Computer Architecture Project

Instruction Set Architecture

Università di Pisa

Federico Cappellini
Andrea Lelli
Alberto Lunghi
Nicola Mota
Giacomo Pellicci
Lorenzo Susini

may, 2019

1 Introduction & Goals

The goal of this project is to create a simulated electronic calculator able to make computation thanks to an orchestrator.
Our group's goal is to build a new Instruction Set Architecture (ISA) for this calculator based on previous examples found on the literature.
For this purpose we will follow these steps:

1) a brief scheme with the instructions selected by us;
2) a table with the general characteristics of the instructions (what each instruction effectively does), the opcodes and the bit representation
3) write an assembler code to translate an assembly program to binary format
4) write a disassembler code, basically to test the assembler

After all these steps we will write a few written-assembly programs to test if our assembler code and our disassembler code work correctly or not.

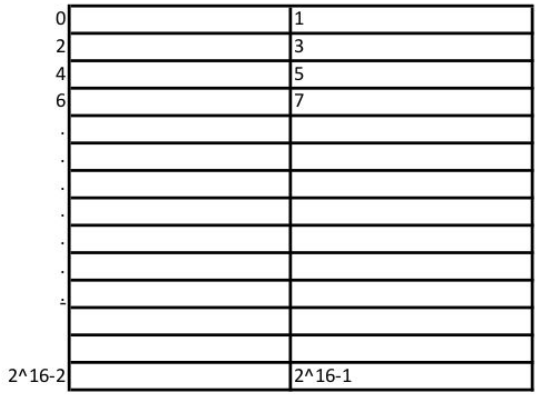
1.1 General structure of the architecture

The architecture that we have considere is based on a 16-bit unit memory, i.e. the minimum unit of memory is 16-bit size, then there is a 32-bit bus and all registers have a dimension of 16 bits. There's a 32-bit bus because each instruction is 32 bits, so in this way it's possible to fetch a full instruction at a time.

The registers are: `AX, BX, CX, DX, SI, DI, FLAG, IP`

All the operations are made on the previous registers and not directly in memory: an address can identify 16 bits, so we can fill a register with a store operation.
Each memory cell has a size of 16-bit.

What follows is a graphical representation of the described memory.



2 RISC type Instruction Set

2.1 What is an Instruction Set Architecture (ISA)

An instruction set architecture (ISA) is an abstract model of a computer.
A realization of an ISA is called an implementation. An ISA permits multiple implementations that may vary in performance, physical size, and monetary cost (among other things); because the ISA serves as the interface between software and hardware. Software that has been written for an ISA can run on different implementations of the same ISA. This has enabled binary compatibility between different generations of computers to be easily achieved, and the development of computer families. Both of these developments have helped to lower the cost of computers and to increase their applicability. For these reasons, the ISA is one of the most important abstractions in computing today.
An ISA defines everything a machine language programmer needs to know in order to program a computer. What an ISA defines differs between ISAs; in general, ISAs define the supported data types, what state there is (such as the main memory and registers) and their semantics (such as the memory consistency and addressing modes), the instruction set (the set of machine instructions that comprises a computer's machine language), and the input/output model.

2.2 Different kinds of ISA

An ISA may be classified in a number of different ways. A common classification is by architectural complexity.
A complex instruction set computer (CISC) has many specialized instructions, some of which may only be rarely used in practical programs. A reduced instruction set computer (RISC) simplifies the processor by efficiently implementing only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines, having their resulting additional processor execution time offset by infrequent use.

2.3 RISC (Reduced Instruction Set Computer)

With RISC we indicate a design idea of the architecture for microprocessors that favors the development of a simple and linear architecture. This simplicity of design allows the creation of microprocessors capable of executing the set of instructions in less time than a classic CISC architecture.
In a RISC architecture we find instructions that are completed in less cycles-per-instruction (CPI) than the same instructions in a CISC architecture.
Another common RISC trait is that memory is accessed through specific instructions (ex: LOAD/STORE), while in a CISC ISA all the instructions are able to access memory in various ways.

3 Registers Set

The microprocessor CA19.32 has a large number of register, identifiable by 8-bit numerical identifier.
We will not use the numerical address as the name of the registers, but we will use acronyms in order to refer to them.
There are 4 kinds of registers:

1. General Registers
2. Instruction Pointer
3. Flag Register
4. Support Registers

and they are all composed by 16 bits.

3.1 General or Operational Registers

The general registers are used to contain data and memory addresses during the program execution. There are 6 general registers and they are 16 bits long.
They are all available to the programmer who will use their acronym writing the mnemonic form of the instruction.

Mnemonic	Meaning	Binary identifier
AX	Accumulator register	00000000
BX	Base register	00000001
CX	Counter register	00000010
DX	Data register	00000011
SI	Source register	00000100
DI	Destination register	00000101

3.2 F - Flag Register

The flag register, called F, is composed by 16 bits and each bit is called flag.
Only 5 of them are relevant and they are:

- OF (Overflow Flag): if OF value is 1 it means that during the execution of the last instruction an overflow occurred. If the instruction works on integer numbers, it means that the result of this particular operation is not representable in 16 bits.
- SF (Sign Flag): if SF value is 1 it means that the result of the execution of the last instruction has 1 as most significant bit. If the instruction works on integer numbers, it means that the result is a negative number.

