie a computer that stores the program it is to execute within its own memory – takes that stored program data and turns it into actual things to do. This is known (as anyone who did computer science GCSE should know) as the 'fetch-decode-execute' cycle. It's known as that because computer scientists and computer engineers aren't very imaginative and it literally consists of the computer 'fetching' an instruction, at a location specified by the 'program counter' (a literal counter that counts through the instructions in the program, occasionally skipping some or going back to an earlier one if we tell it to), 'decoding' it (transforming it from an 8-bit binary 'word' into a set of control signals to send to all the other modules to make them do what we want) and then 'executing' it (actually making those modules do the things).

The modules the instructions can make use of are all displayed in the boxes to the left. The main ones of note are the RAM / variable storage, which are both used for storing data (variables) produced by the program, the 'registers' which are another form of extremely quick access storage which only store a single byte of data (8-bits, up to the value 255) and are manipulated by almost every instruction, the 'adder' which adds the values of the two registers and the comparison unit, which can compare the size of the numbers stored in the registers. Between these modules, the only limits to what this computer can do are storage space and time.

When I was designing the CPU I had the following basic instructions in mind, to guide the control signals I made available and to ensure that the end result would actually be able to do some interesting stuff.
Try and think about how each one would be implemented while we go through the modules – I'll go through them at the end if we have time.

NOP – No operation, ie do nothing
HLT – Halt (stop) the CPU at the end of program execution
ADR – Add the A and B registers
MVA – Put the argument to the instruction into the A register
MVB – Put the argument to the instruction into the B register
RSA – Store the contents of the A register in RAM at the address given by the instruction's argument
RSB – Store the contents of the B register in RAM at the address given by the instruction's argument
RLA – Load the contents of the A register from RAM at the address given by the instruction's argument
RLB – Load the contents of the B register from RAM at the address given by the instruction's argument
STR – Store the contents of the A register in variable storage at the address given by the contents of the B register
LDR – Load the contents of the A register from variable storage at the address given by the contents of the B register
B – Unconditionally jump to the intruction at the address given by the instructions argument
BEQ – Jump to the intruction at the address given by the instructions argument if A = B
BLT – Jump to the intruction at the address given by the instructions argument if A < B
BGT – Jump to the intruction at the address given by the instructions argument if A > B

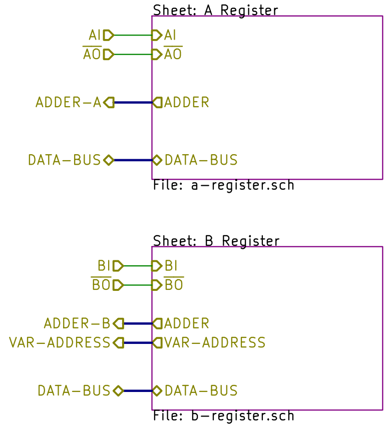
Note that instructions are stored in the form
XXXX YYYY
where XXXX is the 'operand' – ie the 4-bits associated which each of the above commands and
YYYY is the argument to the command, if applicable.

Author: Sebastian Gaume
This page shows an overview of the connections between all the functional blocks of the CPU.

Sheet: /
File: 8-bit-computer.sch

Title: 8 Bit CPU Overview

Size: A4	Date: 2019-11-11	**Rev: 1**
KiCad E.D.A. kicad 5.1.4		Id: 1/14



The two general purpose registers each store 8 bits of information, using 8 D-type flip flops each. A flip flop is a logic circuit that flips between two states based on its inputs, and a d-type flip flop is a flip flop that has a data input line and a clock input. The flip flop stores and outputs whatever logic level the data input had at the last clock pulse. Here the clock pulses are the control signals AI and BI, which cause the A or B register to store the data bus as it is.

Each register also had a line buffer stage. This exists because in the way I've configured the flip flops they are both constantly outputting to the adder and variable storage, as this saves on control signals, however if their outputs were then connected to the data bus they would render it unusable. As such, the buffers only allow the output to be passed through to the data bus when the AO and BO control signals are asserted. They're effectively a set of 8 switches, which allow the data to pass through onto the data bus when the respective control signals are asserted.

Authot: Sebastian Gaume

A sheet that gives access to both of the general purpose registers in the CPU.

