**Nano8 General Purpose Computer**

Version 1.0  
April 2025

By Jakob Flocke  
(TheCodeCurrents)

**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**

[1 CPU Architecture 3](#_Toc1)

[1.1 Data Section 3](#_Toc2)

[1.2 Arithmetic Section 4](#_Toc3)

[1.2.1 Inputs & Outputs 4](#_Toc4)

[1.2.2 Operations 5](#_Toc5)

[1.3 Address Section 6](#_Toc6)

[1.4 Design 6](#_Toc7)

[1.5 Control Unit 7](#_Toc8)

[1.6 Instruction 7](#_Toc9)

[2 Instruction Set Architecture 8](#_Toc10)

[2.1 Instructions 8](#_Toc11)

[2.2 Instruction Formats 10](#_Toc12)

[2.2.1 C-Types 10](#_Toc13)

[2.2.2 M-Type 10](#_Toc14)

[2.2.3 B-Type 11](#_Toc15)

[2.2.4 A-Type 12](#_Toc16)

[2.2.5 S-Type 12](#_Toc17)

[2.2.6 R-Type 13](#_Toc18)

[3 Appendix 14](#_Toc19)

[3.1 List of Tables 14](#_Toc20)

[3.2 List of Figures 15](#_Toc21)

# Nano8

The Nano8 is a homebrew 8-bit computer project born from curiosity, driven by ambition, and guided by simplicity. What began as a modest goal—to build an 8-bit CPU with perhaps 16-bit addressing—has evolved into a full-fledged computing system with astronomical aspirations. Yet, despite the expanded scope, the core principles remain unchanged:

* No compromises
* Simple, elegant design
* Educational value above all

The only thing that has changed is the scale of the vision. Some might argue that increasing scope undermines simplicity. I see it differently: the Nano8 strives to be as simple as possible for what it is. It’s not about minimalism for its own sake—it’s about clarity, transparency, and accessibility in design.

## Project Goals

The ultimate goal of the Nano8 journey is to create a fully functional computer system composed entirely of CMOS logic chips. This includes:

* An 8-bit CPU
* An 8-bit GPU
* Up to 16 MiB of memory
* A hardware bootloader
* A multitasking operating system
* Manually implemented control logic at 0.5 IPC
* All built from discrete logic, with no microcontrollers or prebuilt cores

## CPU Specification

The Nano8 CPU is designed to be both powerful and understandable. It balances performance with architectural clarity.

* Data Bus: 8-bit
* Address Bus: 16-bit physical, extended via MMU
* MMU: 24-bit virtual memory addressing
* Stack: Hardware-supported stack for subroutine calls and interrupts
* Clock Frequency: 10 MHz at first, someday hopefully >40MHz
* Interrupt System: Vector-based, with support for multiple interrupt sources
* Adjustable Clock & Voltage: For performance tuning and power efficiency
* Instruction Efficiency: ~1 IPC (Instruction Per Cycle), enabling high throughput
* Design Philosophy: Fully synchronous, minimal control logic, no microcode

## GPU Specifications

* Color Depth: 8-bit indexed color (256-color palette)
* Sprites: Hardware sprite engine with transparency and layering
* Text Mode: Programmable character set, optional text overlay
* Resolution: Configurable, up to 320×240 pixels
* Memory Access: DMA interface for fast data transfer from system RAM
* Video Output: Composite or VGA (depending on implementation)
* Timing: Synchronized with CPU or independently clocked

## System Architecture

* Memory: Up to 16 MiB addressable via MMU
* Bus Design: Shared system bus with arbitration logic
* Peripherals: Serial I/O, keyboard interface, optional storage
* Power Management: Adjustable voltage rails for experimentation
* Expandability: Modular design for adding new components (e.g., sound, networking)

## NanOS

NanOS is the multitasking operating system designed specifically for the Nano8 architecture. It is built entirely from discrete logic and tailored to the constraints and strengths of the system. Despite its simplicity, NanOS provides a robust foundation for multitasking, I/O, and user interaction.

