Writing a simple 16 bit VM in less than 125 lines of C

andreinc.net/2021/12/01/writing-a-simple-vm-in-less-than-125-lines-of-c

December 1, 2021 30 minute read

This tutorial is intended for C beginners who want to do some coding practice and in the process, gain valuable insights regarding low-level programming and how (some) Virtual Machines operate under the hood.

By the end of the article, we will have a working register-based VM capable of interpreting and running a limited set of ASM instructions + some bonus programs to test if everything works well.

The code is written in C11, and it will probably compile on most operating systems. The repo can be found here, and the exact source code is `vm.c`:

```
git clone git@github.com:nomemory/lc3-vm.git
```

If you are a seasoned C developer that has already dabbled in this sort of stuff, you can skip this reading because it will cover information you probably already know.

The reader should already be familiar with bitwise operations, hexadecimal notation, pointers, pointer functions, C macros, and some functions from the standard library (e.g., `fwrite` and `fread`).

It will be unfair not to mention some existing blog posts covering the same topic as this article; the best in this category is Write your Own Virtual Machine by Justin Meiners and Ryan Pendleton. Their code covers a more in-depth implementation of a VM. Compared to this article, our VM is a little simpler, and the code takes a different route in terms of the implementation.

Later edit: After publishing this article in December, Philip Chimento was nice enough to write a Rust implementation of the same Virtual Machine. If you are curious to see how the solution looks like in another programming language, please check this out.

Table of contents

Virtual Machines

In the world of computing, a VM (*Virtual Machine*) is a term that refers to a system that emulates/virtualizes a computer system/architecture.

Broadly speaking, there are two categories of Virtual Machines:

- *System Virtual Machines* which provide a complete substitute for a real machine. They implement enough functionality that allows operating systems to function on them. They can share and manage hardware, and sometimes multiple environments can function on the same physical machine without hindering each other.
- *Process Virtual Machines* which are simpler and are designed to execute computer programs in a platform-agnostic environment. The JVM is a good example of a *Process Virtual Machine*.

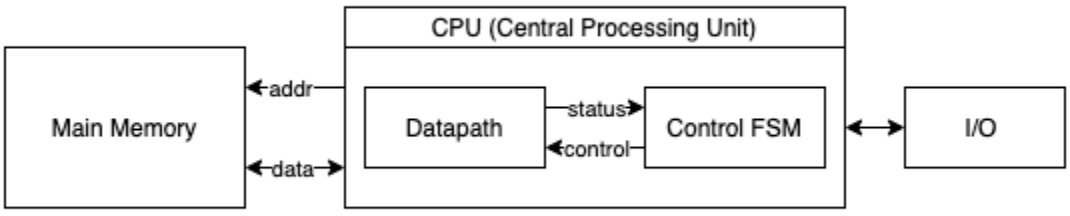
In this article, we will develop a simple *Process Virtual Machine* designed to execute simple computer programs in a platform-independent environment. Our *toy* Virtual Machine is based on the LC-3 Computer Architecture, and will be capable of interpreting and executing (a subset of) LC3 Assembly Code.

> Little Computer 3, or LC-3, is a type of computer educational programming language, an assembly language, a type of low-level programming language. It features a relatively simple instruction set but can be used to write moderately complex assembly programs and is a viable target for a C compiler. The language is less complicated than x86 assembly but has many features similar to those in more complex languages. These features make it worthwhile for beginning instruction, so it is most often used to teach fundamentals of programming and computer architecture to computer science and computer engineering students. (wikipedia)

For simplicity, we deliberately stripped down our LC-3 implementation from the following features: interrupt processing, priority levels, process, status registers (PSR), privilege modes, supervisor stack, user stack. We will virtualize only the most basic hardware possible, and we will interact with the *outside* world (`stdin`, `stdout`) through `traps`.

von Neumann model

Our LC-3 inspired VM, like most of the general-purpose computers nowadays, is based on the von Neumann computer model, and it will have three main components: the **CPU**, the **Main Memory**, the **input/output** devices.



The **CPU**, an abbreviation for *Central Processing Unit* is the “circuitry” that controls and manipulates data. Furthermore, the CPU is divided into three layers: **ALU**, **CU**, and **Registers**.

ALU stands for *Arithmetic/Logic Unit* and represents the circuits that are actually carrying the instructions on the data (operations like ADD, XOR, Division, etc.).

CU, an abbreviation for *Control Unit*, coordinates the activities on CPU.

The registers are quickly accessible “slots” located at the CPU level. The ALU operates on registers. They come in small numbers (that’s a relative statement, as it depends on the architecture), so the amount of data that can be *loaded* inside the CPU is limited. We use registers to interact with the *Main Memory*. A typical scenario involves loading a memory location into a register, performing some changes, and putting the data back into memory.

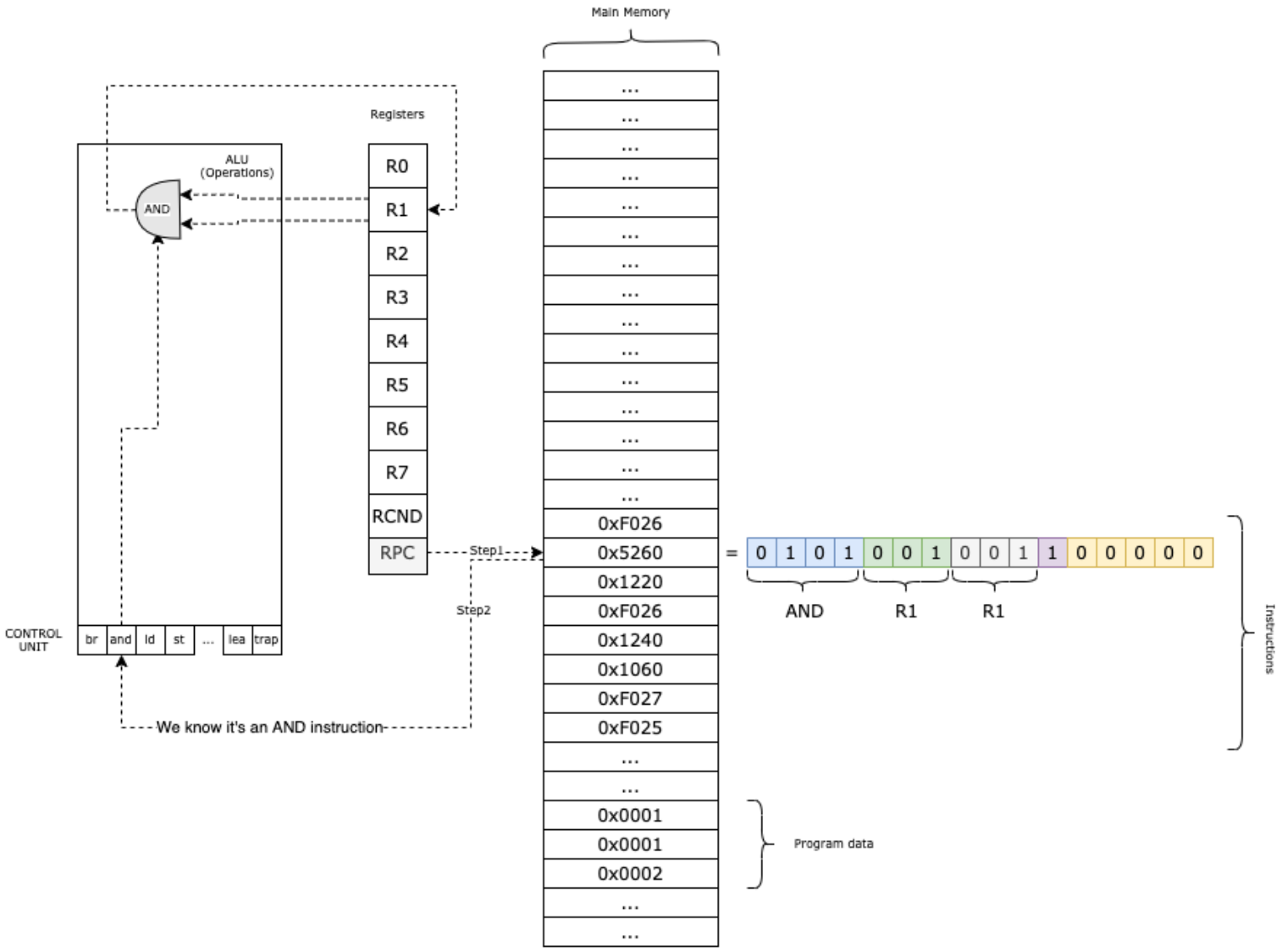
The **Main Memory** is imagined as an extensive “array” of `W` words of `N` bits each. Program instructions and the associated data are stored in the main memory in a binary format. Each memory *word* contains either one instruction or program data (e.g., a number used for computation).

The **Input/Output** devices enable the computer to communicate with the outside world.

Implementing the VM

Our VM functions like this:

- We load the program into the main memory;
- In the `RPC` register, we keep the current instruction that we need to execute;
- We obtain the *Operation Code* (first 4 bits) from the instruction and based on that, we *decode* the rest of the parameters.;
- We execute the method associated with the given instruction;
- We increment `RPC` and we continue with the next instruction;



The Main Memory

Our machine has `W=UINT16_MAX` words, of `N=16` bits each. From a C perspective our memory can be defined as:

```
uint16_t PC_START = 0x3000;
uint16_t mem[UINT16_MAX+1] = {0};
```

`UINT16_MAX` is the maximum size of a `uint16_t` (a unsigned 16 bits integer), `UINT16_MAX=65535`. So to put things into perspective, our system is quite limited. It won't run/load programs with more than `65535` instructions. I know it sounds harsh for our current times, but computers were much more humble than this back in the day (a few decades ago). `65535` is more than enough to write a few ASCII games and keep them all in memory.

As a convention, we should start loading programs into the main memory from `0x3000` onwards. We keep the memory slots up to `0x3000` reserved for other potential components, for example, a toy operating system. But who has time for something like this?

At this point, it's a good idea to write two functions for reading (`mr(...)`) and writing (`mw(...)`) to the main memory:

```
static inline uint16_t mr(uint16_t address) { return mem[address];  }
static inline void mw(uint16_t address, uint16_t val) { mem[address] = val; }
```

Even if this looks redundant to implement `mr` and `mw` (we can access the memory directly), in the future, we might add additional logic or impose some validations, so it's a good idea to keep this functionality isolated.

The registers

Our VM has a total of 10 registers, 16 bits each:

- `R0` is a general-purpose register. We are going to also use it for reading/writing data from/to `stdin`/`stdout`;
- `R1`, `R2`,..`R7` are general purpose registers;
- `RPC` is the program counter register. It contains the memory address of the next instruction we will execute.
- `RCND` is the conditional register. The conditional flag gives us information about the previous operation that happened at ALU level in the CPU.

From a code perspective we can implement them as follows:

```
enum regist { R0 = 0, R1, R2, R3, R4, R5, R6, R7, RPC, RCND, RCNT };
uint16_t reg[RCNT] = {0};
```

To access a register, we simply: `reg[R3]=...`.

The instructions

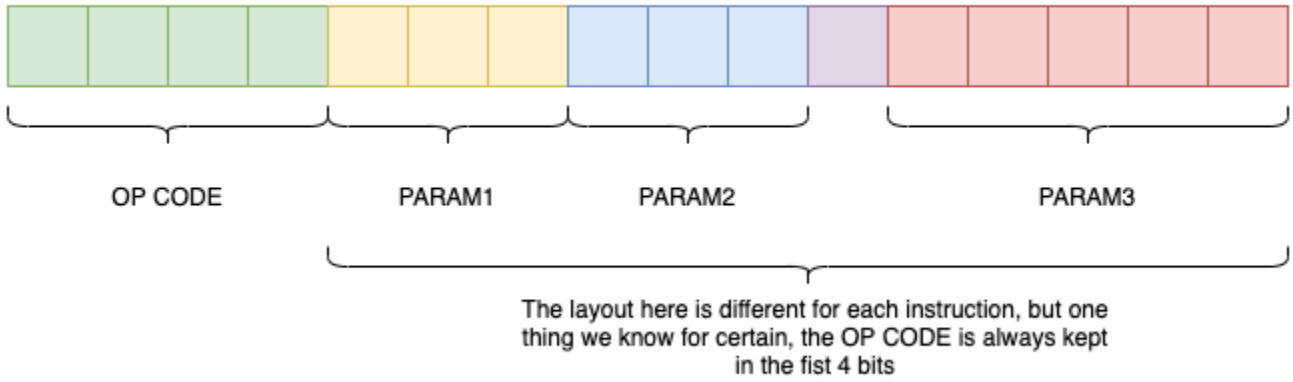
An instruction is like a *command* we give to the VM.