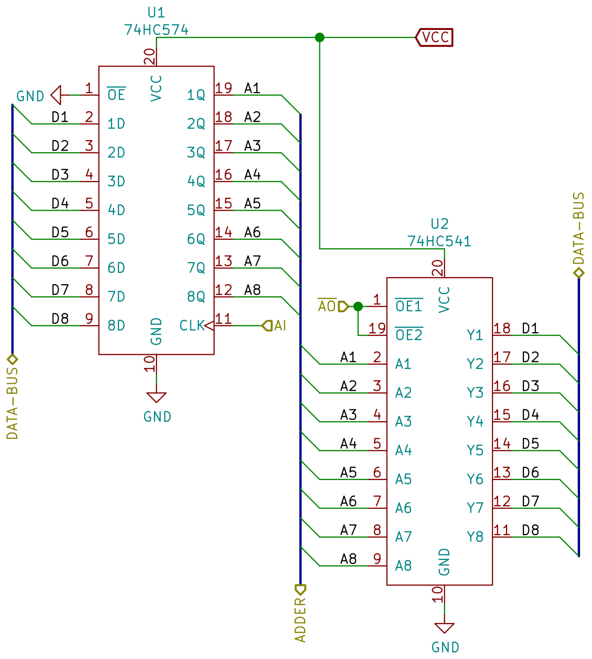
Sheet: /General Purpose Registers/
File: general-purpose-registers.sch

Title: General Purpose Registers

Size: A4 Date: 2019-11-08 **Rev: 1**

KiCad E.D.A. kicad 5.1.4 Id: 2/14

The 74HC547 is the set of 8 D-type flip flops, with its inputs connected to the data bus and its outputs to an internal bus which goes to the line buffer and the adder. Its output enable pin is tied such that it always outputs and the clock signal is hooked up to AI.



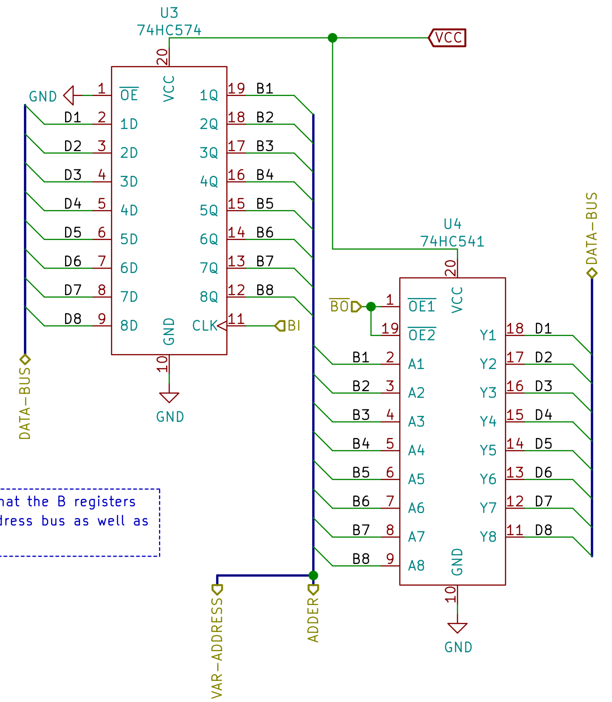
The 74HC541 is the line buffer, which has its inputs connected to the registers internal bus and its outputs connected to the data bus. Its output enable lines are connected to the control signal AO.

Author: Sebastian Gaume
The schematic for the 8-bit A register.

Sheet: /General Purpose Registers/A Register/
File: a-register.sch

Title: A Register

Size: A4	Date: 2019-11-08	**Rev: 1**
KiCad E.D.A. kicad 5.1.4		Id: 3/14



The only difference between the A and B registers is that the B registers internal bus is connected to the variable storage's address bus as well as to the adder.