### Core Features

* Multitasking Kernel Supports cooperative multitasking with optional preemptive extensions. Tasks are managed via a lightweight scheduler and context-switching mechanism.
* Memory Management Utilizes the 24-bit MMU to provide virtual memory addressing. Supports paging and memory protection for isolated task execution.
* Interrupt Handling Vector-based interrupt system allows responsive I/O and timer-driven task switching.
* Shell Interface A minimal command-line interface accessible via serial or video output. Supports basic commands, task management, and file operations.

### Storage System

Primary Storage: Non-volatile FRAM (Ferroelectric RAM)

**Advantages:**

* Fast read/write speeds comparable to SRAM
* Virtually unlimited write cycles
* No need for wear leveling or flash management
* Retains data without power

**Use Cases:**

* Persistent file system
* Configuration storage
* Task state preservation across reboots

**File System:**

* Simple flat file system or directory-based structure
* Metadata stored in reserved FRAM sectors
* Optional journaling for data integrity

### Modular Drivers

NanOS includes modular drivers for:

* GPU: Text and graphics output
* Serial I/O: Communication with external devices or host systems
* Keyboard: PS/2 or matrix scanning
* FRAM: Read/write interface with block-level access

### Development Tools

The on board development tools will hopefulyl include a text editor, a compiler, a linker and some more utility things. I hope to be able to develop updates for the OS on the OS (which is why I am so willing to invest in high RAM suppor).

