**Nano8 General Purpose Computer**

by Jakob Flocke (TheCodeCurrents)

updated 02.04.2025

**User Manual**

Welcome to the Nano8 General-Purpose Computer Project. The goal of this project is to create a general-purpose computer that is simple enough for beginners to understand while maintaining strong performance. Currently, the project is still in its early stages, so this user manual serves more as a progress report—written as if the product were already complete.

The project aims to encompass a wide range of components, including:

* A custom CPU core
* Various computer designs built around the core
* A fully functional operating system
* An assembler and assembly language
* A simple programming language providing some assembly abstractions
* Multiple OS programs and applications
* A simple GPU for graphical applications

It will be implemented as an FPGA soft-core SoC, an emulator, and using TTL logic chips.

Ultimately, this project aims to be both an educational resource and just an enjoyable experience.

**Table of Contents**

[CPU Architecture 3](#_Toc1)

[Table of Contents 3](#_Toc2)

# **CPU Architecture**

TODO: update for new design

This section covers the CPU architecture, from a high level overview down to the function of each module.

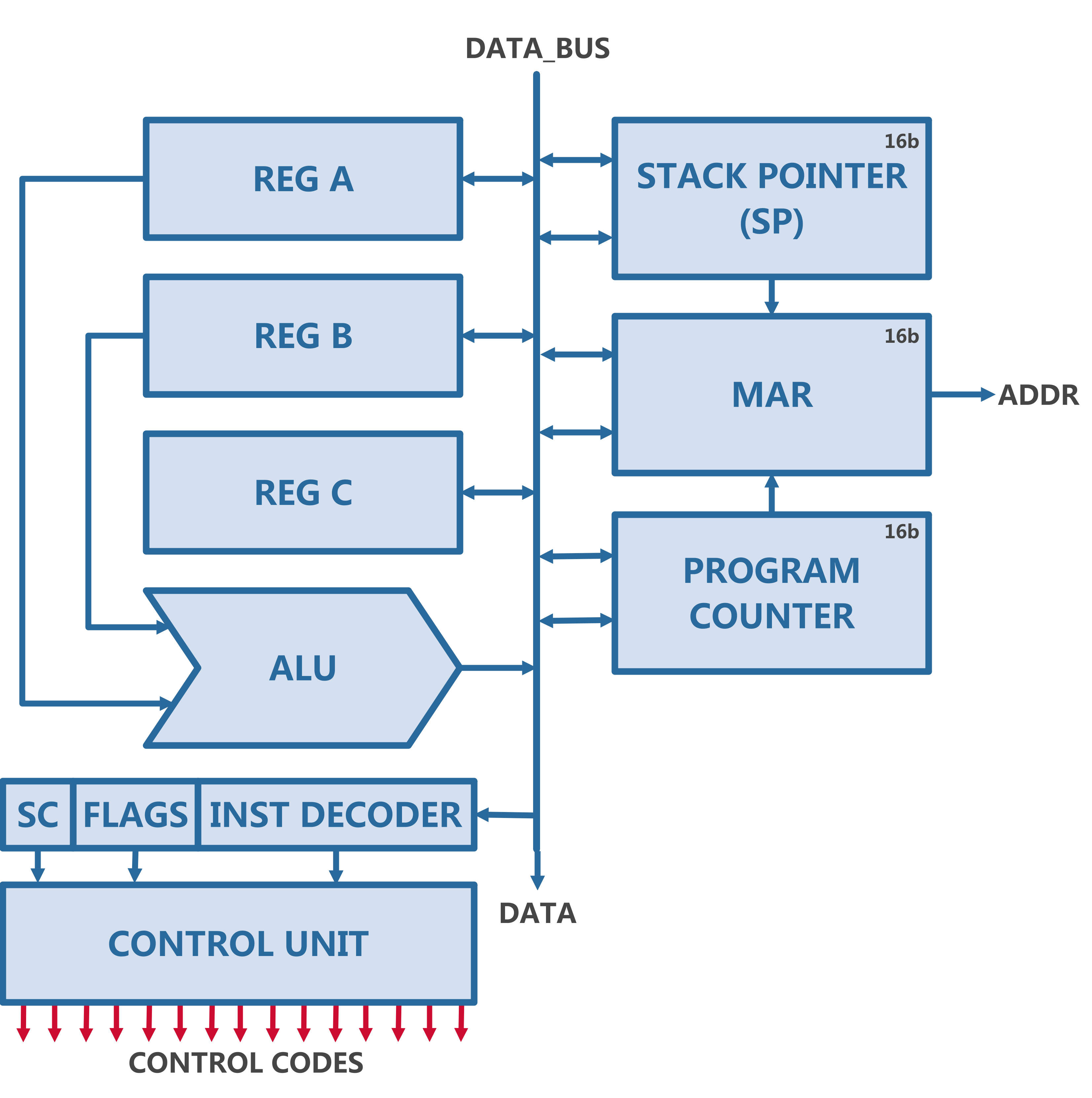


Figure 1: CPU Block Diagram

On a top level the CPU is made up of four individual sections:

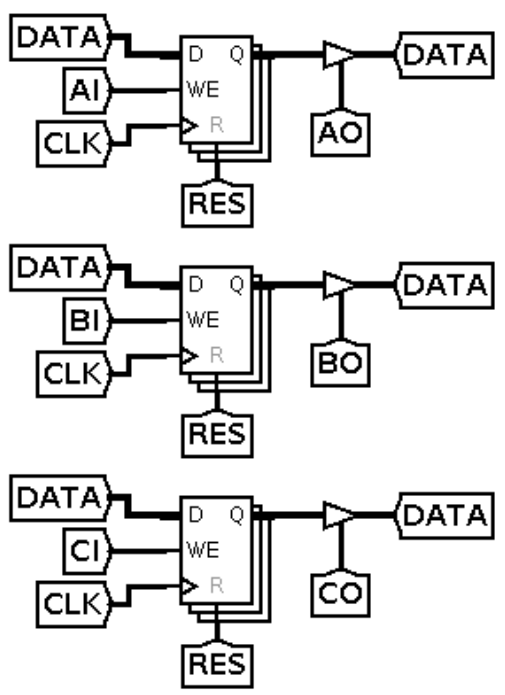
* Data section
  + Register A-C
* Arithmetic section
  + ALU
* Address section
  + stack pointer (SP), program counter (PC) and memory address register (MAR)
* Control section
  + Flags register, step counter (SC), instruction decoder and the control unit (CU)

Note: The clock signal needs to be provided.

## Data Section

The Data Section is the most simple one, it is consists of three 8-bit registers, which are connected to the Data BUS both on the input and on the output. Additionally they all have two control signals each, one for taking data in and the other for outputting the data onto the bus. Once a reset pulse is sent they all reset to their default value 0x00.

Figure 2: CPU Data Section (Logisim)



The behaviour of the registers is simple, they just store a single 8 bit value as long as they are not reset. When the IN (AI, BI or CI) control signal of a register is high, the registers will read the data from the bus on the rising edge of the clock. When the OUT (AO, BO or CO) control signal is high, the value will be outputted to the bus unrelated to the clock signal.

The only instructions that could be done with only the data section (if control signals are controlled manually) are basics like a move operation, where a value is moved from one register to the other or a switch operation, where a third register is used as a buffer to switch the values of two registers.