Author: Sebastian Gaume

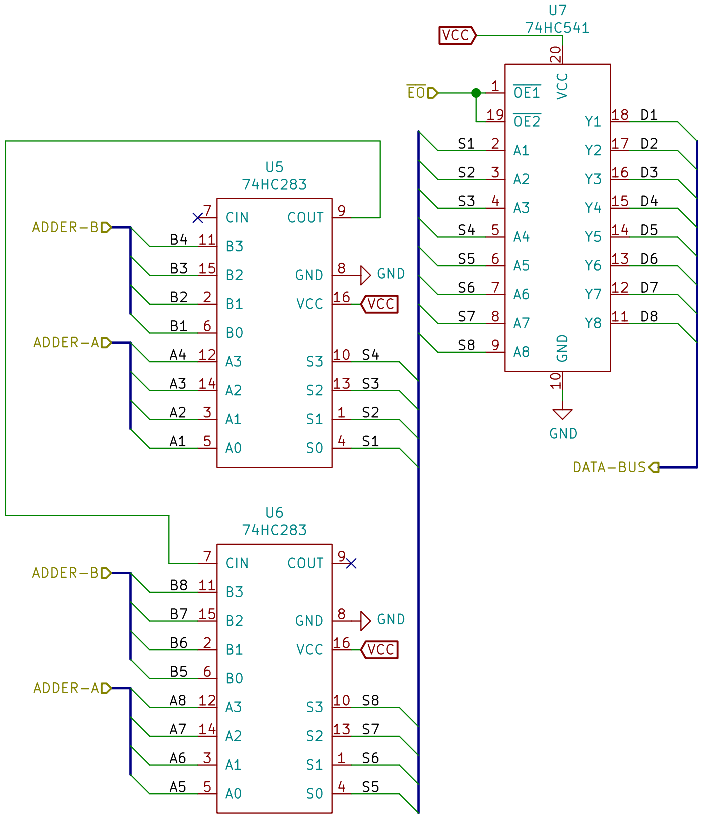
The schematic for the 8-bit B register.

Sheet: /General Purpose Registers/B Register/

File: b-register.sch

Title: B Register

Size: A4	Date: 2019-11-08	**Rev: 1**
KiCad E.D.A. kicad 5.1.4		Id: 4/14



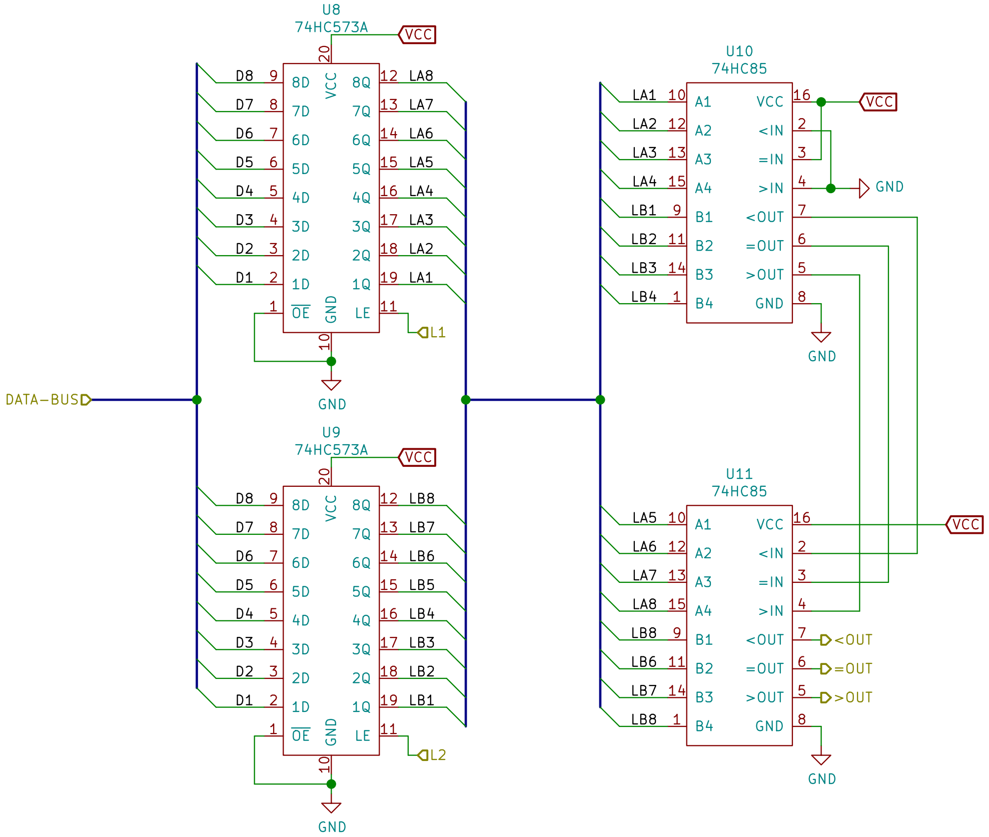
The adder gets constant input from the A and B registers, and constantly outputs its result to the line buffer chip (similarly to the registers), which only asserts to the data bus when the EO control signal is asserted. The actual adding is done by 2 74HC283 4−bit binary full adders, one of which outputs a carry bit to the other, allowing them to act as 1 8−bit adder.