# CPU Architecture

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

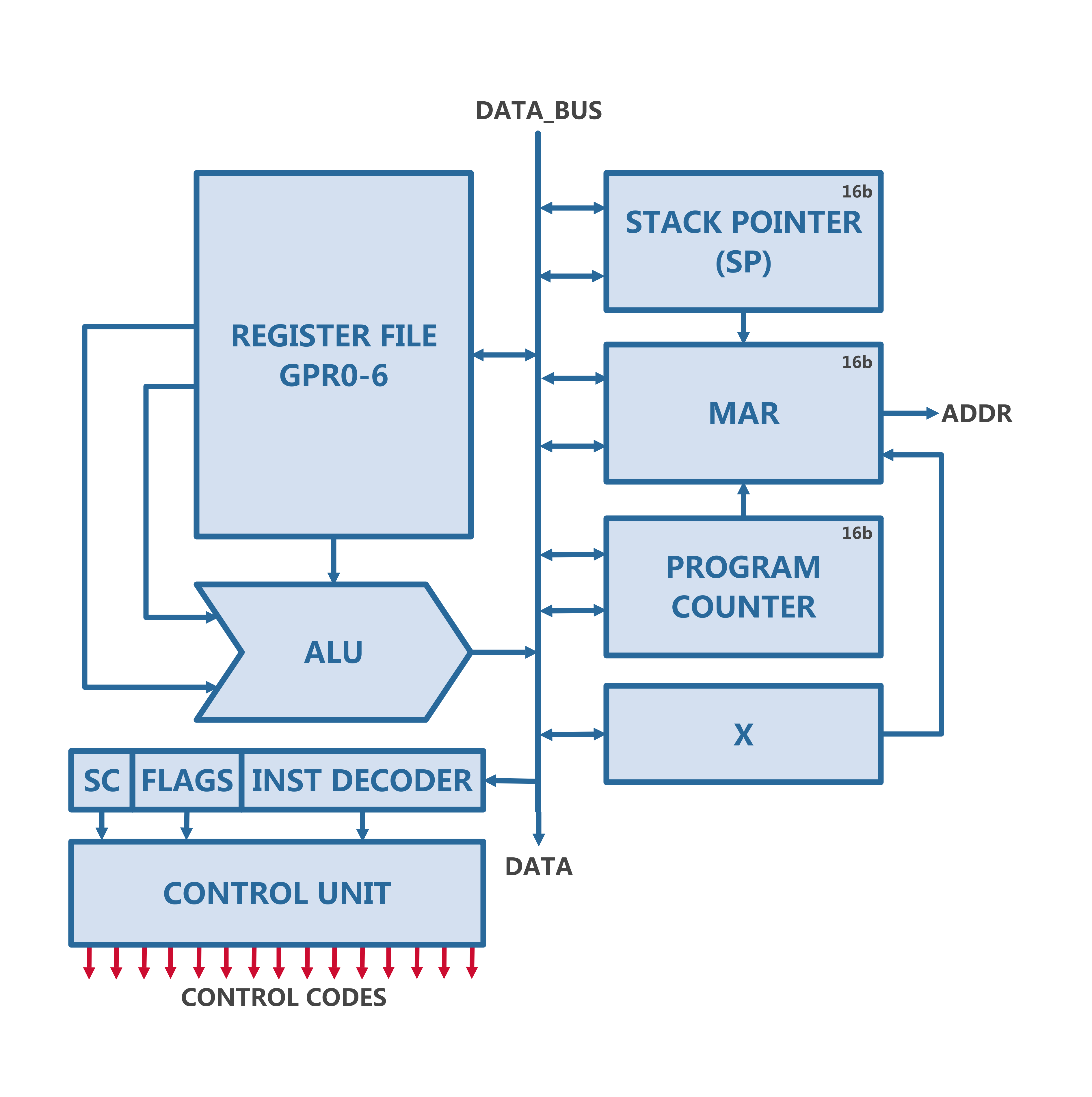


Figure 1: CPU Block Diagram

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

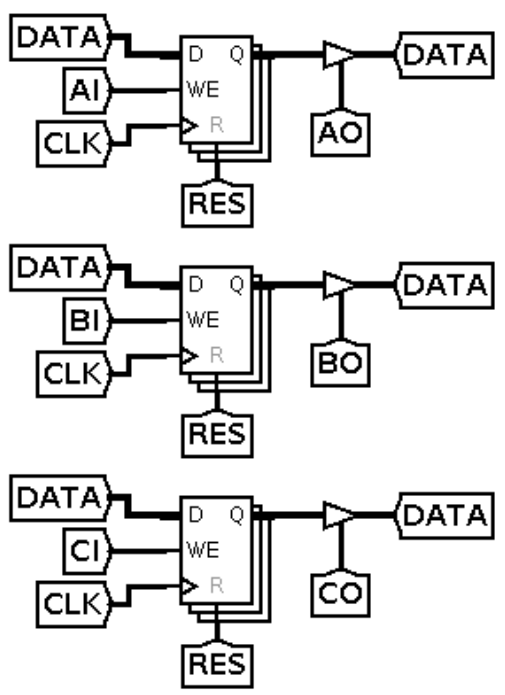
* Data section
  + 7 general purpose registers (GPR)
* Arithmetic section
  + ALU implementing 16 operations and carry variants
* Address section
  + stack pointer (SP), program counter (PC) and memory address register (MAR) and X register
* Control section
  + Flags register, step counter (SC), instruction 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 consists of seven 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 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 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.