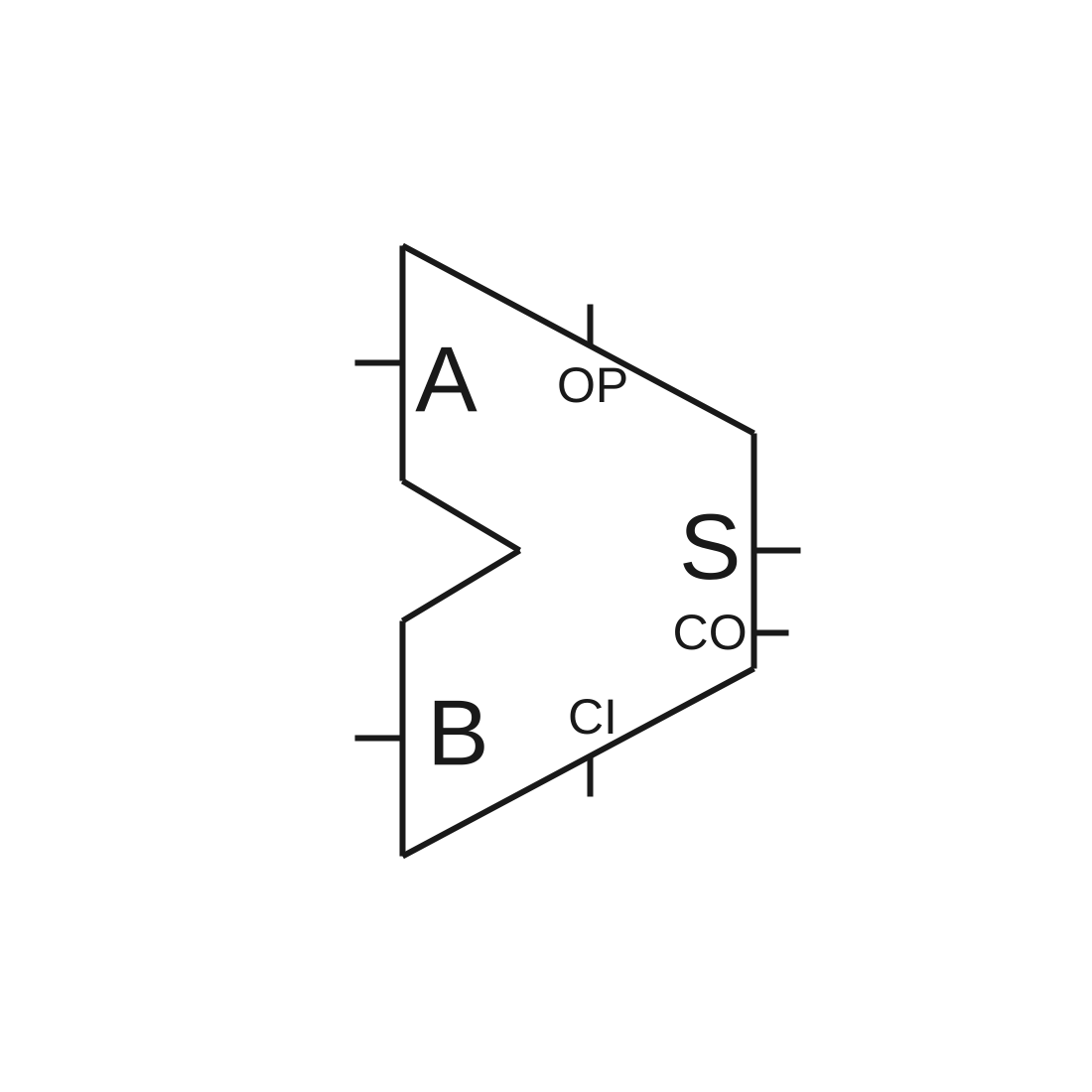
## Arithmetic Section

The Arithmetic Logic Unit (ALU) is a single but essential component in the arithmetic section of a CPU. It is responsible for executing all mathematical and logical operations such as addition, subtraction, logical shifts, and bitwise operations. Despite being just one unit, the ALU can be considered the core of the CPU, as it directly impacts the state of the machine. Without it, the CPU wouldn't be able to perform computations or make decisions, making the ALU integral to the functioning of the processor.