Through instructions, we ask the VM to perform a simple (and granular) task for us: read a char from the keyboard, add two numbers, perform a binary AND on a register, etc.

Instructions have the same word size as the memory, 16 bits. This is a natural decision; after all, we keep instructions loaded inside the *Main Memory*. So, from the C language perspective, instructions are `uint16_t` unsigned integers.

Our VM supports only a limited set of instructions: `16` (actually `14`, because two LC-3 instructions didn't make any sense to implement).

In terms of their format, instructions are usually *encoded* like this (inside a `uint16_t`):



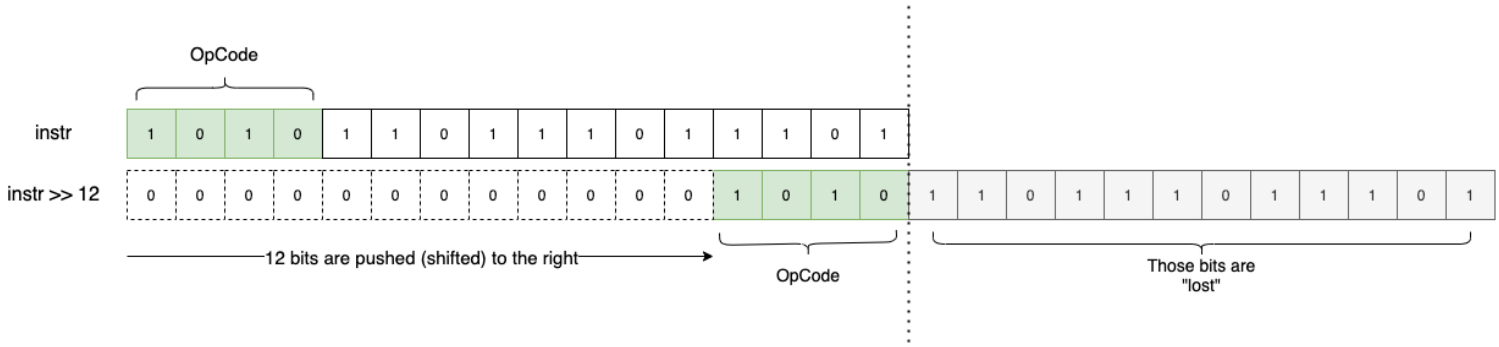
The first 4 bits are always the OpCode of the instruction. And then, depending on the instruction, there are 1,2,3,… params encoded in the remaining 12 bits.

Based on the OpCode, we can identify the instruction and understand how to "decode"/"extract" the rest of the params from the `uint16_t`.

For extracting the OpCode itself, we can write a utility macro that applies a simple bitwise trick:

```
#define OPC(i) ((i)>>12)
```

We shift 12 bits to the right (`i>>12`), to get the 4 most significant bits that contain the OpCode:



Because OpCodes are represented in 4 bits, the maximum number of instructions we can encode is 16 (`2^4=16`).

A nice trick we can perform in C (from a data modeling perspective) is to save all possible instructions (and their associated C functions) in an array. The index will represent the actual OpCode (after all, OpCodes are numbers from 0 to 15), and the value will be a pointer to the corresponding C function.

```
#define NOPS (16) // number of instructions
typedef void (*op_ex_f)(uint16_t instruction);
//
// ... other operations here
//
static inline void add(uint16_t i)  { /* code here */ }
static inline void and(uint16_t i)  { /* code here */ }
//
// ... other operations here
//
op_ex_f op_ex[NOPS] = {
    br, add, ld, st, jsr, and, ldr, str, rti, not, ldi, sti, jmp, res, lea, trap
};
```

`typedef void (*op_ex_f)(uint16_t i);` is a `typedef` for a function pointer, that returns `void` and accepts one parameter the actual `uint16_t instruction`.

All instructions are now accessible if we know the OpCode:

```
uint16_t instr = ...;
op_ex[OP(instr)](instr); // this will execute the function associated with the OP(instr)
                         // For example if OP(instr)==0b0001 then we will execute add(instr)
                         //             if OP(instr)==0b0010 then we will execute ld(instr)
                         //             (and so on)
```

We can avoid writing a `switch` statement with 16 (+1) `cases` with this simple trick.

Now, let see what instructions our VM supports. As I've said earlier, I wasn't original enough to come up with my own ASM, so I've decided to copy the instructions from the LC3 specification.

Instruction	OpCode Hex	OpCode Bin	C function	Comments
br	0x0	0b0000	void br(uint16_t i)	Conditional branch
add	0x1	0b0001	void and(uint16_t i)	Used for addition.
ld	0x2	0b0010	void ld(uint16_t i)	Load RPC + offset
st	0x3	0b0011	void st(uint16_t i)	Store
jsr	0x4	0b0100	void jsr(uint16_t i)	Jump to subroutine
and	0x5	0b0101	void and(uint16_t i)	Bitwise logical AND
ldr	0x6	0b0110	void ldr(uint16_t i)	Load Base+Offset

Instruction	OpCode Hex	OpCode Bin	C function	Comments
str	0x7	0b0111	void str(uint16_t i)	Store base + offset
rti	0x8	0b1000	void rti(uint16_t i)	Return from interrupt (not implemented)
not	0x9	0b1001	void not(uint16_t i)	Bitwise complement
ldi	0xA	0b1010	void ldi(uint16_t i)	Load indirect
sti	0xB	0b1011	void sti(uint16_t i)	Store indirect
jmp	0xC	0b1100	void jmp(uint16_t i)	Jump/Return to subroutine
	0xD	0b1101		Unused OpCode
lea	0xE	0b1110	void lea(uint16_t i)	Load effective address
trap	0xF	0b1111	void trap(uint16_t i)	System trap/call

Regarding their order, instructions can be grouped together (based on their functionality) into 4 main categories:

`trap` is a special instruction that will enable us to interact with the keyboard (read characters or numbers), and print information on `stdout`.

As you notice, our VM won't have a lot of functionality in terms of actual mathematical operations. For example, there's no `XOR`, and no division or multiplication. The good news is that we can implement them as an exercise to learn more about ASM. It's not going to be easy, but it's possible, and it's a good practice to know more about ASM.

But before jumping into the implementation for each instruction, we must note that some operations have additional "side-effects" on the registers.

`RCND`, also known as the conditional register flag, is used to "track" additional information for some instructions. In our implementation it can have only three values:

- `1<<0` (*P* from *positive*) - if the last operation yielded a positive result;
- `1<<1` (*Z* from *zero*) - if the last operation yielded 0;
- `1<<2` (*N* from *negative*) - if the last operation yielded a negative result;

The reason we have `RCND` is to help us with branching. For example, we want to see if a number `a` is bigger than another number `b`. We can calculate their difference, and if the result is a negative number, `RCND` is `1<<2`. Then we can use the `br` instruction to jump to another instruction, just like using an `IF` statement in a high-level programming language.

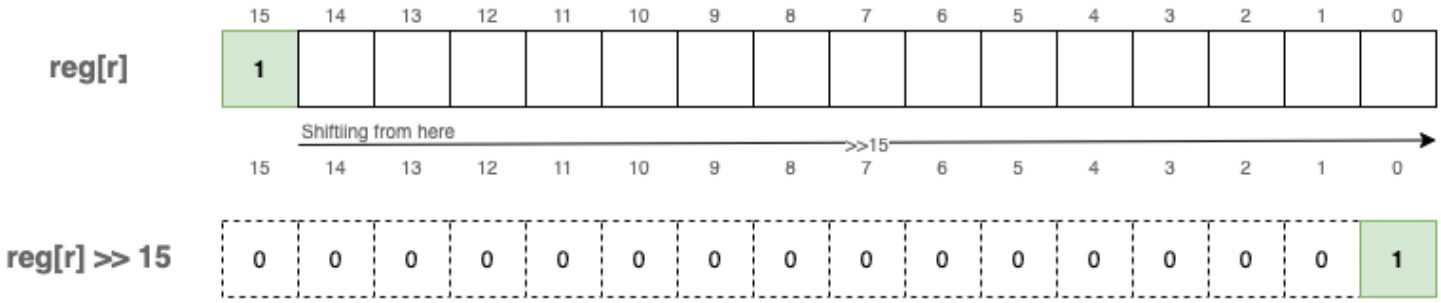
From a code perspective, this can be implemented in C as:

```
enum flags { FP = 1 << 0, FZ = 1 << 1, FN = 1 << 2 };

static inline void uf(enum regist r) {
    if (reg[r]==0) reg[RCND] = FZ;       // the value in r is zero
    else if (reg[r]>>15) reg[RCND] = FN; // the value in r is z negative number
    else reg[RCND] = FP;                 // the value in r is a positive number
}
```

We call `uf(r)`, after each operation that has the “side-effects” we want to test for.

In case you are not familiar with *bitwise operations*, you might wonder what kind of magic is this: `else if (reg[r]>>15)`.

Our VM supports negative numbers (don’t get confused by the fact we’ve used `uint16_t` as the underlying memory type). As a convention, negative numbers have their most significant bit 1 (sitting on position 15).

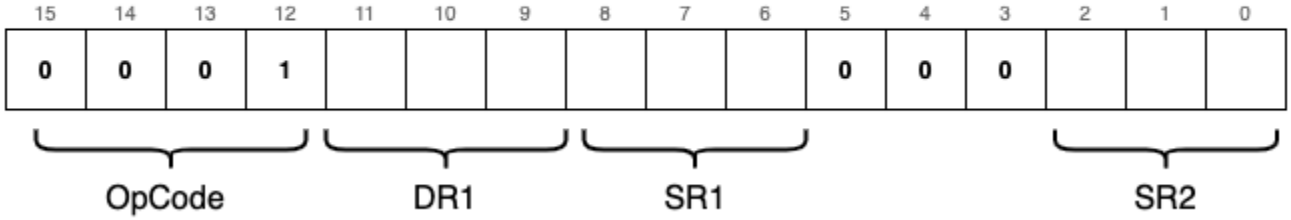


`add` - Adding two values

Having the ability to add two numbers is essential for the VM we are building. In this regard, we will define two `add` instructions. Both of them will have the same OpCode, but the rest of the encoding will be different. `bit[5]` is the one that tells us which version of the `add` instruction we pick.

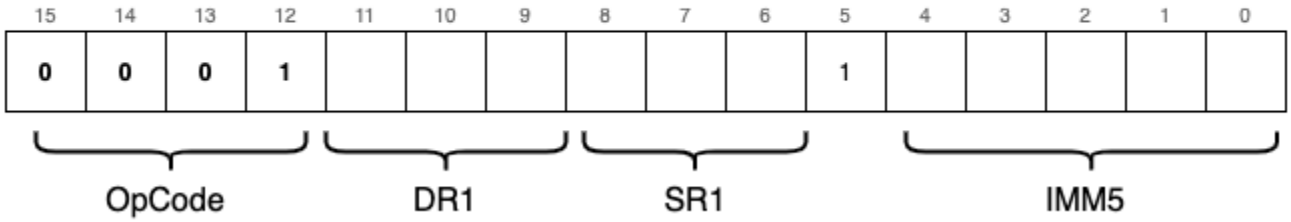
The first one (`add1`) is used for adding the values of two registers: `SR1`, `SR2`, and storing their sum in `DR1`:

add[1]



The second one (`add2`) is used to add a “constant” value (`IMM5`) to `SR1`, and store the result in `DR1`:

add[2]

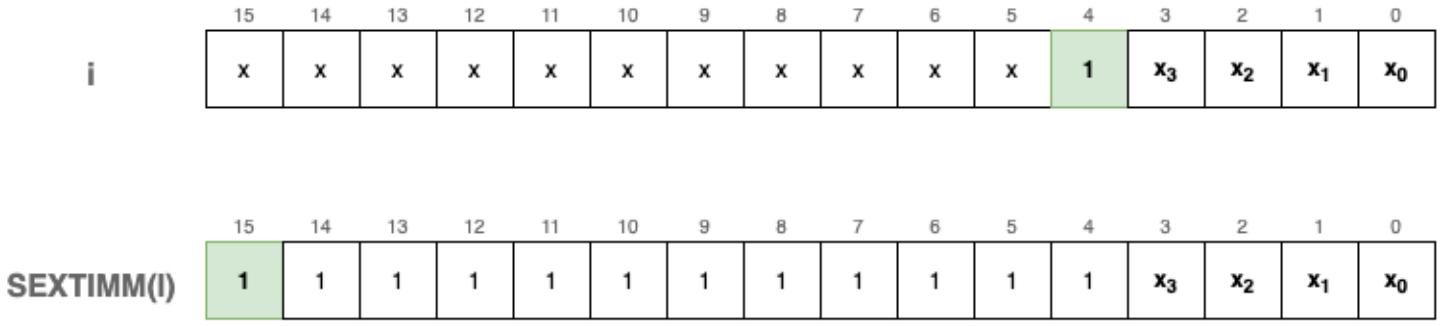


Note: `DR` stands for *Destination Register*. `SR` stands for *Source Register*.

