ANC216 Assembly

The assembly standard for ANC216

Simone Ancona

Introduction

The ANC216 assembly language is a low-level programming language designed for the ANC216 architecture. Programmers who wants to learn this new language must know the target architecture, it is recommended to read ANC216 architecture manual.

Definitions

This section contains all the definitions and abbreviations that can be found in the article.

- BP: Base Pointer.
- Bus: a computer communication system used to connect components and peripherals.
- Byte: 8 bits.
- CPU: Central Processing Unit, is the main processor in a computer.
- EINR: External Interrupt.
- EMEM: External Memory.
- GPR: General Purpose Register.
- HEX: Hexadecimal.
- IMEM: Internal Memory.
- INR: Interrupt.
- ISA: Instruction Set Architecture.
- IO: Input/Output.
- MTU: Memory Table Unit.
- NMI: non-maskable interrupt.
- OPC: Operation Code.
- PC: Program Counter.
- RAM: Random Access Memory.
- ROM: Read Only Memory.
- SP: Stack Pointer.
- Word: 16 bits.

Assembly

Assembly language is a low-level programming language that provides symbols representing computer instructions. It is the human-readable format of CPU language. It is importante to note that assembly is not a standard language and varies based on the architecture and the assembler used. The assembler is the program that reads the source code and translate it into machine code.

In this article we will see the ANC216 assembly.

Instructions

An instruction is a command that directs the CPU to perform a certain operation. There are different kind of operations that the CPU can perform:

- Arithmetic operations, including addition, subtraction, increment and decrement.
- Logical operations like AND, OR, XOR, NOT.
- Bit shifting operations.
- Transfer operations used to transfer some data from a register to another.
- Load operations used to load data from memory.
- Store operations used to store data in memory.
- Stack instructions, used to perform stack operations.
- Flag instructions that can modify the SR.
- Comparison instructions, used to compare data.
- Jumps, used to divert the flow of program execution.
- Software interrupts.
- IO instructions.

In ANC16 an instruction is a word (16-bit long) and consists of the addressing mode (first byte big endian) that specify how the data is fetched (go to addressing modes for more) and the opcode used to specify the instruction.

Registers

Registers are tiny memories used to temporary store data such as addresses, numbers or characters to print on the screen.

Registers can be categorized into different types based on their purpose and usage. General-purpose registers, often referred to as GPRs, are registers that are not designated for a specific purpose but are used for storing data and performing various arithmetic, logic, and other computational operations.

In an ANC216 microprocessor there are 8 GPRs named R0 to R7. Each of these registers is 16-bit long and has a low part of 8-bit that is named Lx where x is the register number.

Additionally, there are 4 other registers used with specific purposes:

- The PC (Program Counter) is a 16-bit register that holds the address of the next instruction to be executed.

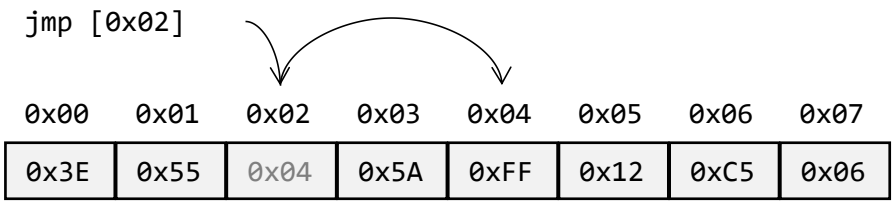
- The SR (Status Register) is a register consisting of a set of individual bits each representing a specific condition, the flags shown here are in ordered like in the SR from bit 7 to bit 0:
  - **N**egative: set if the result is negative.
  - **O**verflow: set if the result of an integer operation is out of the range.
  - **I**nterrupts: set to enable interrupts, (NMI, syscall and reset interrupts are always enabled).
  - **T**imer interrupt: set to enable watch interrupt (go to interrupts section for more).
  - **S**ystem privileges: set to enable system privileges. This flag can be modified only if is set.
  - Reserved, always 1.
  - **Z**ero: set if the result is zero.
  - **C**arry: set if the result has a carry.
- The SP (Stack Pointer) is a 16-bit register used to refer to the top of the stack.
- The BP (Base Pointer) is a 16-bit register used to refer to the base of the stack.

Addressing modes

Addressing modes in ANC216 specify the method by which instruction operands are accessed or fetched from memory. ANC216 incorporates various categories of addressing modes which will not be fully covered in this article, as mentioned before, we recommend viewing the complete article on the ANC216 architecture.