Within the ALU, there are several submodules or units responsible for different tasks:

1. **Adder / Subtractor:** This module is responsible for adding and subtracting numbers. It uses two’s compliment for negative values, ensuring that both positive and negative values get treated correctly.
2. **Shifter:** This unit performs bit-shifting operations, it can do both logical and arithmetic shifts, depending on whether the sign of the number preserved.
3. **Comparator:** The comparator is responsible for every comparison such as equality, greater than or less then.
4. **Logical:** The logical unit implements logical operators like in boolean algebra on a bit level. An example of a bitwise and would be 0001 & 0101 = 0001. Besides the and operation there are also or, xor and the not operation.

### Inputs & Outputs



The ALU has main in- and outputs of the ALU are A, B and S, they are all 8-bit wide and are the two inputs and the output. An Addition would be S = A + B. Additionally there are three other connections, called CI (carry in), CO (carry out) and OP (operation). An addition wit carry in high would look like this: S = A + B + 1.

Figure 3: ALU Block

### Operations

Since the OP signal is 4 bit wide, the ALU could theoretically perform 16 logic functions, but only 15 are used. This way the ALU doesn’t need an additional enable signal, when it’s not used the OP is just 0 and when it is used the OP is set to whatever operation should be executed. Additionally the carry in signal is important for the differentiation of the functions.

Hint: The table below only shows what the ALU itself is capable of, not every one of those operations is supported by every ISA. Therefore the Operations are not linked to instructions.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **CI = 0** | | | **CI = 1** | | |
| **#** | **Binary** | **Operation** | **S** | **CO** | **Operation** | **S** | **CO** |
| 0 | 0000 | NOP | A | 0 | INC | A + 1 | X |
| 1 | 0001 | ADD | A + B | X | ADD\_INC | A + B + 1 | X |
| 2 | 0010 | SUB | A - B |  |  |  |  |
| 3 | 0011 | SHL | A << 1 |  |  |  |  |
| 4 | 0100 | SHR | A >> 1 |  |  |  |  |
| 5 | 0101 | AND |  |  |  |  |  |
| 6 | 0110 | OR |  |  |  |  |  |
| 7 | 0111 | XOR |  |  |  |  |  |
| 8 | 1000 | NOT |  |  |  |  |  |
| 9 | 1001 | GT |  |  |  |  |  |
| A | 1010 | LT |  |  |  |  |  |
| B | 1011 | EQ |  |  |  |  |  |
| C | 1100 |  |  |  |  |  |  |
| D | 1101 |  |  |  |  |  |  |
| E | 1110 |  |  |  |  |  |  |
| F | 1111 |  |  |  |  |  |  |

# Control Unit

As the name suggests the control unit is responsible for controlling the processor. It reads the instructions (like commands for the CPU) from memory in the first clock cycle of every execution and then starts to orchestrate everything that needs to happen to execute a specific instruction.

## Instruction

The control unit works based on instructions, there are a total of 16 instructions in the instruction set of the Nano8v1. Every instruction is made up of the opcode and parameters, that vary based on the opcode. For example the ALU instruction has a 4-bit opcode that specifies the specific operation that will be executed. In the Nano8v1 ISA (instruction set architecture) an instruction is at least 16 bit long. Some also have additional parameters that get used directly, that way some instructions can reach 3 or even four bytes in length.

# Instruction Set Architecture

A common problem for 8-bit processors is that the ISA is often a trade-off, between a nice design with enough features and efficiency, because it’s hard to include much information inside an 8bit instruction and a 16-bit instruction takes double the time to fetch. Of course there are solutions like caching the instructions, pipe-lining and more, but they are often disregarded because of their complexity. Because of that most hobbyist 8-bit computers use a trick, that is a little dirty, they don’t really have an ISA at all and just use the 8bit instruction as an address for a memory chip, that has all the control signals for an instruction hardcoded. This provides a lot of benefits, as you now have 256 instructions, that can do what every you like them to do, white-out any constraints. However, the approach is a bit inelegant and not teasable for many implementations. For example on FPGA’s for example that would be a big waste of LUTs and for a real chip that would be much wasted silicon.