Author: Sebastian Gaume
The schematic for the 8−bit adder.

Sheet: /Adder/
File: adder.sch

Title: Adder

Size: A4	Date: 2019−11−08	**Rev: 1**
KiCad E.D.A. kicad 5.1.4		Id: 5/14



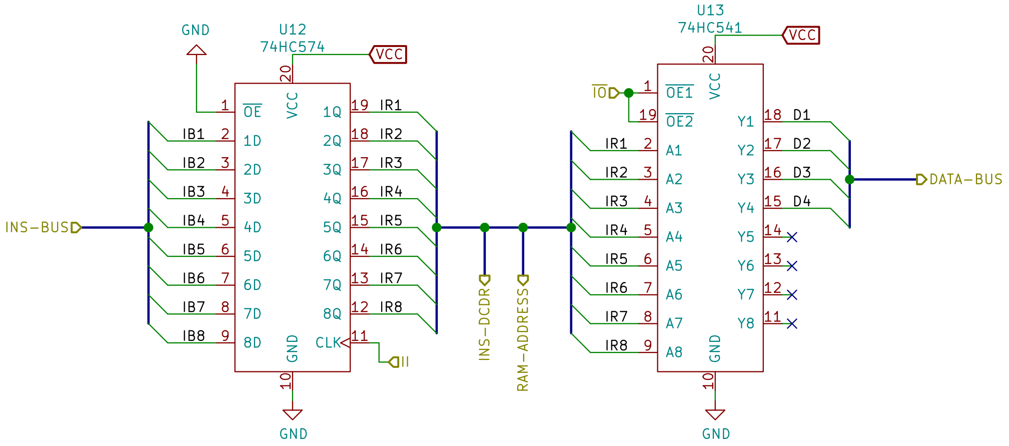
The comparison unit consists of two data latches (chips that can 'latch' the current value of a bus and output it) being fed into two 4-bit comparators which, similarly to the adders and counters elsewhere in the CPU have outputs that can carry between chips, allowing the two 74HC85s to act as a single 8-bit comparator. The two 74HC573A latches are used to store the two words to compare, each one storing the value of the data bus on the relevant control signal of either L1 or L2. The comparators output their result to the instruction decoder where it is used as part of the instruction lookup process.

Author: Sebastian Gaume
The schematic for the comparison unit of the 8-bit CPU, which can compare the magnitude of two 8-bit words.