Below a brief description of all available addressing modes:

- Implied: the instruction does not take any operand.
- Immediate: the operand is a constant value.
- Register access: the operand value is stored in one register.
- Register-to-register: the operands are two registers.
- Memory related addressing modes: are those addressing modes that describe how to access memory:
  - Absolute addressing: when the argument of the instruction is the address of a cell in memory.
  - Absolute indexed: from the absolute address, a value stored in a register is added.
  - Indirect addressing: when the argument of the instruction is an address stored in a cell in memory referred by an absolute address. Example:



  - Indirect indexed: from the indirect, a value stored in a register is added to the final address.
  - Relative to PC: when the argument of the instruction is the address of a cell calculated by adding an 8-bit signed value (constant or register) to the PC.

    - Relative to SP: similar to the previous addressing mode, the address i calculated by adding an 8-bit singed value (constant or register) to the SP.
- Register-to-memory: one operand is a register and the other one is a memory related addressing:
    - Register-absolute: one operand is a register and the other is an absolute address. The direction is specified by the instruction. Example:
      `load r6, & 0x30FF` This instruction loads in the register R6 the value stored in `0x30FF`.
      `store & 0x30FF, r3` This instruction is the opposite one, the value on R3 is stored in `0x30FF`.
    - Register-immediate: one operand is a register and the other is a constant value. Example:
      `load r0, 10` This instruction loads the number 10 (`0x0A`) in R0.
    - Register-relative to PC.
    - Register-relative to SP.

Note that in register-to-memory the register can be a low part of the register but in register-to-register it can be only a full-size register.

Basic codes

In this section we will introduce you to the ANC216 with examples and explanations.

The first thing that we will see is how an ANC216 assembly source code always starts.

```
; First ANC216 assembly example

section .text     ; specifying the section text
_code:            ; specifying the entry point
      load r0, 10
```

This is our first example. Note that text after ; is seen as comment from the assembler, a comment is not part of the code.

The `section .text` code is used to specify a section of the memory that contains our code, we will see other section types in the article for other purposes.

The `_code:` is the label that specify the starting of our program. A label is used to make a reference to an address using a symbol.

`load r0, 10` is an instruction used to load the value 10 into the register R0.

In the next code we see arithmetic and logic instructions.

```
; Addition, subtraction and logical operations

section .text
_code:
      load r0, 20
      load r1, 16
      add r0, r1  ; 36 is expected in R0
      load r2, 6
      sub r0, r2  ; 30 is expected in R0
      load r3, 128
      or r0, r3   ; 158 is expected in R0
      and r0, r3  ; 128 is expected in R0
```

In the above code we notice 4 new instructions: add, sub, or, and.

The add instruction is used to add a value into a register.

The sub instruction is used to subtract a value from a register.

The or instruction is used to make a bit-wise or between the value in a register and another value.

The and instruction make a bit-wise and between the register and a value.

Labels and jumps

In this section we will analyze slightly more advanced codes with more labels instead of the only label we have used so far (`_code`).

A jump is an instruction that change the execution flow of our program, it can be unconditional or conditional. In this first code of this chapter we will see the unconditional jump to a custom sub-label into `_code`:

```
; Unconditional jump

section .text
_code:
      load r0, 1
      loop:
            inc r0
            jmp loop
```

In this code we see a sub-label called `loop` into the `_code` label. The first instruction loads the number 1 into the register R0, the second instruction in an `inc` instruction, inc stands for increment, is used to add 1 to a register. Finally, we meet the jump instruction, it is an unconditional jump, that means that the jump will be always executed.

So, we created an infinite loop that increments the value stored in the R0 register.

In the second code proposed below we will use a conditional jump and a high-level label. A high-level label is a label that is not nested into another label but is under a section.

```
; Conditional jump

section .text
_code:
      load r0, 16
      load r1, 16
      cmp r0, r1
      jeq are_equal

are_equal:
      load r0, 0
```

Although this code is not exactly correct, and we will see why in the chapters to come, we take it as good.

We can notice two new instructions: jeq and cmp.

Jeq stands for jump if equal, is a conditional jump which is performed when the cmp instruction compares two values that are equal. Since R0 and R1 are equal, the jump is performed.

Cmp is used to compare two values.

Note: even if R0 and R1 were not equal, the execution of this program would not change because we did not put an instruction after jeq to interrupt and close our program and since labels are just symbols for humans, there is no real separation between the code in the `_code` label and the one in the `are_equal` label.