For the Nano8v1 I decided not to use such ROM to decode instructions, but to have a decoder that fetches an 8bit instruction and then based on the opcode decides if it needs to fetch a second byte into the 16 bit instruction register. This way an instruction can be 8- bits, 16-bits or longer. The instruction register itself is only 16bit long, but additional 1 or two byte long parameters can be used directly from memory without being fetched into the decoder.

## Instruction Formats

Obviously different instructions need different information to work, some need a register and an address, while others need two registers and an ALU operation. Therefore there are multiple different formats, that the instructions can be in. The shortest are one byte long, while the longest are two bytes long. As discussed previously the length of the instruction dictates how many cycles the fetch takes, but immediate values and addresses don’t increase the fetch cycles, because they can be read when needed. (That doesn’t necessarily mean that they don’t cause a delay)

|  |  |  |  |
| --- | --- | --- | --- |
| **Format** | **Use Case** | **Layout Example** | **Format ID** |
| C-Types | Custom (can do whatever) | OP[8] / ID[3] META[5] | 0 |
| R-Type | Register-Register | OP[4] RD[4] RS[4] ALU[4] | 1 |
| B-Type | Branch / Jumps / Calls | OP[4] COND[3] T[1] (OFF[8] / ADDR[16]) | 2 |
| I-Type | Register-Immediate | OP[4] RD[4] IMM[8] | 3 |
| M1-Type | Register-Memory | OP[4] T[1] REG[3] ADDR[16] | 4 |
| M2-Type | Register-Memory | OP[4] T[1] REG[3] IMM[8] | 5 |
| U-Type | Unary operations (quicker) | OP[4] REG[4] | 6..7 |

Table 1: Instruction Formats

The one component all these formats have is the opcode (OP), it is the code that says what the instruction is generally about. The first three bits of the instruction specify the format using the format id. Since the format id is 3-bit long, one format id can only have two different opcodes, to increase that amount you need multiple format ids for one format. Of course these opcodes don’t link directly to instructions, there can be many instructions using the same format because of meta parameters.

### C-Type (Custom)

Custom instructions use an 8-bit opcode with no encoded parameters, but they can accept full byte-sized parameters, such as an 8-bit immediates (imm8) or a 16-bit addresses (addr16). For example nop, hlt and similar instructions can be found here.

### R-Type (Register)

Register to register operations with an optional 4-bit ALU value are handled by the R-Type instruction format. It includes every ALU operation and every other operation done between two registers, like moves and switches.

(Maybe this needs to be sized up, to fit ALU with and without carry and to hold stuff like moves)

### B-Type (Branch)

The Nano8 is capable of three kinds of branches; non conditional branches (jumps), conditional branches and subroutine calls. They all jump based on a 8-bit offset or a 16-bit address. The COND bits decide, which flags to check and the T bits sets the addressing mode either to absolute (16-bit adresses) or to relative (+-~128), indirect and immediate.

### I-Type (Immediate)

An I-Type instruction takes a register and an 8-bit immediate in, which is technically two bits, but is fetched in one cycled, because the second byte can be fetched whenever it is needed.

### M-Type (Memory)

The M-Type instructions are a bit different and sadly a bit inelegant, because there was no way to handle all the necessary information for memory instructions in just two opcodes (one format id). There are four thing that would optimally be included: the 4-bit opcode, the 4-bit register code, the 2-bit addressing mode and the immediate or the address. The problem is, that those are two too much and I didn’t find an elegant solution, so I just split it into two and removed one bit from the register address.

Now the M1-type handles loading and storing values to and from memory using absolute and indirect addressing modes, while the M2-type does the same using relative addressing or zero-page addressing.

### U-Types (Unary)

Instructions that only need a single register to operate are formatted in the U-type. That would be specifically the INC, DEC, PUSH and POP instructions.