Sheet: /Comparison Unit/
File: comparison-unit.sch

Title: Comparison Unit

Size: A4	Date: 2019-11-11	**Rev: 1**
KiCad E.D.A. kicad 5.1.4		Id: 6/14



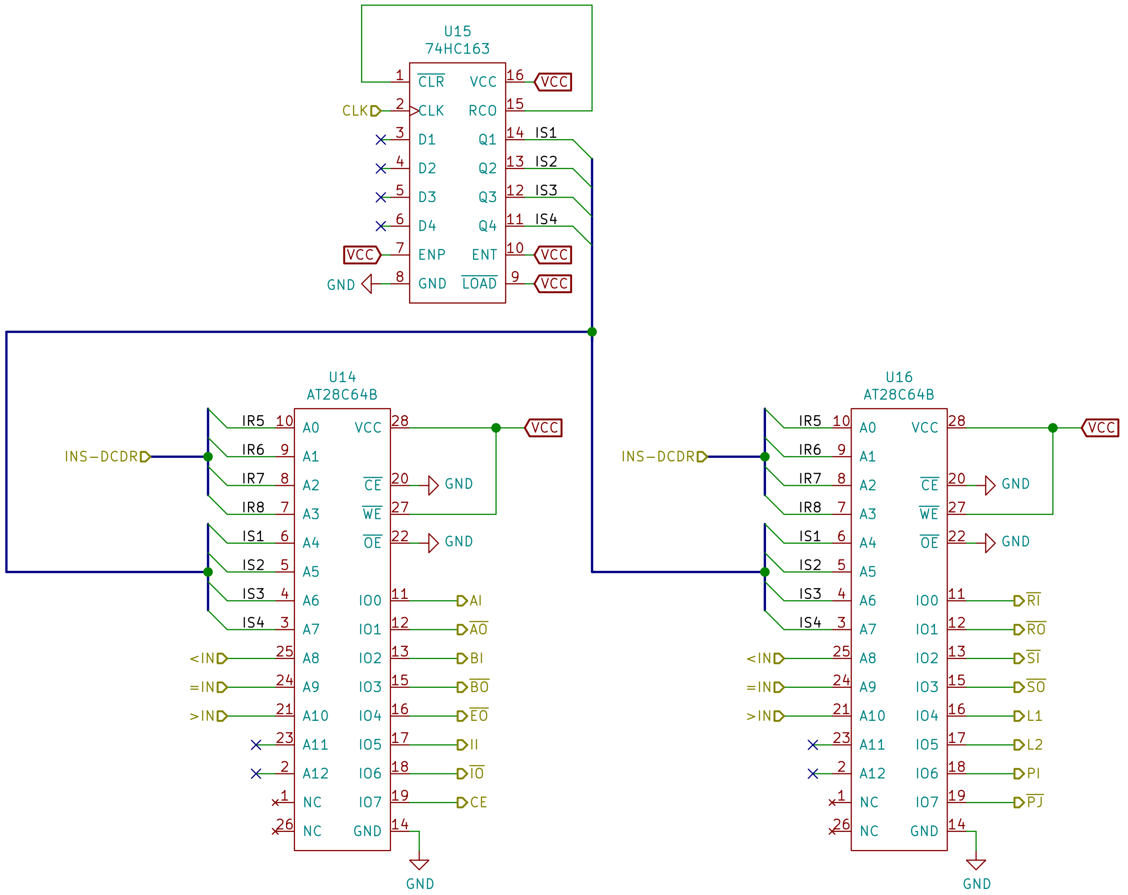
The instruction register is identical to the general purpose registers, except that it only outputs its 4 least significant bits to the data bus since the others are used for operands which shouldn't ever go on the data bus. It constantly outputs its 4 most significant bits to the instruction decoder, for it to decode the instructions from machine code to control signals so that they can be executed and it constantly outputs its 4 least significant bits to the RAM's address bus, as it is the only thing that assets those lines, so it saves control signals to not have an 'output address to RAM signal'.

Author: Sebastian Gaume
The schematic for the instruction register of the 8−bit CPU.

Sheet: /Instruction Register/
File: instruction−register.sch

Title: Instruction Register

Size: A4 Date: 2019−11−11 **Rev: 1**
KiCad E.D.A. kicad 5.1.4 Id: 7/14



The instruction decoder, as the name suggests, decodes instructions. It does so by looking up the instruction's 'op-code' (the 4-bit binary operand from the instruction register), and the current step (of which each instruction has at most 8) in an EEPROM. The EEPROMs use these two fields, along with the current output status of the comparison unit as an address, where the data at that address is the value of each control signal for that instruction, at that step, depending on the status of the comparison unit (which is useful for conditional jump instructions, ie jump if A = B).

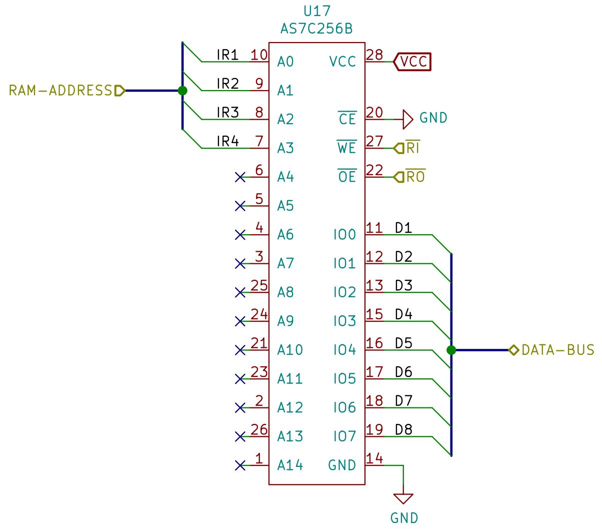
Here, the instruction decoded is implemented using a single 4-bit counter 74HC163 chip and two AT28C64B, whic are necessary since we have 16 control signals and each one can only set 8 for any given address (this isn't strictly true, and we could very much optimise this and probably save an EEPROM, but in the interest of keeping this nice and understandable, this is how I've done it here – ask me about it if you're interested). Both of the EEPROMS have exactly the same inputs to their address lines, but each one outputs to 8 different control signals. The counter is incremented by the system clock, which thus sets the speed at which the whole computer runs.