`IMM5` is a positive or negative number on 5 bits. The most significant bit of those 5 is the sign bit. We need to keep this in consideration when we are writing our code. In this regard, we have to write a function that extends the sign to make it compatible with the 16 bits format:

```
#define SEXTIMM(i) sext(IMM(i),5)
static inline uint16_t sext(uint16_t n, int b) {
    return ((n>>(b-1))&1) ?         // if the bth bit of n is 1 (number is negative)
        (n|(0xFFFF << b)) : n;      // fill up with 1s the remaining bits (15)
                                    // else return the number as it is
}
```

Visually, `SEXTIMM(i)` works like this (if the number is negative):



If the number is positive, we don't have to perform any change on it.

For example, running the following code will give you the correct results:

```
uint16_t a = 0x16;          // The 5th bit is 1
                            // This means that the number kept in the last 5 bits
                            // is negative.
                            // So it's important to store it as correctly in a uint16_t type
                            // In this regard we apply SEXTIMM(a) to make it happen

fprintf_binary(stdout, a);
fprintf(stdout, "\n");

fprintf_binary(stdout, SEXTIMM(a));
fprintf(stdout, "\n");

// Output
//
//  0000 0000 0001 0110 <--- a in binary
//  1111 1111 1111 0110 <--- SEXTIMM(a) in binary
```

Now, let's get back to our two `add` (`add1` and `add2`) functions.

From a C perspective, we can group them inside a single method that tests if `bit[5]` is `0` or `1` to decide how to decode further.

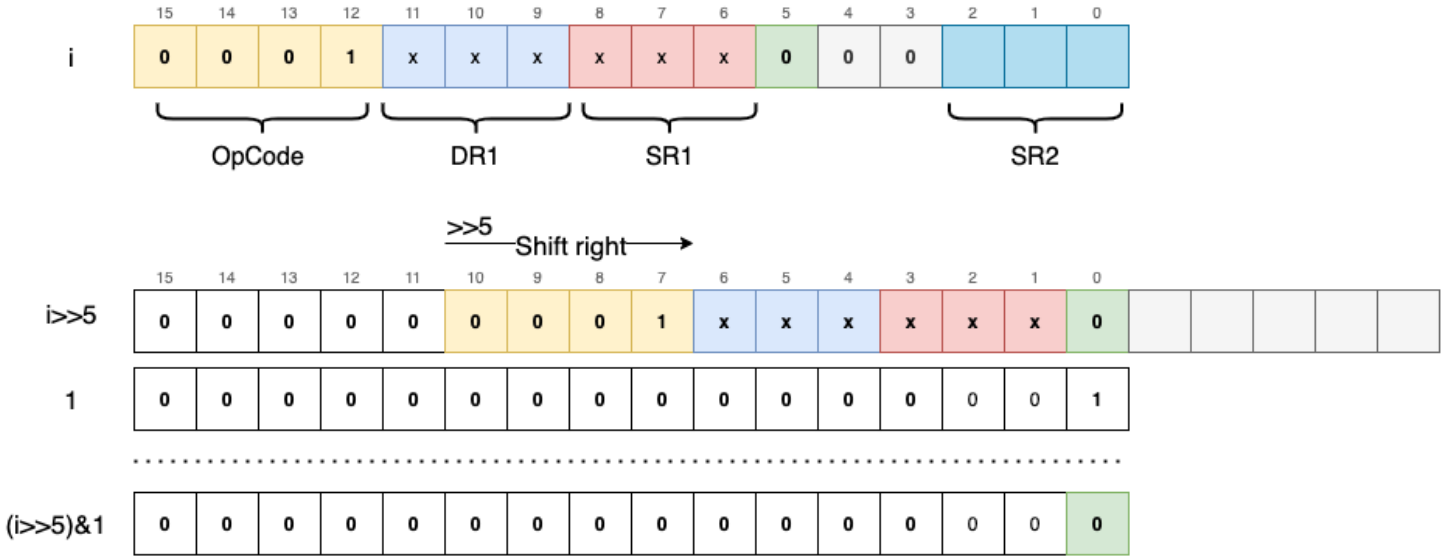
If `bit[5]` is `0`, then we implement `add1`. Otherwise we implement `add2`.

To get the 5th bit of the instruction, we will define a macro:

```
// Gets the 5th bit of i
// We shift to the right with 5 bits so we can have bit on the last position
#define FIMM(i) ((i>>5)&1)
```

From a visual perspective, `FIMM` works like this:

- It shifts the `i` to the right with 5 bits;
- It obtains the last bit by applying `&1`;



Other useful macros we can use to “extract” `SR1`, `SR2`, `DR1` and `IMM5` are:

```
#define DR(i) (((i)>>9)&0x7)
#define SR1(i) (((i)>>6)&0x7)
#define SR2(i) ((i)&0x7)
#define IMM(i) ((i)&0x1F)
```

Explaining each of them it’s not exactly in the scope of this article, but things can become much clearer if you read a good tutorial on bitwise operations. Also, they work quite similar to `FIMM`; we just use a different mask to get the last 3 bits instead of 1.

All in all, our `add` function finally looks like this:

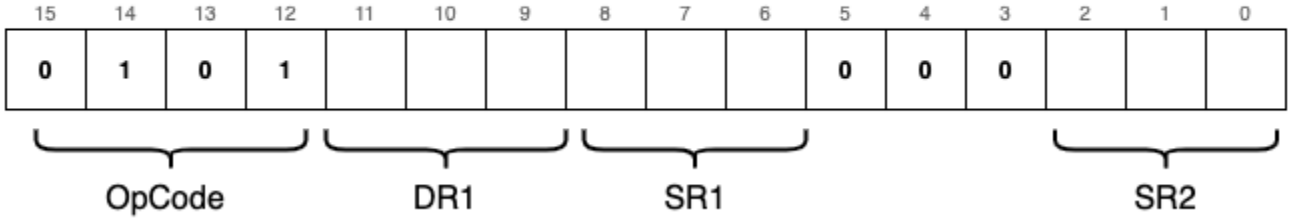
```
static inline void add(uint16_t i)  {
    reg[DR(i)] = reg[SR1(i)] +
        (FIMM(i) ?                  // If the 5th bit is 1
            SEXTIMM(i) :            // we sign extend IMM5 and we add it to SR1 (add2)
            reg[SR2(i)]);           // else we add the value of SR2 to SR1 (add1)

    uf(DR(i)); // !! Update the conditional register depending on the value of DR1
}
```

and - Bitwise logical AND

This instruction is very similar to `add, and it comes in two formats.

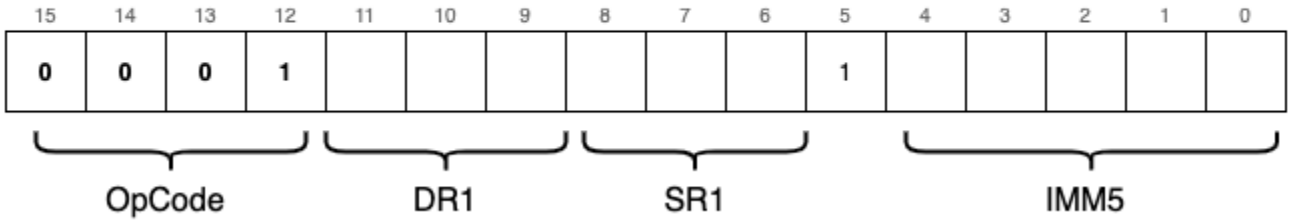
The first one (`add1`) applies binary `&` on the values of two registers: `SR1`, `SR2`, and storing the result in `DR1`:

and[1]



The second one (`add2`) applies binary `&` between `SR1` and `IMM5` and stores the result in `DR1`:

add[2]



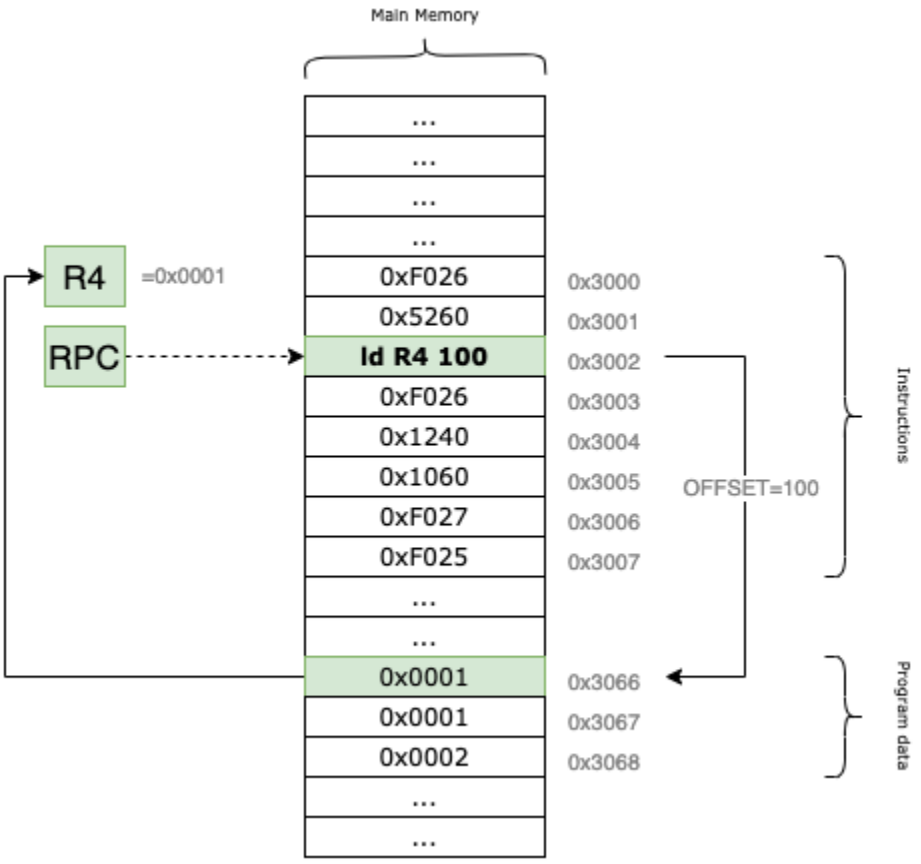
The same idea as before applies; we check `bit[5]` to determine which format we decode.

From a code perspective, the implementation is more or less the same as the previous one, and re-uses the same macros. We just change the operation from `+` to `&`:

```
static inline void and(uint16_t i)  {
    reg[DR(i)] = reg[SR1(i)] &
        (FIMM(i) ?                  // If the 5th bit is 1
            SEXTIMM(i) :            // We sign extend IMM5 and we & it to SR1 (and1)
            reg[SR2(i)]);           // Otherwise we & the value of SR2 to SR1
    uf(DR(i));  // Update the conditional register
}
```

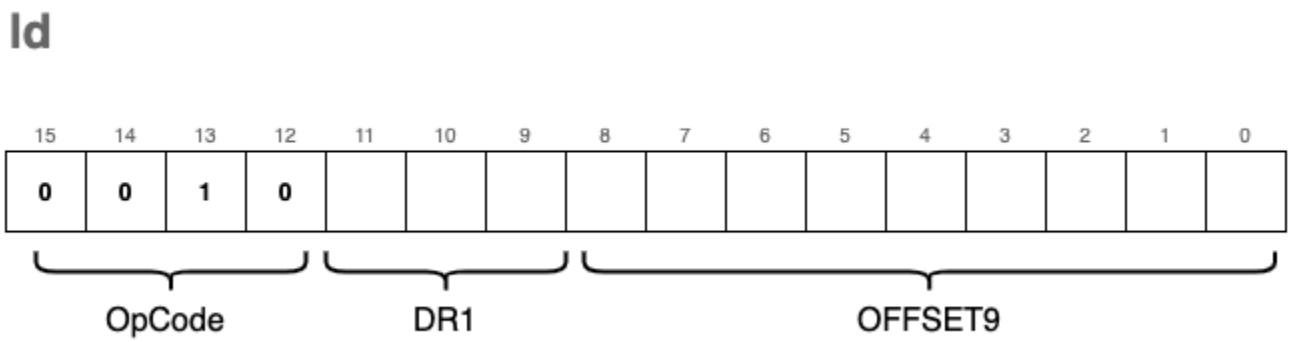
`ld` - Load RPC + offset

`ld` is an instruction to load data from a main memory location to a destination register, `DR1`. The memory location is obtained by adding to the `RPC` register an offset value. Calling `ld` doesn't modify the `RPC`; we use `RPC` just as a referencing point.