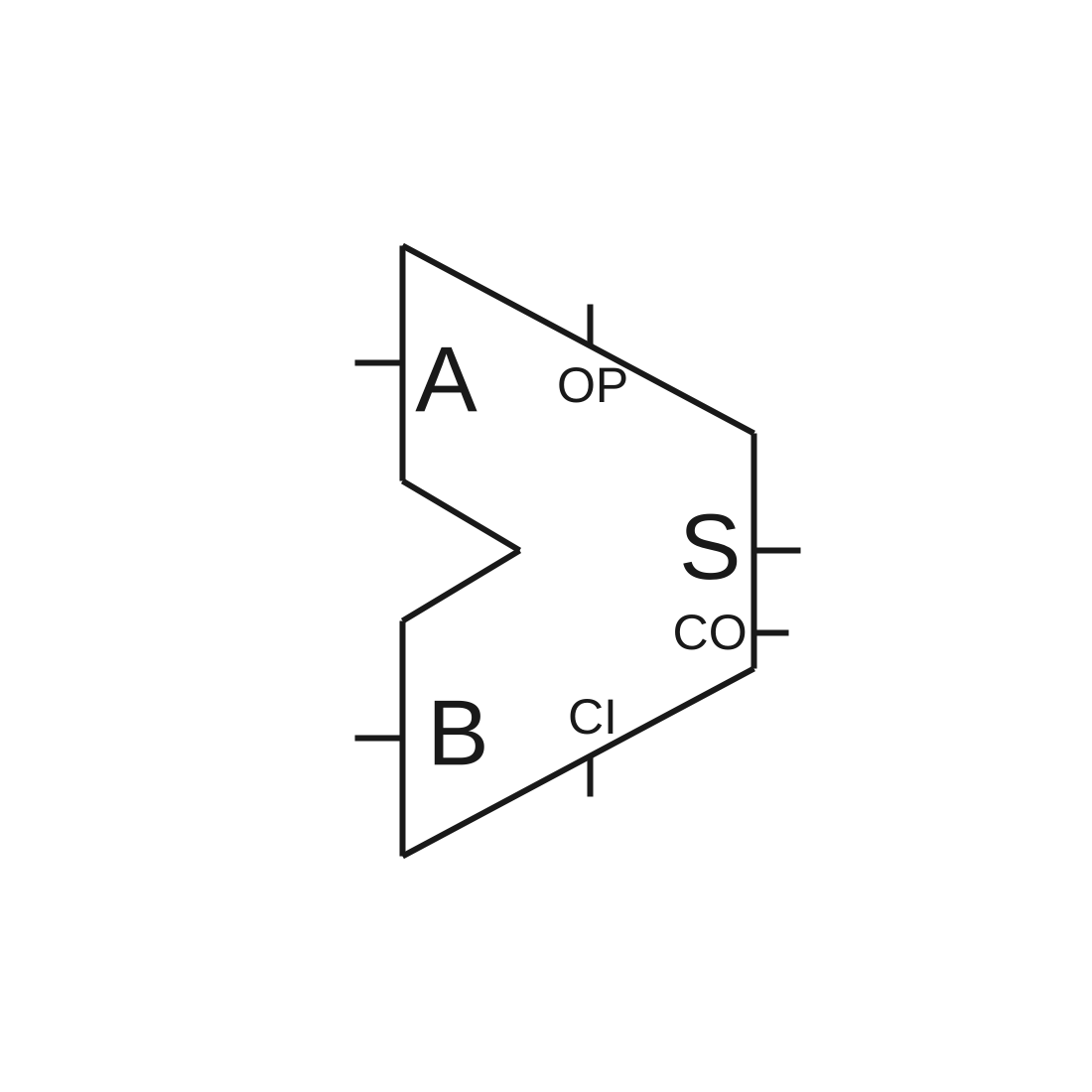
## Arithmetic Section

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

The ALU can perform

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **CI = 0** | | **CI = 1** | |
| **#** | **Binary** | **Operation** | **S** | **Operation** | **S** |
| 0 | 0000 | ADD | A + B | ADDC | A + B + C |
| 1 | 0001 | SUB | A - B | SUBC | A – B - C |
| 2 | 0010 | AND | A & B |  |  |
| 3 | 0011 | OR | A | B |  |  |
| 4 | 0100 | XOR | A ^ B |  |  |
| 5 | 0101 | CMP | ? |  |  |
| 6 | 0110 | SET | A |= (1 << B) |  |  |
| 7 | 0111 | CLR | A &= ~(1 << B) |  |  |
| 8 | 1000 | TGL | A ^= (1 << B) |  |  |
| 9 | 1001 | NOT | ~ A | NEG | ~ A - 1 |
| A | 1010 | INC | A + 1 | INC2 | A + C |
| B | 1011 | DEC | A - 1 | DEC2 | A - C |
| C | 1100 | SHL | A << 1 | SHLC |  |
| D | 1101 | SHR | A >> 1 | SHRC |  |
| E | 1110 | ASL | A << 1 | ASLC |  |
| F | 1111 | ASR | A >> 1 | ASRC |  |