There are other conditional jumps:

- Jge: jump if grater or equal
- Jgr: jump if grater
- Jle: jump if less or equal
- Jls: jump if less
- Jn: jump if the value is negative
- Jne: jump if not equal
- Jnn: jump if not negative
- Jno: jump if not overflow
- Jnz: jump if the value is not zero
- Jo: jump if overflow
- Jz: jump if zero

System calls

A system call is a software interrupt managed by the operating system and is used by programs to read and modify resources that only the operating system can access. We use system calls to simplify our code and to manage IO.

In this section we will see the exit system call that allows us to terminate the flow of our program.

In order to make a system call we must load some registers with some specific values. For example, the exit syscall requires that L0 to contain 0 and the exit code to be loaded into the R1 register. The exit code is used to specify whether the program terminated with an error or not (the exit code `0x0000` means no errors).

```
; Exit system call

section .text
_code:
      load r0, 10
      loop:
            dec r0
            jnz loop
      load l0, 0
      load r1, 0
      syscall
```

The syscall instruction interrupt the flow of our program and requires an action to be done by the OS. The type of the syscall is stored in L0, the exit syscall has code `0x00`, so we loaded L0 with 0.

Routines

A routine, also called procedure or function, is a custom code that can be used by programmers to simplify the assembly code. To execute a routine you need to 'call' it, the call instruction in ANC216 allows you to to this. Routines can accept parameters that will modify the output or cannot accept any parameter. Parameters are passed via registers or via the stack. For now we will use registers.

```
; Routines

section .text
_code:
      load r0, 56
      load r1, 16
      call add_two_numbers
      load l0, 0
      load r0, 0
      syscall     ; exit

add_two_numbers:  ; numbers are passed via R0 and R1, the output is in R3
      add r0, r1
      tran r3, r0 ; save the result in R3
      ret         ; return from routine
```

This code shows how to use routines. We notice three new instruction, two of which are related to routines.

The call instruction is used to call a routine.

The tran instruction is used to transfer the content of a register into another register. Note that the first register specified is always the destination register and the second is the source register.

Finally, the ret instruction is used to return from a routine.

Strings and constants

A string of character is a sequence of characters, a constant is a value that does not change.

In this chapter we will introduce a new section called `.data`, in this section we put all the strings and constants values.

```
; Strings and constants

section .data
message: "Hello, World!"
message_size: $ - message

section .text
_code:
      load l0, 5
      load r1, message
      load r2, & message_size
      load r3, 0
      syscall
      load l0, 0
```

```
    load r0, 0
    syscall     ; exit
```

We made our first important program with some output. there are many things to explain in this code.

First we set up the data section in order to save the message to be printed on the screen, we saved also the message size with this peculiar syntax, `$ - message`.

In the text section we created our code to print the messag on screen. We used the print system call which has code `0x05` (in fact we loaded L0 with 5). The print system call requires to load the register R1 with the address to the message, the register R2 with the message size and R3 with the stream option (indicates if the message should be printed tn stdout or stderr).

The `$ - message` syntax is used usually to save the size of the content in a label. We could have used `sizeof` as well, however `$ - message` is perfect to introduce operations between pointers, addresses and peculiar addressing modes. We will see the dollar sign in chapters to come.

Advanced codes

Maybe you noticed that up until now we have used only 4 of the 8 registers available, R0, R1, R2 and in some cases also R3. As per standard, registers from R0 to R5 can be used without any problems. R6 and R7 are called also non-volatile registers, this means that their original content must be always restored every time whenever they are used in a routine. The BP is also a non-volatile register.

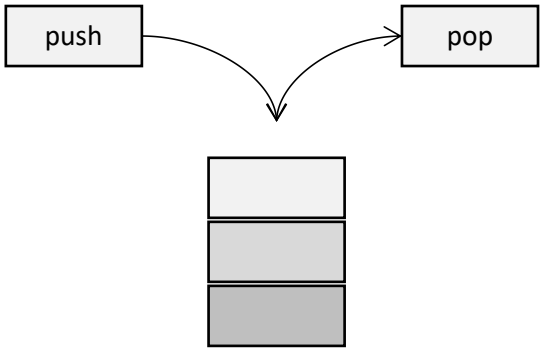
To ensure data retention in non-volatile registers is necessary to know stack instructions.

You have also noticed that we used few addressing modes compared to those that the architecture makes available to us. From now on we will start to see other addressing modes such as indirect, indexed, and relative addressing modes.

Stack

The stack is a memory region used by computers to manage local variables and routine calls. The stack is delimited by two registers, the Base Pointer and the Stack Pointer. The BP points always to the base of the stack and it does not change during the execution of a process, the SP points to the top of the stack.