So let's say the `RPC` points to the memory address: `0x3002`. If the offset is set to `100` we just read the data from location `0x3002+100==0x3066` and load it in the destination register (in our case `R4`, but this can be everything).

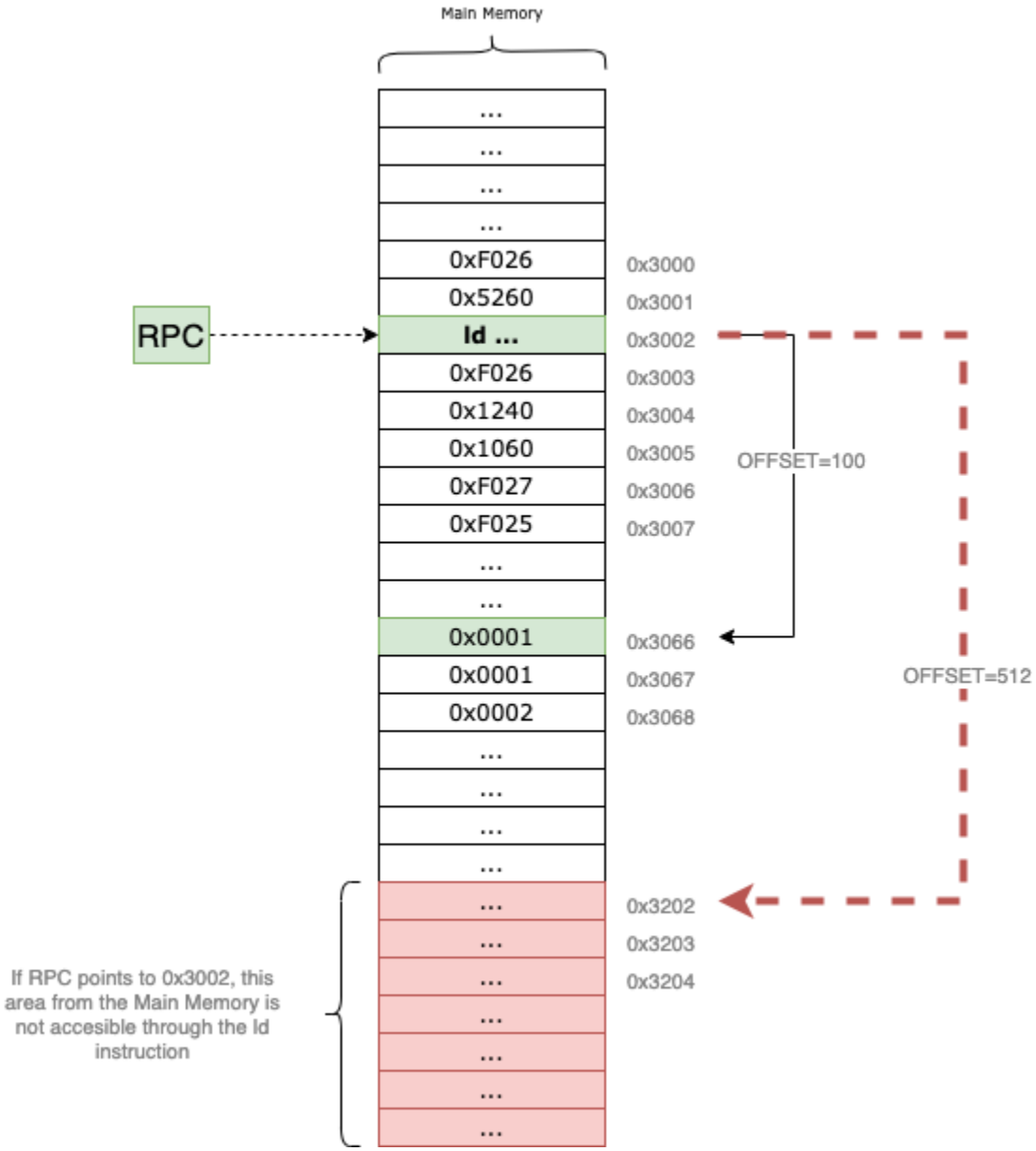
The instruction looks like this:



The associated C code for this functionality is:

```
#define POFF9(i) sext((i)&0x1FF, 9)
static inline void ld(uint16_t i) {
    reg[DR(i)] = mr(reg[RPC] + POFF9(i));
    uf(DR(i));
}
```

The offset is encoded in the last 9 bits of the instruction, this means that the maximum integer value the offset can hold is `2^9-1=512-1=511`. So depending on where (and how) the program is stored, some memory areas will remain inaccessible to the `ld` instruction.

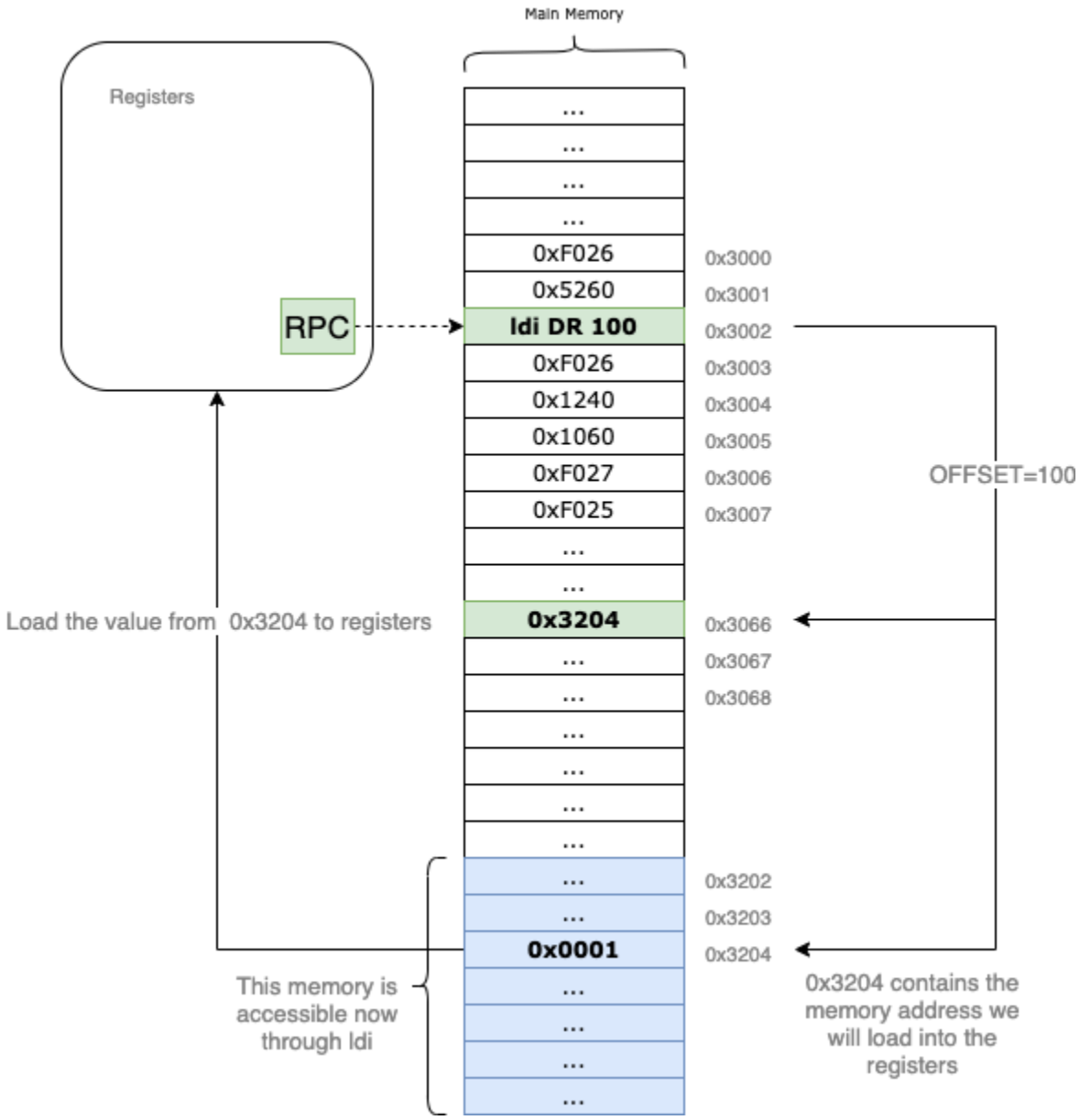


`512` in binary is `0b1000000000`. As you can see it needs 10 bits for the representation. Thus, the maximum value the offset can have is `511`, which in binary can be written as `0b111111111` (9 bits).

To solve this limitation `ld` has, let's look at another instruction for loading memory into registers called `ldi`.

`ldi` - Load indirect

This instruction is used to load data into registers, using intermediary addresses to access "far-away" memory areas.



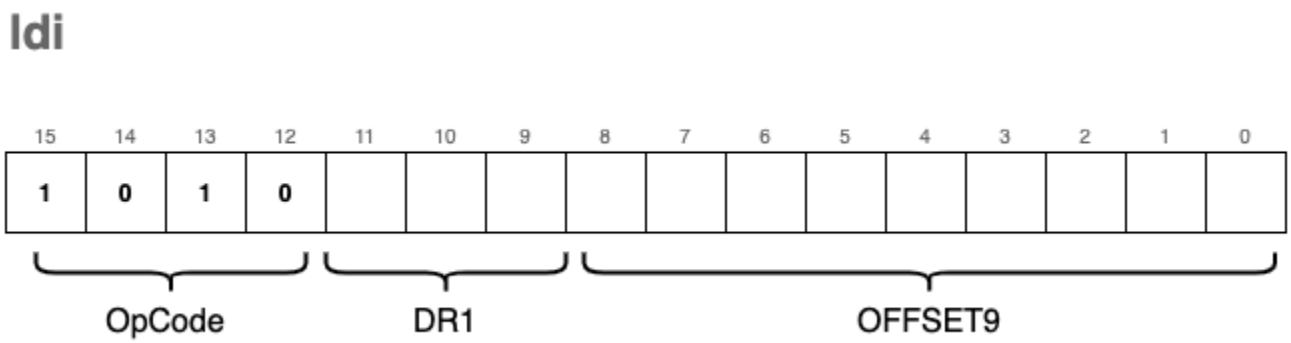
So let's say the `RPC` points to `0x3002`. Just like before, we use a 9 bit offset to access another memory address at position (`RPC+offset`). In our case, the `offset=100`, so the memory address we read is `0x3066`.

But instead of loading (directly) `0x3066` into `DR`, we look at the value contained by `0x3066`, which is `0x3204`. We now, bring the value of `0x3204` inside the `DR`.

Using `ldi` eliminates the issue with the offset's 9-bit limitation that *affects* `ld`.

As a side-note, this doesn't mean `ldi` is better than `ld`, because instead of performing one read, we have to perform two. It just serves another purpose.

The format of the instruction is almost identical to `ld`, only the OpCode is different:



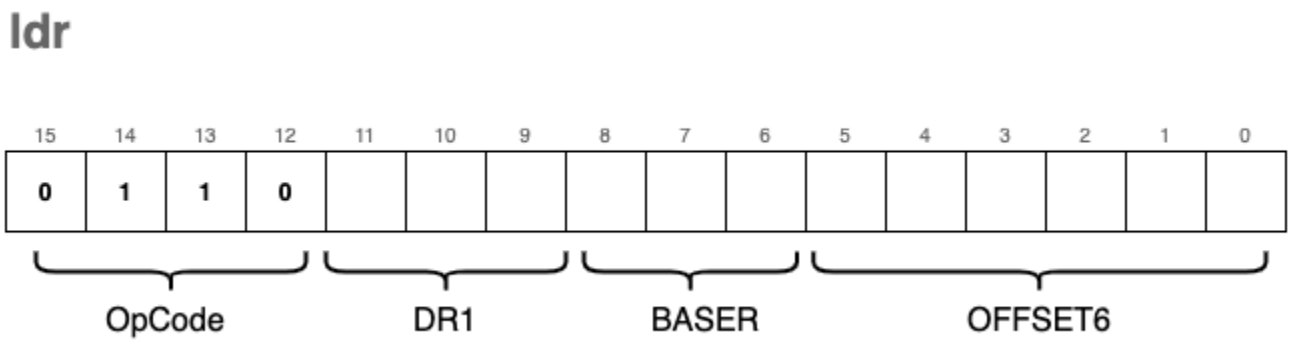
The C code looks like this:

```
static inline void ldi(uint16_t i)  {
    reg[DR(i)] = mr(mr(reg[RPC]+POFF9(i))); // We perform two memory reads
    uf(DR(i));
}
```

`ldr` - Load Base+Offset

This is another instruction we use to load data into registers, but compared to `ld` where we start from `RPC`, this time we can use a different `base` (by base, we mean a memory address kept in a register).