Table 1: ALU Functions

## Address Section

To store values in memory, there has to be some sort of system, that enables the computer to keep track of where they stored a value and where they can retrieve it again. For this purpose there are memory addresses, which are just numbers that correspond to a specific byte of storage.  
In 8 bit computers the memory addresses are a limiting factor, because with 8 bit memory addresses, there can only be 2⁸ different memory locations, which would be 256 bytes. Compared to a common modern computer with about 16 GB of RAM and terabytes of data storage, this is peanuts. We obviously won’t reach such heights, but there are some techniques to increase the amount of memory we have.

The *first* is to just double the width of the memory addresses to 16 bit. This immediately increases the address space to 2¹⁶, which is 64KiB of storage, but it also introduces overhead. Because the addresses are 2 words (a word is the data width of a CPU) wide, we need double the time to load them, because the upper and the lower byte need to be loaded separately. In addition everything you want to do with addresses in the CPU takes longer. Due to some small tricks, this will not be necessary all the time, but you should always be aware of the trade-offs, when designing a CPU.

The *second* trick is a bit more complex and maybe a bit overkill for this system, but the goal of this CPU is to run a nice multiprocess operating system, so I decided to use memory paging by implementing a simple MMU. If you don’t know what that means, it will be explained in more detail in the section 1.3.x MMU. Just know for now, that it’s a way to efficiently handle memory access from multiple processes. It ensures, that one process can’t overwrite another memory and it can shrink and grow based on what you need.

### Basic 16 bit Addresses

The CPU’s address section mainly operates on 16 bit addresses, that is the size of the program counter (PC), stack pointer (SP) and also of the memory access register (MAR). Each of them plays a unique role in managing the memory. The *program counter* does what the name suggests, it always points to the current (or next) instruction and automatically counts up when that instruction starts executing. The *stack pointer* always points at the top of the stack, which is where most values and pointers will be stored. The final block that connects these two with the address bus is the *memory access register,* which always puts it’s value directly on to the address bus. When you for example want to read the byte on top of the stack, you only have to write the *SP* value to the *MAR* and you can start reading the value from RAM. While the *PC* and *SP* are just simple counters, the *MAR* is a bit more complicated, as it doesn’t only support absolute addressing but also relational addressing. This means it needs to implement 16 bit adding and subtracting. (Which isn’t done via the ALU) To do that it has an optional 16-bit offset input and control lines for adding and subtracting.  
Note: Why did I plan that? I should do rel addressing for the PC and SP not the MAr, otherwise it‘s just a temporary offset ...

#### Program Counter

Like previously mentioned, the program counter keeps track of the position of the next instruction and is incremented whenever the a byte is fetched from the program. This is implemented as two 8bit counters, which can be individually written to and read from like all the other SPRs.

#### Stack Pointer

The stack pointer always holds the address of the top of the stack and is functionally the same as the program counter, but instead of counting up it counts down. This is useful, because the last 256 bytes of the stack are accessible by just adding a byte.

#### Memory Access Register

The memory access register is the unit that connects the program counter, the stack pointer and the custom offsets or addresses with the actual address bus. This is a bit more complicated, then the SP or the PC, as it has to decide, weather to put the value from the stack pointer or the program counter onto the address bus and maybe it even has to add or subtract a 16 bit offset.

### MMU

The MMU is a module that takes in the 16 bit CPU address bus and expands it to the 24 bit external address bus. It implements a basic hardware paging system, that uses the 16 bit CPU address as the virtual address space. There are 65536 1KB pages, which totals to 64MiB of RAM, alternatively if the pages are 256 B big, there would be 16 MiB of RAM.