Author: Sebastian Gaume
The schematic for the EEPROMs that, along with a counter, decode the intstruction words into the correct set of steps of command signals.

Sheet: /Instruction Decoder/
File: instruction-decoder.sch

Title: Instruction Decoder

Size: A4 Date: 2019-11-12 **Rev: 1**
KiCad E.D.A. kicad 5.1.4 Id: 8/14



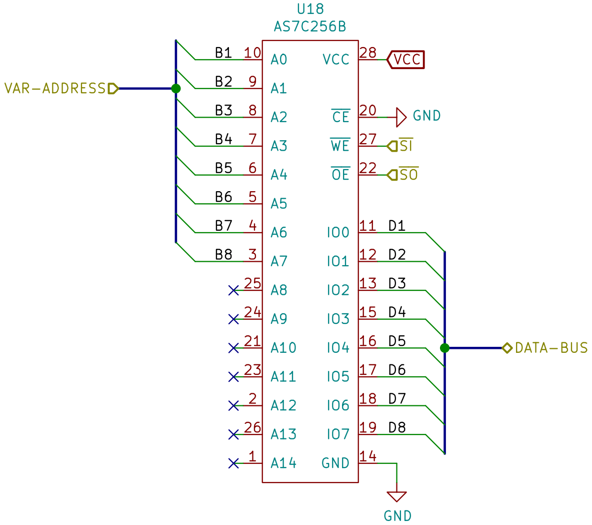
The RAM is a single AS7C256B chip, which while capable of storing up to 32kB of data, is only being used to store up to 16 bytes, as it is purely for temporary storage at addresses small enough to be arguments to instructions. There is another memory chip in the CPU which uses the B register for addresses and thus can store far more, however this is intended for very short term storage, as the other chip requires the address first be loaded into the B register first before it can be accessed. This chip simply has the address from the instruction register coming to its inputs and the data bus connected to its outputs. The control signal RO causes it to output the value at the current address to the data bus, while RI causes it to store the value of the data bus at the current address.

Author: Sebastian Gaume
The schematic for the main RAM of the 8−bit CPU.

Sheet: /RAM/
File: ram.sch

Title: RAM

Size: A4 | Date: 2019−11−11 | **Rev: 1**
KiCad E.D.A. kicad 5.1.4 | Id: 9/14



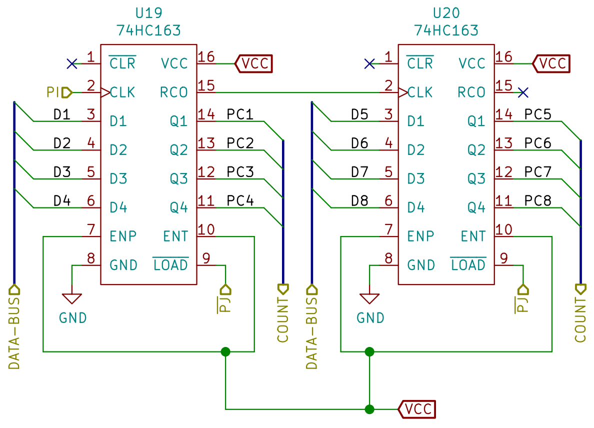
This 'variable storage' block is actually just an extension to the available RAM in the CPU, consisting of the same AS7C256B SRAM chip as the other RAM. The only difference is that this chip uses the B register for its addressing, allowing it to access 256 bytes instead of 16. The tradeoff is that it is more complex to access and so should be used for storing things to be used much later on. Please note that the way I've done memory storage throughout the CPU is very simplistic and not feasible in real CPUs. In fact, real 8−bit CPUs only have an 8−bit data bus, and have far more avaiable addressing lines, generally done by appending multiple 8−bit words onto each other before sending it as an address. While this massively expands the available memory, it is also more complicated and this system is only really meant as a demonstration.

Author: Sebastian Gaume
The schematic for the extra RAM chip, which allows access to much more memory for storing variables during program execution.

Sheet: /Variable Memory/
File: variable−memory.sch

Title: Variable Memory

Size: A4	Date: 2019−11−11	**Rev: 1**
KiCad E.D.A. kicad 5.1.4		Id: 10/14



The program counter stores the next address within the program storage to fetch an instruction from. Its count can either be incremented by 1 with the PI control signal, or set to the value of the data bus with the PJ control signal.