The format of the instruction is as follows:



To extract `BASER` from the instruction, we can re-use the macro we've firstly defined for `ld` (`SR1(i)`), because the bits have the same position.

The extract `OFFSET6` from the instruction we will use the following macro:

```
#define POFF(i) sext((i)&0x3F, 6)
```

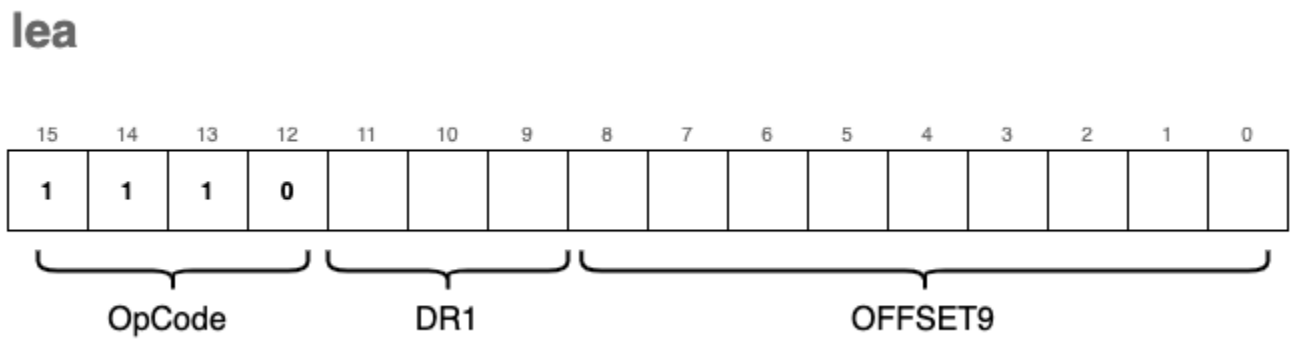
The code is also very similar to `ld`, with just one small change. Instead of using `reg[RPC]`, we offset from `BASER`.

```
static inline void ld(uint16_t i) {
    reg[DR(i)] = mr(reg[RPC] + POFF9(i));
    uf(DR(i));
}

// VERSUS

static inline void ldr(uint16_t i) {
    reg[DR(i)] = mr(reg[SR1(i)] + POFF(i));
    uf(DR(i));
}
```

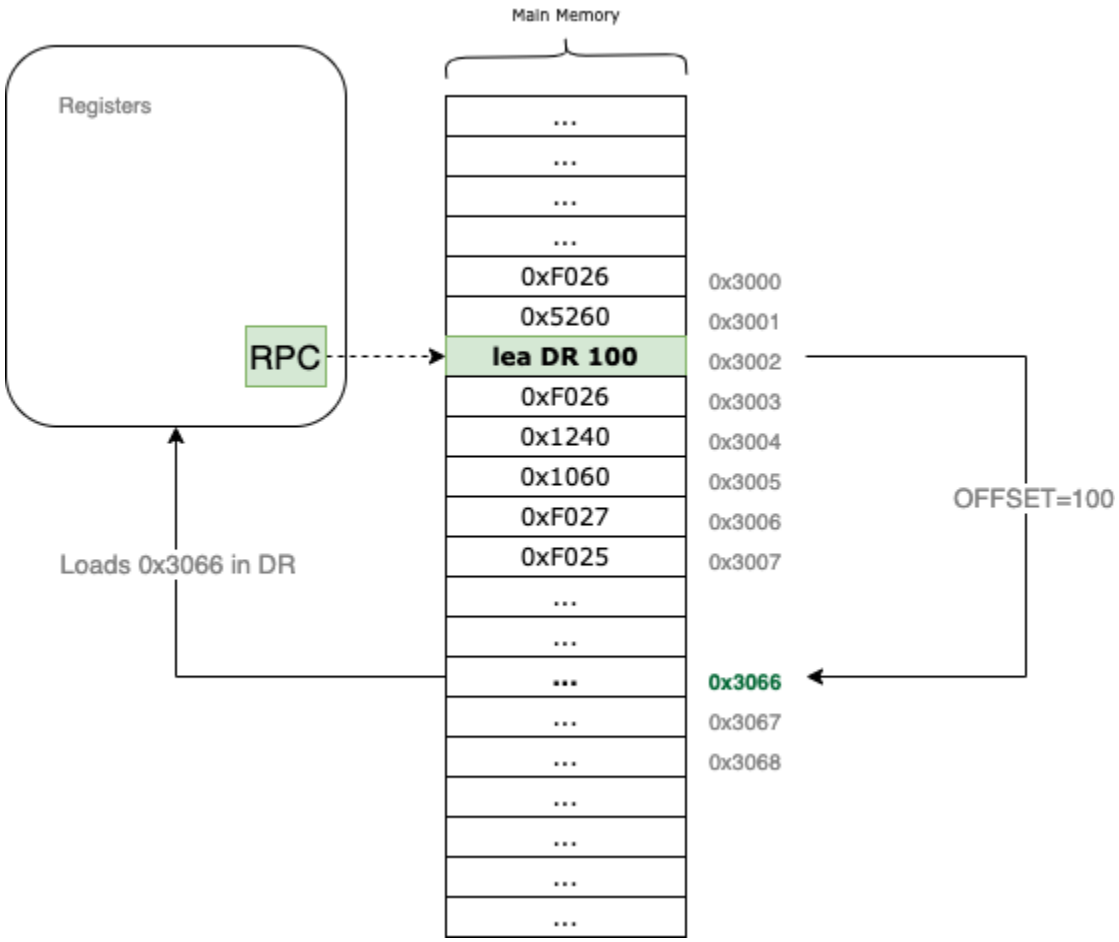
`lea` - Load effective address

This instruction helps us load memory addresses into registers. Compared to `ld`, `ldi` and `ldr`, `lea` doesn't bring program data into registers, but memory locations.

The format of `lea` is:



From a visual perspective, the instruction works like this:



Let's say the `RPC` points to `0x3002`. The `offset9=100`, so we load into the `DR` register the result of `0x3002+100=0x3066`.

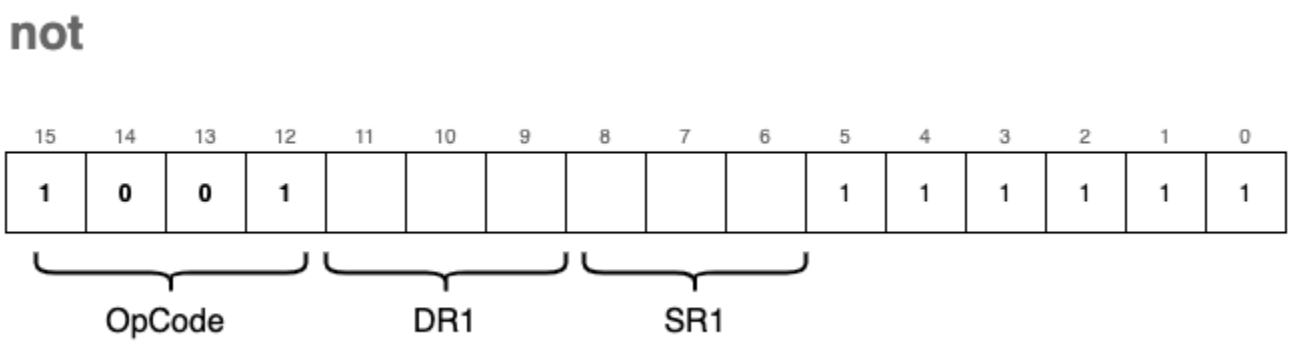
From a code perspective things are look like this:

```
static inline void lea(uint16_t i) {
    reg[DR(i)] =reg[RPC] + POFF9(i);
    uf(DR(i));
}
```

`not` - Bitwise complement

This instruction simply performs a bitwise complement `~` on `SR1` and stores the value in `DR1`.

The format of the instruction is:



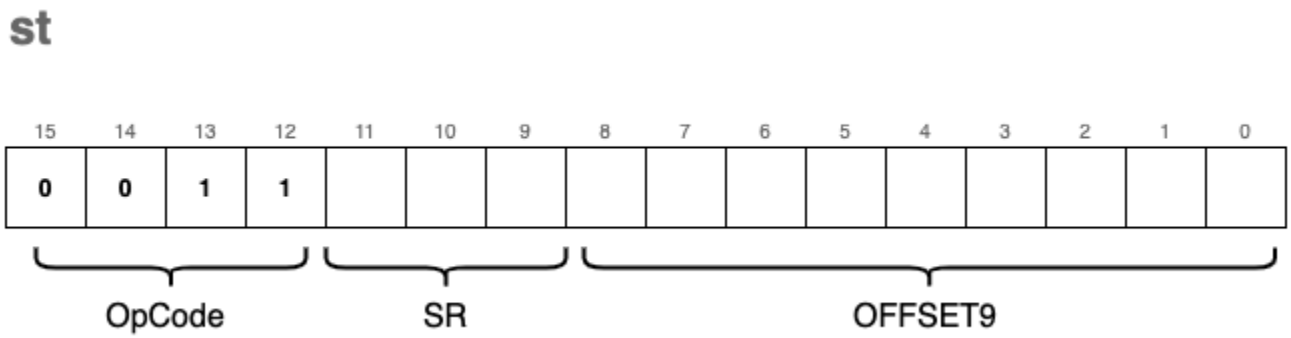
The corresponding C code is:

```
static inline void not(uint16_t i) {
    reg[DR(i)]=~reg[SR1(i)];
    uf(DR(i));
}
```

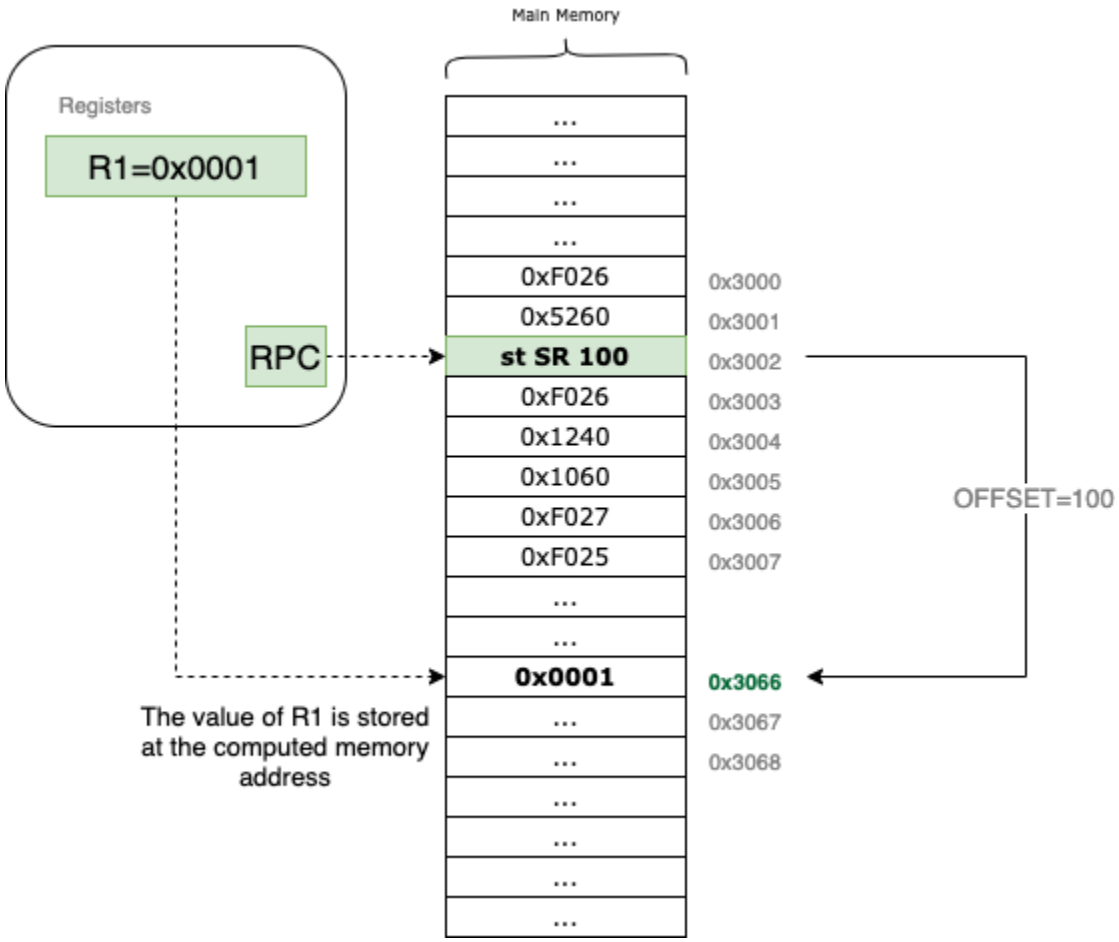
`st` - Store

We use the `st` instruction to store the value of a given register to a memory location.