The stack operates using a "last-in, first-out" (LIFO) principle, meaning that the most recently added item to the stack is the first one to be removed.



There are two main instructions related to the stack: the push instruction and the pop instruction. The push instruction is used to add a value to the top of the stack and increment the SP. The pop instruction, on the other hand, remove the last value added and decrement the SP.

```
; Stack instructions

section .text
_code:
      load r0, -43
      push r0
      pop r1      ; -43 is expected in r1

      load l0, 0
      load r0, 0
      syscall     ; exit
```

In this simple example we passed the value stored in R0 into R1 without using a transfer operation. The stack is a valid alternative is a good alternative for exchanging data from one routine to another without using registers.

returning to the previous discussion, regarding non-volatile registers, on solution to store the original value is to storing it temporary in the stack.

```
; Non-volatile registers

section .text
_code:
      load r0, 10
      load r1, -99
      call routine

      load l0, 0
      load r0, 0
      syscall     ; exit

routine:
      push r6
      phbp
      tran r6, r0
      ; doing stuff with R6...
      pop r6
      pobp
      ret
```

As you can see, we used the R6 register but after using it we restored it to its original value, we also used phbp and pobp to push and pop the base pointer register. Since we did not modify the BP register these instructions are useless, but you should start doing it every time you call a routine to avoid bugs.

Every time you make a call to a routine you should also restore the value of BP after the call to its original value

```
; Restore BP

section .text
_code:
```

```
      phbp
      call my_routine
      ldbp & bp   ; restore BP to its original value

      load l0, 0
      load r0, 0
      syscall     ; exit syscall

my_routine:
      /*do stuff*/
      ret
```

In this code, after the `call my_routine` instruction in the _code label, we restore the value of BP to its original value, otherwise we would use BP as if it were the BP of `my_routine`.

Absolute addressing mode

The absolute addressing mode is used to refer to a cell in the internal physical memory, it can also be indexed by a register, this means that from the specified address you add an 8-bit signed value stored in register from L0 to L7. Remember that Lx registers are the 8-bit low part or Rx registers.

The following code shows how absolute addressing mode works:

```
; Absolute addressing mode

section .data
my_pointer_to_a_word: word 0x0123

section .text
_code:
      load r0, & my_pointer_to_a_word

      load l0, 0
      load r0, 0
      syscall     ; remember always to make an exit syscall
```

In this example we load in R0 the value pointed by our label `my_pointer_to_a_word` which is `0x0123`. To specify the absolute addressing mode we use the & sign. If we had not used the & sing, we would have had an immediate addressing mode. An immediate addressing mode is when you load into a register a constant value, in this case we would have loaded the address of `my_pointer_to_a_word` into the R0 register.

As mentioned above, the absolute addressing mode can be indexed.

```
; Absolute indexed addressing mode

section .data
first_byte: byte 0x01

second_byte: byte 0x02

third_byte: byte 0x03

section .text
```

```
_code:
      load l0, 2
      load r1, & first_byte + l0

      load l0, 0
      load r0, 0
      syscall     ; remember always to make an exit syscall
```

Can you guess what value is stored in R1 after the CPU executes the second instruction in the _code label? Try to guess, the solution is on the next page.

The answer is 3 because we move 2 (because the value stored in L0 is 2) bytes ahead from the `first_byte` label and we end up at `third_byte` label.

The absolute addressing mode (and the indexed one) can be used also in jumps, IO operations and more.

Indirect addressing mode

In indirect addressing mode, instead of specifying the address to the cell containing our data, you specify the address to a cell that contains an address which refers to a cell containing our data. The indirect addressing mode can be indexed, you add the value stored in a register to the second address you fetch (the one stored in a cell in memory).

```
; Indirect AM and indirect indexed AM

section .data
reference_to_my_routine: word my_routine

section .text
my_routine:
      phbp
      load r0, 10
      load r1, -56
      add r0, r1
      popbp
      ret

my_second_routine:
      phbp
      load r0, 10
      load r1, -56
      sub r0, r1
      pobp
      ret

_code:
      phbp        ; save the BP of _code
      call [reference_to_my_routine]
      load l0, sizeof my_routine
      call [reference_to_my_routine] + l0
      ldbp & bp   ; restore BP to original value

      load l0, 0
      load r0, 0
      syscall     ; remember always to make an exit syscall
```

This code may be difficult to understand but it is not. Consider that the indirect addressing mode (and the indexed one) can only be used in jumps or call instructions and since you should not use it in a user program you probably won't see it that often.