## 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

Every program is nothing more than a sequence of instructions, these can be things like adding values, storing them or jumping to another address. These instructions are given to the control unit, which decodes them and controls the cpu according to what it‘s supposed to do.

# 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.

## Instructions

As the ISA focuses on versatility over design simplicity, there are many different instructions, based on the formatting types discussed previously. There are many register manipulation operations, memory operations with different addressing modes, branches with different conditions and addressing modes, many ALU operations and some other ones.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction** | | | **Description** | **Operands** | **Length** |
| **OP [3]** | **Class** | **Mne** |  |  | **Cycles** |
| 000 | C | NOP | Does nothing | - | 2 or 1 |
| 000 | C | HLT | Halts the CPU until manually released | - | ? |
| 000 | C | RES | Resets the entire CPU | - | 2 |
| 000 | C |  |  | - |  |
| 000 | C |  |  | - |  |
| 001 | M | LD | Loads value from memory into registers (available in immediate, absolute, relative and indirect mode) | SRC, DST |  |
| 010 | M | ST | Stores value from registers into memory (available in absolute, relative and indirect mode) | SRC, DST |  |
| 011 | B | JMP | Unconditionally jump to absolute, relative or indirect addresses | Target |  |
| 011 | B | BRP | Branch if positive | Target |  |
| 011 | B | BRN | Branch if negative | Target |  |
| 011 | B | BRZ | Branch if zero | Target |  |
| 011 | B | BRNZ | Branch if not zero | Target |  |
| 011 | B | BRO | Branch if OVF | Target |  |
| 011 | B | BRNO | Branch if no OVF | Target |  |
| 100 | A | ADD | Add two register (A = A + B) | A, B |  |
| 100 | A | ADDC | Add two register (A = A + B + O) | A, B |  |
| 100 | A | SUB | Subtract two register (A = A - B) | A, B |  |
| 100 | A | SUBC | Subtract two register (A = A - B - O) | A, B |  |
| 100 | A | AND | Bitwise and (A = A & B) | A, B |  |
| 100 | A | OR | Bitwise or (A = A | B) | A, B |  |
| 100 | A | XOR | Bitwise xor (A = A | B) | A, B |  |
| 100 | A | CMP | Set Flags based on A and B | A, B |  |
| 100 | A | SET | Set specific bit in register | A, B |  |
| 100 | A | CLR | Clear specific bit in register | A, B |  |
| 100 | A | TGL | Toggle specific bit in register | A, B |  |
| 100 | A | NOT | Bitwise inversion (A = ~A) | A |  |
| 100 | A | NEG | Convert to 2’s complement (A = ~A + 1) | A |  |
| 100 | A | INC | Increment register | A |  |
| 100 | A | INC2 | Increment register by 2 | A |  |
| 100 | A | DEC | Decrement register | A |  |
| 100 | A | DEC2 | Decrement register by 2 | A |  |
| 100 | A | SHL | Shift one bit to the left | A |  |
| 100 | A | SHLC | Shift one bit to the left with carry in | A |  |
| 100 | A | SHR | Shift one bit to the right | A |  |
| 100 | A | SHRC | Shift one bit to the right with carry in | A |  |
| 100 | A | ASL | Arithmetic shift one bit to the left | A |  |
| 100 | A | ASLC | Arithmetic shift one bit to the left with carry in | A |  |
| 100 | A | ASR | Arithmetic shift one bit to the right | A |  |
| 100 | A | ASRC | Arithmetic shift one bit to the right with carry in | A |  |
| 101 | S | PUSH | Push SRC value onto the Stack (works with immediate, relative, register and PC addressing) | SRC |  |
| 110 | S | POP | Pop one or more bytes from stack (returns top to DST) | DST |  |
| 110 | S | PEEK | Load the top (or 8-bit offset) value from stack to DST | DST |  |
| 111 | R | SWI | Switch R1 and R2 | R1, R2 |  |
| 111 | R | MOV | Move SRC into DST | R1, R2 |  |