The format of the instruction is:



Visually this instruction works like this:



Let's say `RPC` is pointing to `0x3002`, and `SR` refers to `R1=0x0001`. The `st` instruction will write the value of `R1` to `RPC+offset`.

The C code is straightforward:

```
static inline void st(uint16_t i)  {
    mw(reg[RPC] + POFF9(i), reg[DR(i)]); // writes the value of DR(i) to the memory address
}
```

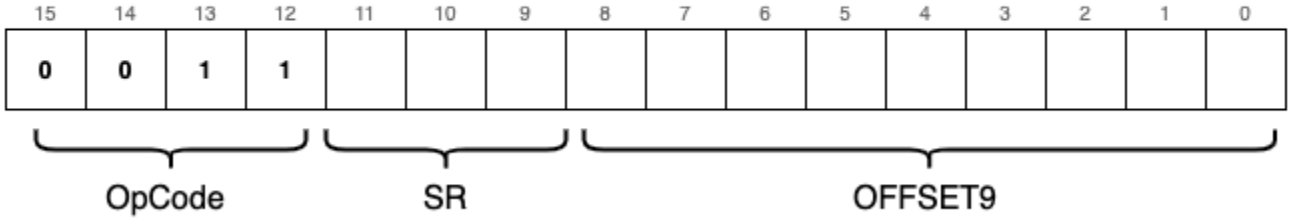
There's no need to update any flags because we don't do any computation on our registers.

Similar to `ld` before, `st` suffers from the same memory *addressability*limitation. In this regard, we introduce a new instruction called `sti`.

`sti` - Store indirect

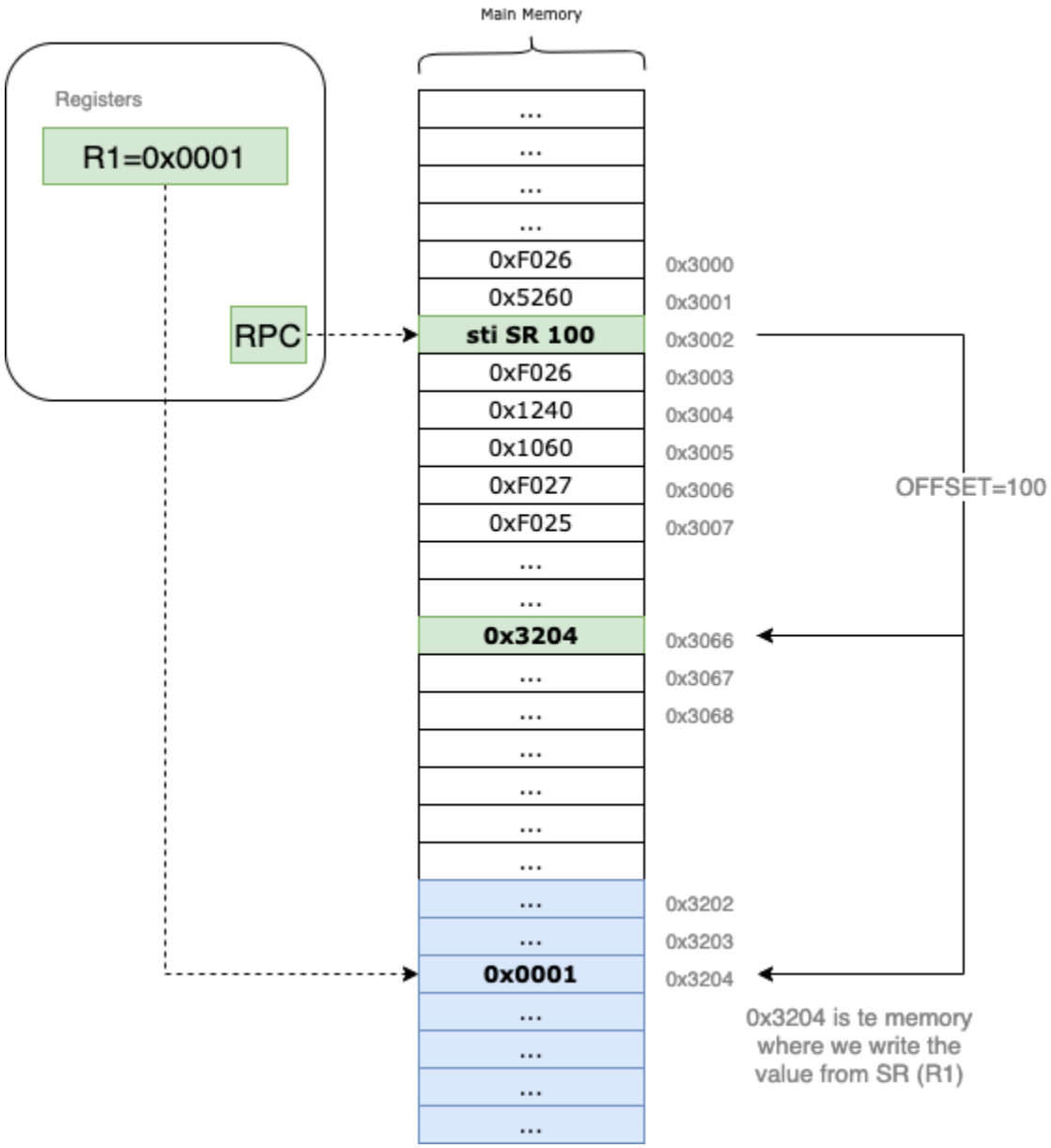
The format of the instruction is similar to `st`, only the OpCode changes:

sti



The behavior is somewhat different.

Instead of writing to the memory address directly, we use an intermediary address from the main memory to intermediate the write (`mw(..)`). This secondary address contains the actual memory location where we will write.



To write the content of `R1=0x0001` (`SR`), to `0x3204`, we will first have to read the memory location `RPC+offset = 0x3204`. Here we will find the value of the address we wish to write to: `3204`.

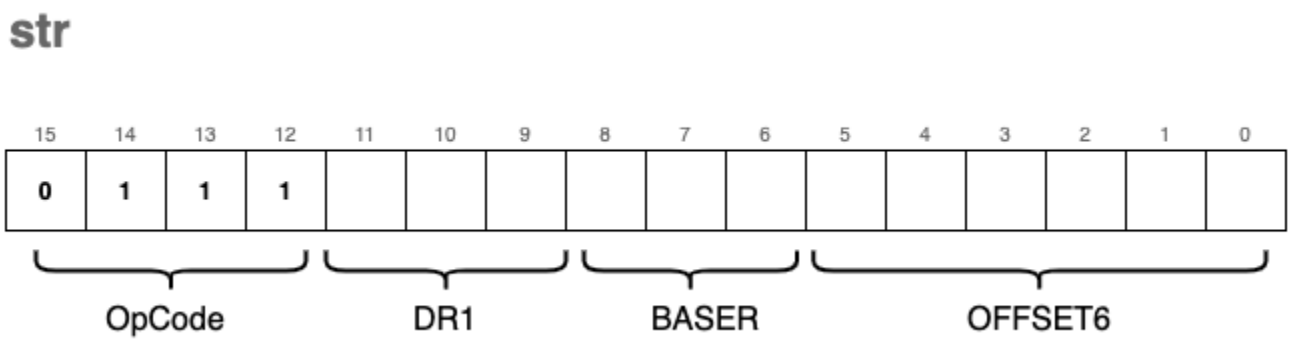
From a code perspective, the implementation in C of `sti` looks like:

```
static inline void sti(uint16_t i)  {
    mw(mr(reg[RPC] + POFF9(i)), reg[DR(i)]);
}
```

`str` - Store base + offset

This instruction is similar `st`, with one difference. Instead of starting from `RPC` we can specify another base as the reference register (`BASER`), to which we add the offset (`OFFSET6`).

The format of the instruction is the following:



The C code:

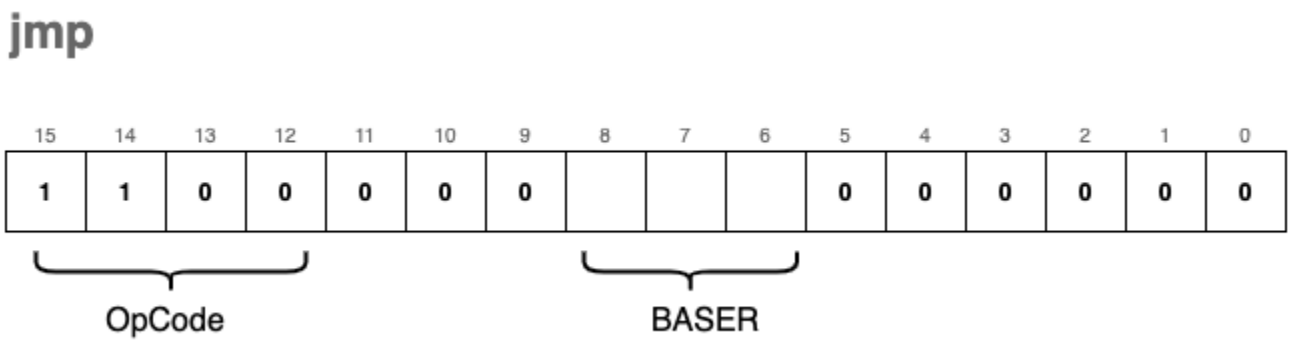
```
static inline void st(uint16_t i)   {
    mw(reg[RPC] + POFF9(i), reg[DR(i)]);
}

// VERSUS

static inline void str(uint16_t i)  {
    mw(reg[SR1(i)] + POFF(i), reg[DR(i)]);
}
```

`jmp` - Jump

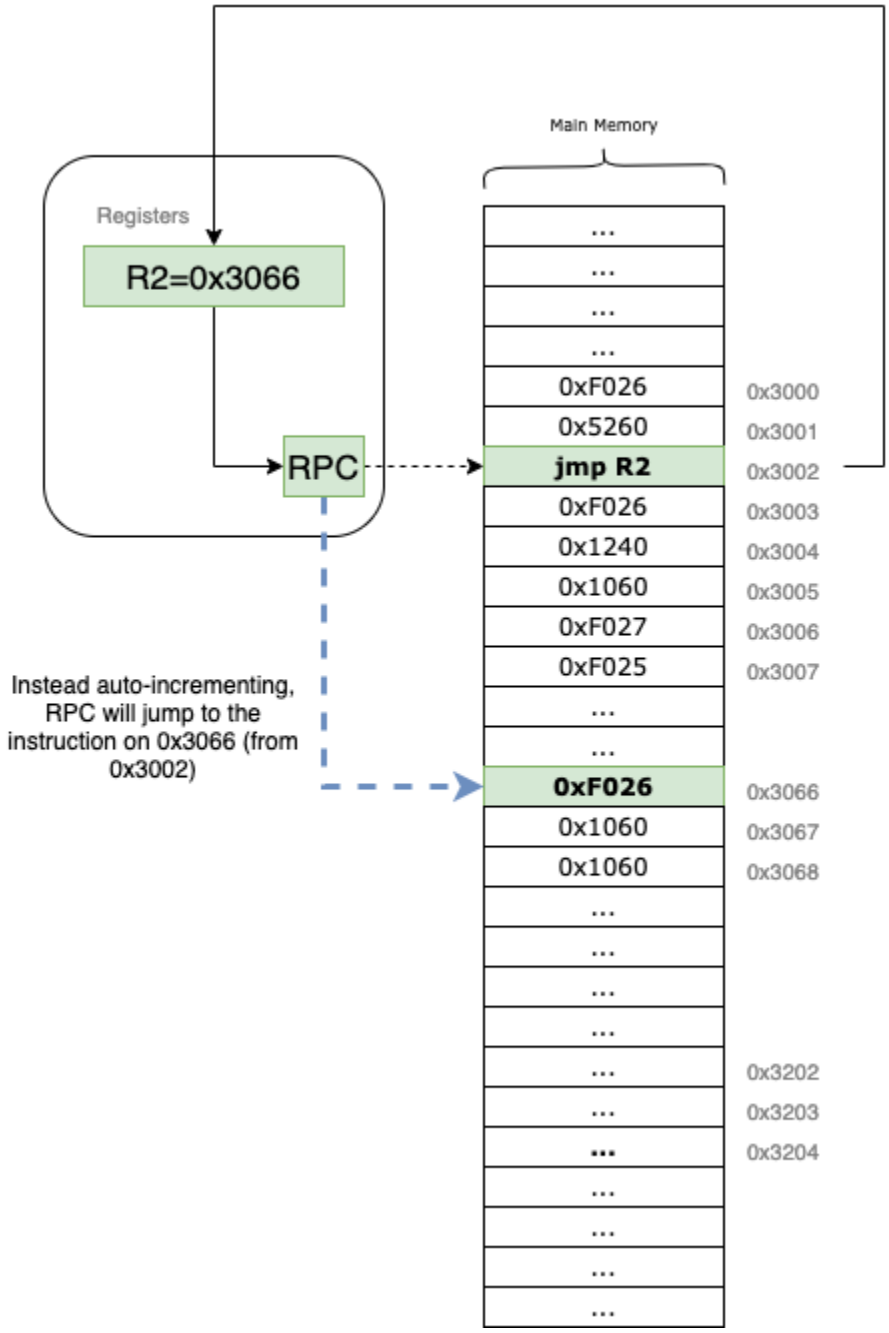
Typically, the `RPC` register auto-increment after each instruction gets executed.