In the `reference_to_my_routine` label we store the address to the my_routine. In the `_code` label we call my_routine, then we load the size of my_routine (in bytes) in the L0 register, then we call the routine stored in the address `my_routine + sizeof my_routine` which is

`my_second_routine` (the order of the labels is maintained by the assembler with the exception of the label `_code` which is sometimes placed above all).

The sizeof of my_routine is easy to calculate: an instruction always take 16 bits (a word), the instructions that use register-to-register addressing modes do not take up any more space in memory. The instructions with addressing mode register-immediate take other 8-bits to 16-bits and since we are using Rx registers (full-size register) these instructions will occupy another word (16-bits). The `ret` instruction is implied so it takes 16-bits. If we add up all the calculations made we will see that my routine occupies 12 bytes so the offset between my_routine and my_second_routine is 12 bytes.

Relative

The relative addressing mode is used to specify a location in memory relative to the Program Counter (PC). Is used by user programs since it decreases the probability of causing crashes due to accesses to invalid memory addresses.

An 8-bit singed integer is added to the value stored in the PC, this means that you can specify only addresses that are 127 bytes ahead of your current position or 128 bytes behind.

```
; Relative addressing mode

section .text
_code:
      phbp
      call * parity_check_r0
      ldbp & bp

      load l0, 0
      load r0, 0
      syscall     ; exit syscall

parity_check_r0:
      par r0
      jz * return
      inc r0
      return: ret
```

In order to specify a relative addressing mode, you use the * sign before a label or a constant value. The offset is calculated for us by the assembler.

The `parity_check_r0` routine it is only used to show a new instruction called par. The par instruction sets the zero flag if a value has an even number of 1s.

Relative to the Base Pointer

Is like the relative addressing mode except that the offset is added to the base pointer. It is useful for addressing local variables.

```
; Relative to BP

section .text
_code:
      phbp
```

```
      call * sum_array_of_bytes
      ldbp & bp

      load l0, 0
      load r0, 0
      syscall     ; exit syscall

sum_array_of_bytes:     ; return value in R0
      load l0, 0

      init_loop:
            push l0
            inc l0
            cmp l0, 10
            jl * init_loop

      load l0, 0
      load r0, 0

      sum_loop:
            add r0, & bp + 10
            inc l0
            cmp l0, 10
            jle * sum_loop

      ret               ; pop the return address and return from routine
```

This code sums all values from 0 to 9 stored in an array. An array is a sequence of contiguous cells, in this case is stored in the stack. Note that after running our sum loop, we had to clean up the stack.

The C-like code would look like this:

```
// Sum numbers from 0 to 9 stored in an array

void main()
{
      sum_array_of_bytes();
      exit(0);
}

short sum_array_of_bytes()
{
      char bytes[10];
      short sum;        // R0 register

      for (char i = 0; i < 10; i++) bytes[i] = i;     // see init_loop
      for (char i = 0; i < 10; i++) sum += bytes[i];  // see sum_loop

      return sum;       // R0 stores returned values
}
```

Expert codes

In this final chapter we will see every detail of the ANC216 assembly.

Local variables

Is it possible to save variables into the stack without using push and pop and without accessing them with base pointer relative addressing mode.

```
; Local variables

section .text
_code:
      phbp  ; always push the BP first
      var my_var: byte = 20   ; see my_var as an & bp + <address>
      var my_var2: byte = 23
      load R0, 0
      add R0, my_var
      add R0, my_var2

      load l0, 0
      load r0, 0
      syscall
```

The var can also contains an array reference

```
; Local arrays

section .text
_code:
      phbp
      var array: [10 of byte]
      store array[0], 50

      load l0, 0
      load r0, 0
      syscall
```

Finally we introduce the structures, a structure is a user-defined data type and is used to store and access several variables

```
; Structures

structure My_struct:
      name: word        ; pointer to a string
      last_name: word   ; pointer to a string
      age: byte

section .text
_code:
      var person: My_struct
      store person.name, name
      store person.last_name, last_name
```

```
      store person.age, 20

      load l0, 0
      load r0, 0
      syscall

section .data
name: "Andrew", 0
last_name: "Smith", 0
```

Exercises

Hint: see the OS documentation

1. Write a program that prints a Christmas tree like the one below.
   *
   **
   ***
   ****
2. Write a program that reads a string from the keyboard and counts the vowels.
3. Write a program that sorts the values into an array saved on the stack, the initial values can be chosen but not already in order.