Table 2: Instruction listing

## 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** |
| C-Types | Custom | OP[8] / ID[3] META[5] |
| M-Type | Load and Store | OP[3] AD[2] REG[3] (OFF[8] / ADDR[16]) |
| B-Type | Branches and Jumps | OP[3] COND[2] AD[2] RES [1] (OFF[8] / ADDR[16]) |
| A-Type | ALU | OP[3] CI[1] ALU[4] DES[4] SRC[4] |
| S-Type | Stack | See S-Type |
| R-Type | Register-Memory | OP[3] T[1] REG[3] IMM[8] |

Table 3: Instruction formats

### C-Types

The C-Types (custom) is a collection of instructions, that didn’t fit into the other categories. These will likely be decoded using a small ROM to store the custom microcodes for up to 8 (or maybe 4) steps. These include core instructions like NOP, HLT and RST.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **op [3]** | | | **meta op [5]** | | | | |

Table 4: C-Type Instructions

### M-Type

The memory type instructions have the codes 001 (Load) and 010 (Store) and take a 2 bit addressing mode.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |  | **8 - F** | **10 - 1F** |
| **op [3]** | | | **ad [2]** | | **reg [3]** | | | **imm [8] | null | addr [8]** | **addr [8]** |
| 0 | 0 | 1 | 0 | 0 | reg [3] | | | imm [8] | / |
| 0 | 0 | 1 | 0 | 1 | reg [3] | | | addr [8] | addr [8] |
| 0 | 0 | 1 | 1 | 0 | reg [3] | | | imm [8] | / |
| 0 | 0 | 1 | 1 | 1 | reg [3] | | | addr [8] | addr [8] |
| 0 | 1 | 0 | 0 | 0 | reg [3] | | | imm [8] | / |
| 0 | 1 | 0 | 0 | 1 | reg [3] | | | addr [8] | addr [8] |
| 0 | 1 | 0 | 1 | 0 | reg [3] | | | imm [8] | / |
| 0 | 1 | 0 | 1 | 1 | reg [3] | | | addr [8] | addr [8] |

Table 5: M-Type Instructions

|  |  |  |
| --- | --- | --- |
| **ad** | **Mode** | **Length** |
| 01 | Immediate | 3 |
| 10 | Absolute | 3 |
| 11 | Relative (1 byte + X as HB) | 2 |
| 11 | Indirect (pointer) | 3 |

Table 6: M-Type addressing modes

### B-Type

For the branches there is only a single opcode (011), followed by two bits for the condition field and two bits for the addressing mode, followed by the inv bit, that can invert the condition. Optionally (depending on the ad bits), the next two bytes contain an 8bit immediate value or a 16 bit address.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |  | **8 - F** | **10 - 1F** |
| **op [3]** | | | **cond [2]** | | **ad [2]** | | **inv** | **imm [8] | null | addr [8]** | **addr [8]** |
| 0 | 1 | 1 | x | x | 0 | 0 | 0 | imm [8] | / |
| 0 | 1 | 1 | x | x | 0 | 1 | 0 | addr [8] | addr [8] |
| 0 | 1 | 1 | x | x | 1 | 0 | 0 | imm [8] | / |
| 0 | 1 | 1 | x | x | 1 | 1 | 0 | addr [8] | addr [8] |
| 0 | 1 | 1 | x | x | x | x | 1 | none | none |

Table 7: B-Type Instructions

|  |  |  |
| --- | --- | --- |
| **#** | **Condition** | **Used in** |
| 00 | None | JMP |
| 01 | Zero | BRZ |
| 10 | Negative | BRN |
| 11 | Carry | BNC |

Table 8: B-Type Conditions

|  |  |  |
| --- | --- | --- |
| **#** | **Mode** | **Length** |
| 00 | Reserved | / |
| 01 | Absolute | 3 |
| 10 | Relative (1 byte + X as HB) | 2 |
| 11 | Indirect (pointer) | 3 |

Table 9: B-Type addressing modes

### A-Type

There are 16 possible ALU operations, which could all have a CI variant, for instructions where a carry in isn’t logical, the CI signal can act like a 5th OP signal. Originally the msb of the OP signals determined if the operation used one or two operands, which could theoretically reduce the cycles per fetch. Sadly I didn’t manage to find an elegant way to incorporate this.