`jmp` is an instruction that makes our `RPC` jump to the location specified by the contents of the `BASER` (base register).

The format is the following:



And visually the instruction works like this:



Let's say our `RPC=0x3002`, and `R2=0x3066` (this is `BASER`). When the `jmp` instruction is encountered, `RPC` will jump directly to the memory address kept in `BASER` (`R2=0x3066`), and the program flow will continue.

In some high-level programming languages, `jmp` is similar to a `go to` statement.

The corresponding `C` code is:

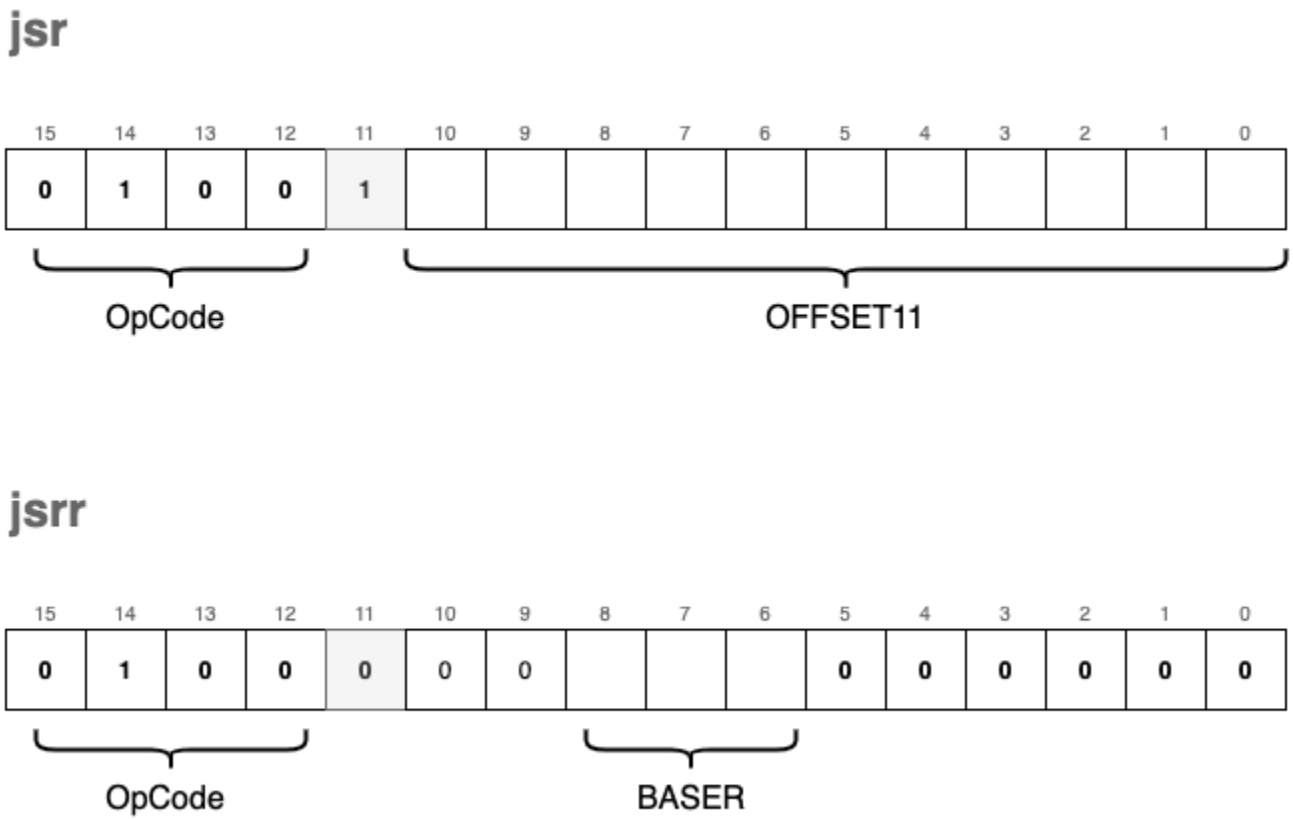
```
static inline void jmp(uint16_t i)  {
    reg[RPC] = reg[BR(i)];
}
```

`jsr` - Jump to subroutines

`jsr` is a control flow instruction that helps us implement *subroutines*.

Subroutines in (our) ASM are similar to functions from a high-level programming language. They can be viewed as a series of instructions and their starting. They have an input (expect to read data from registers), and an output (they put the return value to a register).

`jsr` comes into two formats:



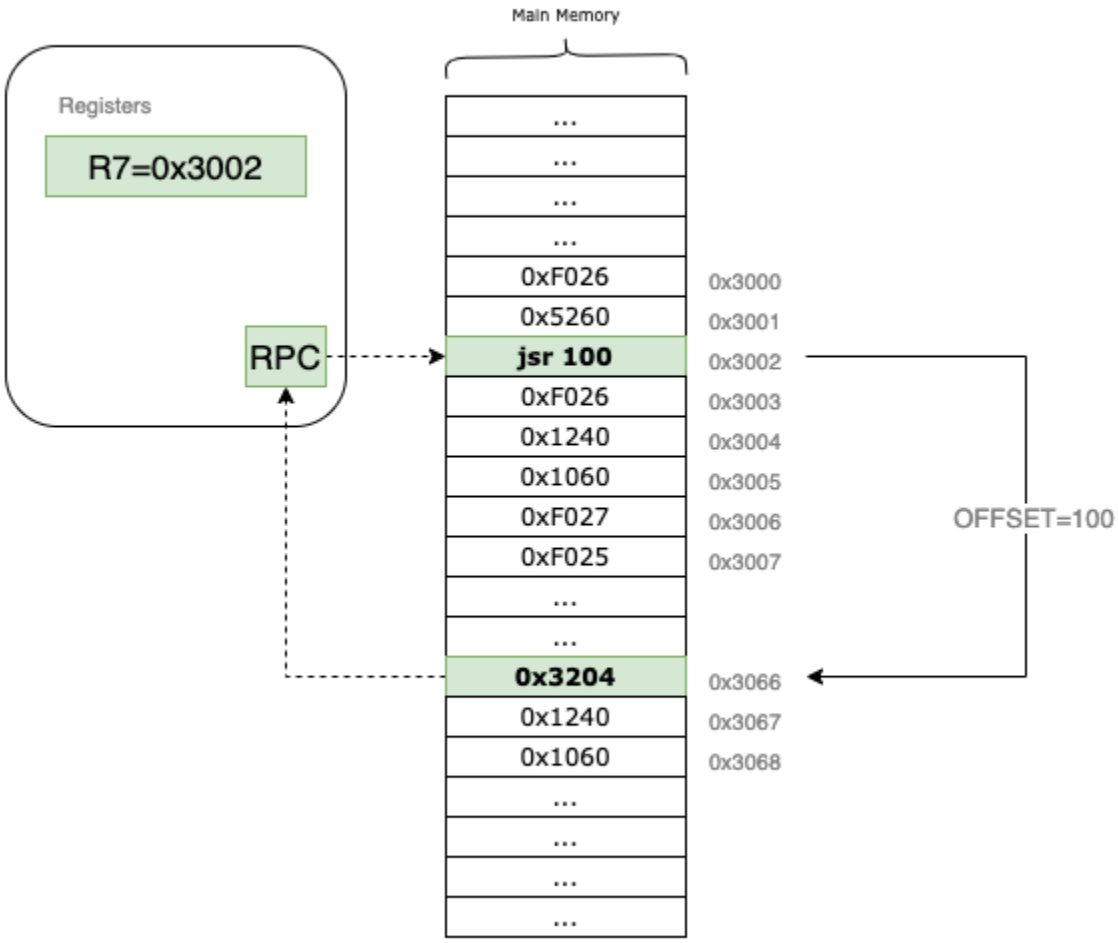
The pseudo-code for the operation is as follows:

1. We save the `RPC` in `R7` (to *remember* from where we branch);
2. If `bit[11]` is set to `0` then we set the `RPC = BASER`;
3. If `bit[11]` is set to `1` then set `RPC = RPC + OFFSET11`;

The associated C code is the following:

```
static inline void jsr(uint16_t i) {
    reg[R7] = reg[RPC];
    reg[RPC] = (FL(i)) ?            // Checks bit[11]
        reg[RPC] + POFF11(i) :      // rpc + offset
        reg[BR(i)];                     // base register
}
```

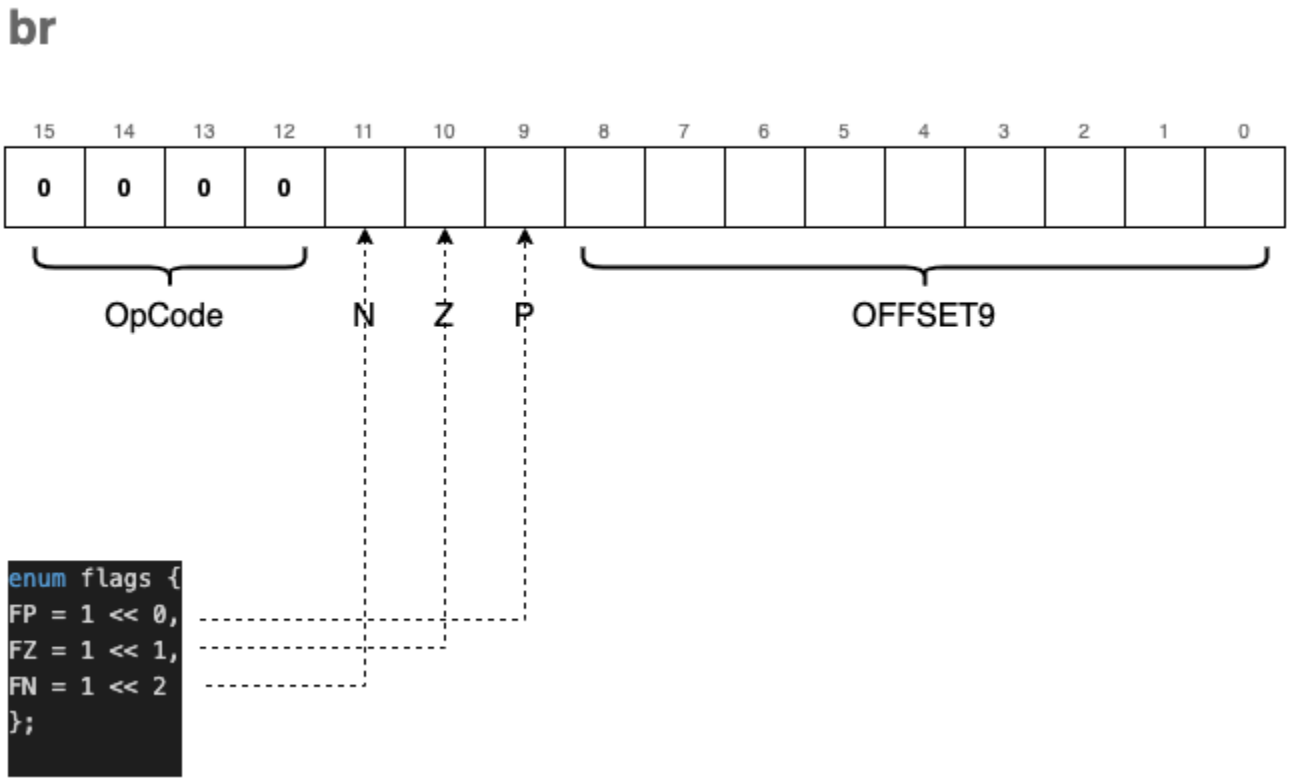
The visual explanation for this instruction looks like this:



In the above example, `RPC` is initially set to `3002`. At this position, it's a `jsr` instruction. We store `RPC` in `R7` to remember from where we branched off. Then we jump with the required `offset=100`, to position `0x3066`, and we update `RPC` to this.