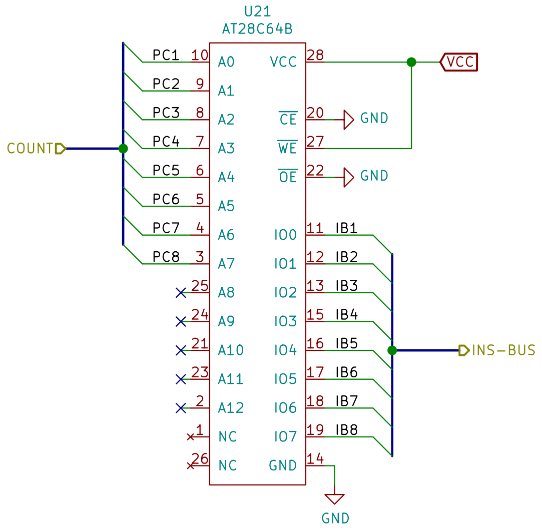
The program counter is implemented with 2 4-bit counters, which are synchronised through the use of a 'ripple carry' - ie once one reaches the maximum and resets, it causes the other to count one, which is how binary counting works (I'll probably demonstrate this more clearly while I'm talking through it). The counters used are 74HC163s, with their inputs connected to the data bus and PI/PJ control signals and their outputs connected to the 'count' bus, which goes to the address lines of program storage.

Author: Sebastian Gaume
The schematic for the program counter of the 8-bit CPU.

Sheet: /Program Counter/
File: program-counter.sch

Title: Program Counter

Size: A4 | Date: 2019-11-11 | **Rev: 1**
KiCad E.D.A. kicad 5.1.4 | Id: 11/14



The program storage stores all of the instructions that make up the program that the computer will exectute. They are stored in 8-bit binary words, where the 4 most significant bits (the top 4) store the 'operand', ie the instruction to perform and the 4 least significant bits store the 'argument', ie any data needed to carry out the instruction, for example an address to load.

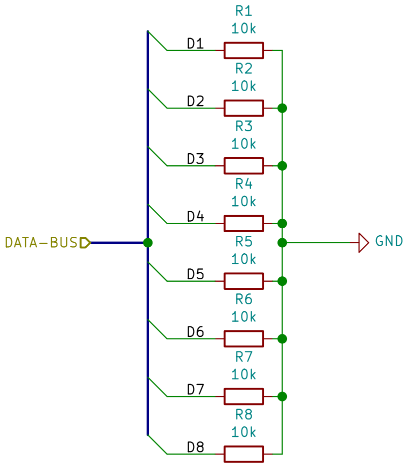
The implementation here is an EEPROM - an 'electronically erasable programmable read only memory'. The name may be entirely self-contradictory, but it's basically just a chip that retains data even when not powered. I've used an AT28C64B, which can store up to 8kB of data, however since this is an 8-bit computer and I'm only using an 8-bit address bus, it can only really store 256 bytes in this configuration. It will constantly output the value at the address given by the program counter, since the instruction register will decide when to load it, and so all its enable pins are tied high or low to facillitate constantly outputting. Its address lines go to the program counter, while its IO lines go to the instruction register.

Author: Sebastian Gaume
The schematic for the EEPROM that stores the program instructions.

Sheet: /Program Storage/
File: program-storage.sch

Title: Program Storage

Size: A4	Date: 2019-11-12	**Rev: 1**
KiCad E.D.A. kicad 5.1.4		Id: 12/14



This sheet only exists because the logic chips used don't like it when an input isn't specifically high or low, so this termination makes it so that if nothing is asserted to the data bus, a low value will be asserted.