The ALU instruction allows the use of 4bit register addresses, which should be used with caution. I will dive a bit deeper into this in the R-Type instruction section. If you only want to use the normal 3bit addresses you can use them just like normal.

100 + CI + ALU [4] + DES[4] + SRC[4]

### S-Type

PUSH (101)

op [3] ad [2] (r[3] | res[3]) (imm[8] | addr[16] | null)

|  |  |  |
| --- | --- | --- |
| **ad** | **Mode** | **Length** |
| 00 | immediate | 1 |
| 01 | register | 2 |
| 10 | Relative (1 byte + X as MSB) | 2 |
| 11 | PC | 3 |

Table 10: S-Type PUSH addressing

POP (110)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |  | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **op [3]** | | | **pop** | **ad** | **target reg** | | | **imm [8] | null** | | | | | | | |
| 1 | 1 | 0 | x | x | reg | null | | | imm [8] | null | | | | | | | |
| 1 | 1 | 0 | x | 0 | reg | null | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | x | 1 | reg | null | | | x | x | x | x | x | x | x | x |
| 1 | 1 | 0 | 0 | x | reg | | | imm [8] | null | | | | | | | |
| 1 | 1 | 0 | 1 | x | null | | | imm [8] | | | | | | | |

Table 11: S-Type POP instruction

|  |  |  |
| --- | --- | --- |
| **ad** | **Mode** | **Length** |
| 0 | Single byte | 1 |
| 1 | N bytes (imm8 as LSB + X as MSB) | 2 |

Table 12: S-Type POP addressing

Hint: It isn’t possible to combine pop = 0 and ad = 1, as it’s impossible to read multiple bytes into memory.

### R-Type

The R-Type instructions are instructions like MOV or SWI, which operate on two registers. Like the ALU instructions, they use 4bit register addresses instead of 3bit, so that they can directly manipulate the special purpose registers. While this is exciting, it’s also a double edged blade and can cause a huge mess if done wrong, which is why I generally wouldn’t call this a smart design decision, but I had the space and didn’t want to waste it.

The layout of R-Type instructions is pretty simple, it starts as usual, with the 3 bit opcode. The next 5-bit specify the specific R-Type instruction and the second byte contains the 4-bit addresses of the destination and the source registers.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |  | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** |
| **op [3]** | | | **meta [5]** | | | | | **DES [4]** | | | | **SRC [4]** | | | |

# Appendix

## List of Tables

[Table 1: ALU Functions 5](#_Toc1)

[Table 2: Instruction listing 8](#_Toc2)

[Table 3: Instruction formats 9](#_Toc3)

[Table 4: C-Type Instructions 9](#_Toc4)

[Table 5: M-Type Instructions 9](#_Toc5)

[Table 6: M-Type addressing modes 10](#_Toc6)

[Table 7: B-Type Instructions 10](#_Toc7)

[Table 8: B-Type Conditions 10](#_Toc8)

[Table 9: B-Type addressing modes 10](#_Toc9)

[Table 10: S-Type PUSH addressing 11](#_Toc10)

[Table 11: S-Type POP instruction 11](#_Toc11)

[Table 12: S-Type POP addressing 11](#_Toc12)

## List of Figures

[Figure 1: CPU Block Diagram 3](#_Toc1)

[Figure 2: CPU Data Section (Logisim) 3](#_Toc2)

[Figure 3: ALU Block 4](#_Toc3)