`br` - Conditional branch

This instruction works similar to `jsr`, but there's one big difference, the branching happens only if some conditions are met.



If you look closer, the NZP bits are not *accidentally* named like this. They are used to reflect the changes we are making with `uf(...)` on `RCND`:

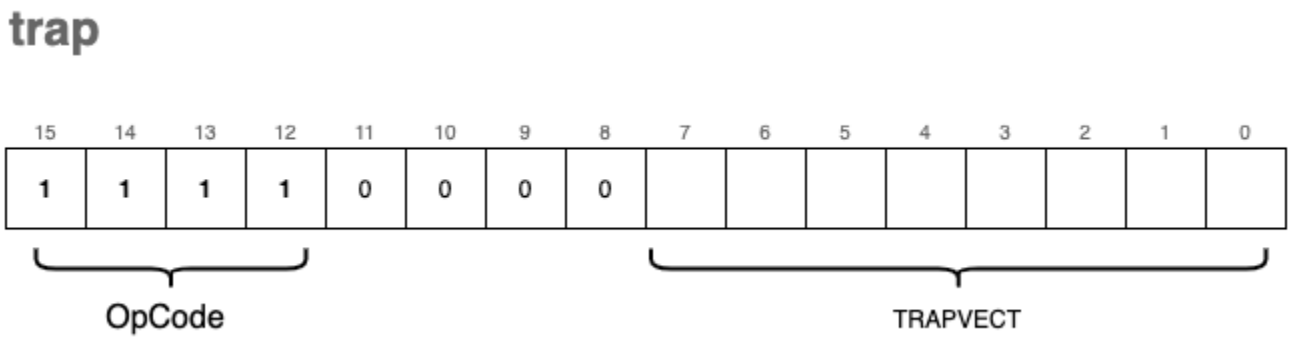
```
enum flags { FP = 1 << 0, FZ = 1 << 1, FN = 1 << 2 };
static inline void uf(enum regist r) {
    if (reg[r]==0) reg[RCND] = FZ;       // the value in r is zero
    else if (reg[r]>>15) reg[RCND] = FN; // the value in r is z negative number
    else reg[RCND] = FP;                 // the value in r is a positive number
}
```

Basically all the possible values of `RCND` are `FP=001`, `FZ=010` or `FN=100`. When implementing our C code for `br` we can easily take this into consideration, and compare the NZP segment of the instruction with `RCND`:

```
#define FCND(i) (((i)>>9)&0x7)
static inline void br(uint16_t i)   {
    if (reg[RCND] & FCND(i)) {          // if the condition is met
        reg[RPC] += POFF9(i);           // branch to offset
    }
}
```

`trap`

This instruction is more complex than the rest because it enables us to interact with I/0 and, theoretically, other devices.



In the `TRAPVECT` we keep an index to other functions that offer us the possibility to read numbers and strings from `stdin` and write them on `stdout`.

Each trap will be kept in an array `trp_ex` that contains pointers to the associated C functions. We follow the same strategy as we did with the instructions, but we will use a separate array.

If we would've kept things *kosher*, traps should've been implemented in ASM, and we would've implemented another mechanism to interact with the keyboard and `stdout`.

Anyway, this is how the code is structured:

```
#define TRP(i) ((i)&0xFF)

static inline void tgetc()  { /* code */ }
static inline void tout()   { /* code */ }
static inline void tputs()  { /* code */ }
static inline void tin()    { /* code */ }
static inline void tputsp() { /* code */ }
static inline void thalt()  { /* code */ }
static inline void tinu16() { /* code */ }
static inline void toutu16() { /* code */ }

enum { trp_offset = 0x20 };
typedef void (*trp_ex_f)();
trp_ex_f trp_ex[8] = { tgetc, tout, tputs, tin, tputsp, thalt, tinu16, toutu16 };

static inline void trap(uint16_t i) {
    trp_ex[TRP(i)-trp_offset]();
}
```

In total, there are 8 supported trap functions:

Trap Function	TRAPVECT	`trp_ex[]` index	Comments
`tgetc`	`0x20`	0	Reads a character (`char`) from the keyboard that get copied in `R0`

Trap Function	TRAPVECT	trp_ex[] index	Comments
tout	0x21	1	Writes the character (char) from R0 to the console.
tputs	0x22	2	Writes a string of characters to the console. As a rule, the characters are kept in a contiguous memory location, one char per memory location. Starting with the address specified in R0. If 0x0000 is encountered, printing stops.
tin	0x23	3	Reads a character (char) from the keyboard, and it gets copied in R0. Afterward, the char is printed on the console.
tputsp	0x24	4	Not implemented. This trap is used to store 2 characters per memory location instead of 1. Otherwise, it works like tputs. It's left out as an exercise.
thalt	0x25	5	Halts execution of the program. The VM stops.
tinu16	0x26	6	Reads a uint16_t from the keyboard and stores it in R0.
toutu16	0x27	7	Writes the uint16_t found inside R0.

Now, let's see how each method is implemented.

tgetc

We simply use getchar() and stores the returning character to R0:

```
static inline void tgetc() { reg[R0] = getchar(); }
```

toutc

We print the R0 to stdout:

```
static inline void tout() { fprintf(stdout, "%c", (char)reg[R0]); }
```

tputs

We iterate through the memory location starting at R0 until we find 0x0000 and print each character in order:

```
static inline void tputs() {
    uint16_t *p = mem + reg[R0];
    while(*p) {
        fprintf(stdout, "%c", (char)*p);
        p++;
    }
}
```

tin

This method is almost identical to `tgetc`, with one small difference, we print the char after storing it inside `R0`:

```
static inline void tin() { reg[R0] = getchar(); fprintf(stdout, "%c", reg[R0]); }
```

thalt

To keep track of the running VM, we keep a global `bool` variable called `running`. Once `thalt` is invoked, `running` is set to `false`, and the machine automatically stops:

```
static inline void thalt() { running = false; }
```

tinu16

We read the `uint16_t` from keyboard and we store it inside `R0`:

```
static inline void tinu16() { fscanf(stdin, "%hu", &reg[R0]); }
```

toutu16

We write the `uint16_t` content of `R0` to the console:

```
static inline void toutu16() { fprintf(stdout, "%hu\n", reg[R0]); }
```

Loading and running programs

Congratulations, if you kept coding while you were reading this article, at this point you have a working toy-VM, capable of running simple programs written in our ASM language.

We have only two missing functionalities: the main loop and the ability to load programs.

The main loop of our VM looks like this:

```
bool running=true;
uint16_t PC_START = 0x3000;
void start(uint16_t offset) {
    reg[RPC] = PC_START + offset; // The RPC is set
    while(running) {
        uint16_t i = mr(reg[RPC]++); // We extract instructions from the memory
                                     // location pointed by RPC
                                     // We (auto)increment RPC
        op_ex[OPC(i)](i);            // We execute each instruction
    }
}
```

Now, the only thing left missing is the ability to load programs into our VM in this regard we will write a `ld_img` method capable of loading binary files directly into our main memory:

```
void ld_img(char *fname, uint16_t offset) {
    // Open (binary) file containing the VM program
    FILE *in = fopen(fname, "rb");
    if (NULL==in) {
        fprintf(stderr, "Cannot open file %s.\n", fname);
        exit(1);
    }
    // The position from were we start copying the file
    // to the main memory
    uint16_t *p = mem + PC_START + offset;
    // Load the program in memory
    fread(p, sizeof(uint16_t), (UINT16_MAX-PC_START), in);
    // Close the file stream
    fclose(in);
}
```

This method returns `void` and accepts two input parameters:

- The path to the binary file containing our program;
- The offset from where we start loading the first program instruction into the main memory.

The main method of our VM looks like this:

```
int main(int argc, char **argv) {
    ld_img(argv[1], 0x0);
    start(0x0);
    return 0;
}
```

Our first program

Our first program will not be `"Hello, world!"`, but something more exciting: a complex software that reads two numbers from the keyboard and prints their sum to `stdout`.

Jokes aside, here how it looks:

```
0xF026    //  1111 0000 0010 0110  TRAP tinu16      ;read an uint16_t in R0
0x1220    //  0001 0010 0010 0000  ADD R1,R0,x0     ;add contents of R0 to R1
0xF026    //  1111 0000 0010 0110  TRAP tinu16      ;read an uint16_t in R0
0x1240    //  0001 0010 0010 0000  ADD R1,R1,R0     ;add contents of R0 to R1
0x1060    //  0001 0000 0110 0000  ADD R0,R1,x0     ;add contents of R1 to R0
0xF027    //  1111 0000 0010 0111  TRAP toutu16     ;show the contents of R0 to stdout
0xF025    //  1111 0000 0010 0101  HALT             ;halt
```

The *syntax* is not user friendly, isn't it ? Our program is actually this series of numbers: `0xF026 0x1220 0xF026 0x1240 0x1060 0xF027 0xF025`. But if we look closer, in those numbers we've been encoding ASM instructions.

For example, let's look at this number `0xF026`. Its binary representation is `1111 0000 0010 0110`. It is easy to spot `1111` as the encoding for `trap`, and the `TRAPVECT`, `100111`, corresponding to `tinu16`.

Or for a more visual representation, let's analyze `0x1220`:

```
0x1220 ->

0001   001 000 1  00000
ADD    R1  R0     IMM5=0
```

Running our first program

Bad news, there's no compiler. We will have to write programs by hand, with pen and paper (like the people in the old days).

The good news is we don't need a *compiler* to generate a binary file that our VM can run. We can write a C program for that.

The main idea is to keep our instructions in an array (`uint16_t program[]`), and then use `fwrite()` to generate the binary file, which we can further load with `ld_img()`.

```
#include <stdio.h>
#include <stdlib.h>

uint16_t program[] = {
    /*mem[0x3000]=*/    0xF026,    //  1111 0000 0010 0110            TRAP trp_in_u16  ;read an
uint16_t from stdin and put it in R0
    /*mem[0x3002]=*/    0x1220,    //  0001 0010 0010 0000            ADD R1,R0,x0     ;add
contents of R0 to R1
    /*mem[0x3003]=*/    0xF026,    //  1111 0000 0010 0110            TRAP trp_in_u16  ;read an
uint16_t from stdin and put it in R0
    /*mem[0x3004]=*/    0x1240,    //  0001 0010 0010 0000            ADD R1,R1,R0     ;add
contents of R0 to R1
    /*mem[0x3006]=*/    0x1060,    //  0001 0000 0110 0000            ADD R0,R1,x0     ;add
contents of R1 to R0
    /*mem[0x3007]=*/    0xF027,    //  1111 0000 0010 0111            TRAP trp_out_u16;show the
contents of R0 to stdout
    /*mem[0x3006]=*/    0xF025,    //  1111 0000 0010 0101            HALT             ;halt
};

int main(int argc, char** argv) {
    char *outf = "sum.obj";
    FILE *f = fopen(outf, "wb");
    if (NULL==f) {
        fprintf(stderr, "Cannot write to file %s\n", outf);
    }
    size_t writ = fwrite(program, sizeof(uint16_t), sizeof(program), f);
    fprintf(stdout, "Written size_t=%lu to file %s\n", writ, outf);
    fclose(f);
    return 0;
}
```

If we compile and run the above program, a binary file called `"sum.obj"` will be generated.

Then we can use our VM to load it and run it:

The current git repo contains another program: `simple_program.c` which sums up the numbers in an array. If you are curious about it, you can run it yourself.

Final thoughts

First of all, thank you for reading up until this point, and congratulations on the fact you have a running VM.

Writing a simple toy like this is simple, but creating something remotely useful is hard work and involves much more than we've covered up until this point.

Maybe in another article, we will implement a Stack-Based VM, or a *modern* hybrid between the two.

After submitting this article on various communities (btw, feedback was amazing), people constructively suggested the naming conventions are confusing: functions names are too short, code is too terse, etc.

I wouldn't usually name things like I did here, but given the fact that the program, in essence, is short, contained, and geared towards the low-level world of ASM (Where things have *non-modern* naming conventions), it was more natural for me to write the code like this. On short, the code was not imagined to be easily understood without this companion article aside.