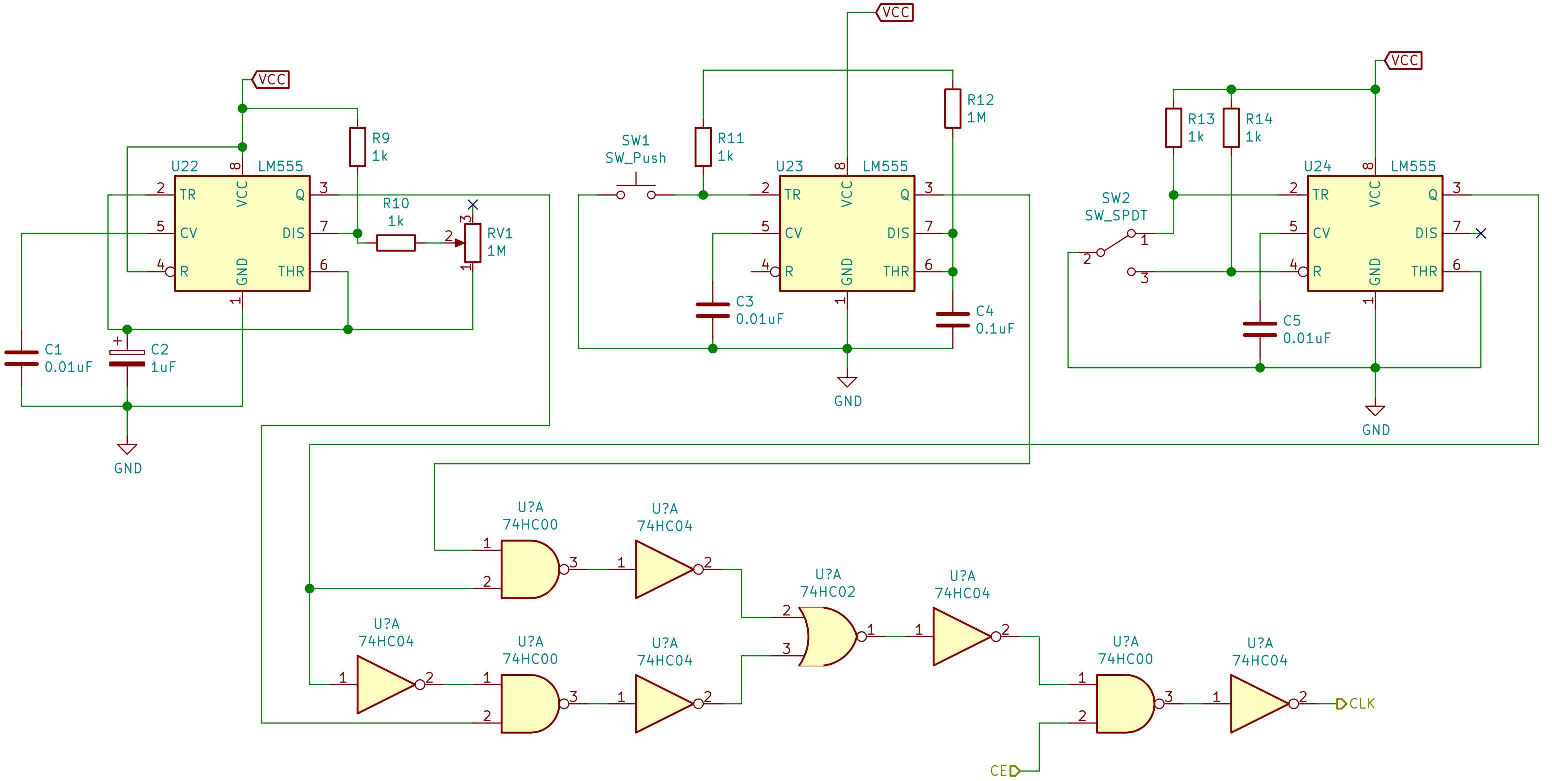
Author: Sebastian Gaume

This page just neatly terminates the data base with pull-down resistors.

Sheet: /Data Bus Termination/
File: data-bus-termination.sch

Title: Data Bus Termination

Size: A4	Date: 2019-11-12	**Rev: 1**
KiCad E.D.A. kicad 5.1.4		Id: 13/14



The system clock comprises 3 555 timers, of which one pulses at a set rate, one allows you to step forward one clock step at a time, and the other lets you choose which of those two modes you want to be in. You can basically ignore those last two and think of them as switches, because they are – the only good thing about them is that they don't 'bounce' – a problem with most mechanical switches that I won't go into unless someone asks. Basically, this whole page is dedicated to either making a constant pulse of adjustable frequency, or allowing you to make the pulses yourself, at your own speed (for demonstration purposes). The control signal CE must be high for the clock pulses to be output, and so if it is set low, the computer will halt (stop), which is useful after a program has finished execution so you can see the output.

Author: Sebastian Gaume
The clock module of the CPU – not very interesting, nor very fast, but very necessary.

Sheet: /Clock/
File: clock.sch

Title: Clock

Size: A4 Date: 2019-11-17 **Rev: 1**
KiCad E.D.A. kicad 5.1.4 Id: 14/14