- ZF (Zero Flag): if ZF value is 1 it means that the result of the last instruction has each bit to 0.
- CF (Carry Flag): if CF value is 1 it means that during the execution of the last instruction it occurs a carryover or a loan. If the instruction works on natural numbers, it means that the result of the operation is not representable.
- NV (Not Valid): if NV value is 1 it means that during the execution of the instruction occurred an error like division by 0 or wrong opcode for an instruction.

3.3 IP - Instruction Pointer

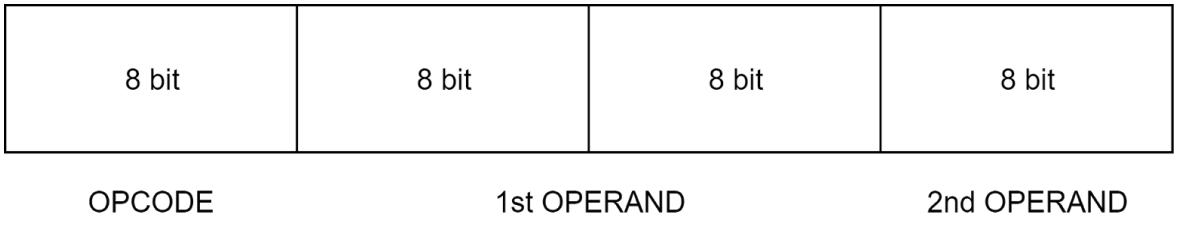
The Instruction Pointer contains the address of the next instruction which have to be executed. This register is automatically set by hardware during the initial reset to the binary value 0x0000 that corresponds to the first memory address and it will be incremented automatically during the execution phase of the previous instruction, in order to maintain the correct computational flow.
Since every instruction is 32 bits long and a single address refers to a 16 bits memory cell, the IP will be incremented by 2 every completed instruction.
Of course jump instructions exist in this instruction set, allowing the programmer to use both conditional and unconditional jump in order to build complex programs.
In case of jump the IP value will be changed to the destination address if the jump condition is met.

4 Proposed instruction set

4.1 Fixed Size Instruction

All the instruction of the ISA have the same fixed length of 32 bits and they have to be naturally aligned.

- The first 8 bits of the instruction are reserved to the opcode
- The following 16 bits are used to represent the first operand
- The last 8 bits are used to represent the second operand



All the instructions, also in the mnemonic format, are in the form `source, dest` for better understandability for the programmer and easier handling in the cpu's fetch phase. Only the instruction `STORE sourceRegister,destinationAddress` , in order to allow the programmer to still write in "source, dest" format, is swapped by the assembler.

All the instruction available are:

Instruction	Operands	Description
`HLT`	none	Halt the processor
`NOP`	none	Does nothing
`JE`	address register	Jump to destination address if equal (ZF=1)
`JNE`	address register	Jump to destination address if not equal (ZF=0)
`JA`	address register	Jump to destination address if above (CF=0 && ZF=0)
`JAE`	address register	Jump to destination address if above or equal (CF=0)
`JB`	address register	Jump to destination address if if below (CF=1)
`JBE`	address register	Jump to destination address if below or equal (CF=1 || ZF=1)
`JG`	address register	Jump to destination address if greater (ZF=0 && SF=OF)
`JGE`	address register	Jump to destination address if greater or equal (SF=OF)
`JL`	address register	Jump to destination address if less (SF!=OF)

JLE	address register	Jump to destination address if less or equal (ZF=1 || SF!=OF)
JZ	address register	Jump to destination address if zero (ZF=1)
JNZ	address register	Jump to destination address if not zero (ZF=0)
JC	address register	Jump to destination address if carry (CF=1)
JNC	address register	Jump to destination address if not carry (CF=0)
JO	address register	Jump to destination address if overflow (OF=1)
JNO	address register	Jump to destination address if not overflo (OF=0)
JS	address register	Jump to destination address if sign (SF=1)
JNS	address register	Jump to destination address if not sign (SF=0)
JMP	address register	Jump to destination address
INC	register	Increment register value
DEC	register	Decrement register value

NEG	register	Change sign to register value
NOT	register	NOT logical operation to register value
MOV	immediate, register register, register	Move source value to destination register and modify flag register accordingly
ADD	immediate, register register, register	Add source value to destination register and modify flag register accordingly
SUB	immediate, register register, register	Subtract source value to destination register and modify flag register accordingly
CMP	immediate, register register, register	Compare two values and modify flag register accordingly
MUL	immediate, register register, register	Multiply destination register by source value and modify flag register accordingly
IMUL	immediate, register register, register	Multiply destination register by source value and modify flag register accordingly
DIV	immediate, register register, register	Divide destination register by source value and modify flag register accordingly
IDIV	immediate, register register, register	Divide destination register by source value and modify flag register accordingly
AND	immediate, register register, register	Logical and between source value and destination register
OR	immediate, register register, register	Logical or between source value and destination register

SHL	immediate, register register, register	Shift left of destination register by source positions
SAL	immediate, register register, register	Shift arithmetic left of destination register by source positions
SHR	immediate, register register, register	Shift right of destination register by source positions
SAR	immediate, register register, register	Shift arithmetic right of destination register by source positions
LOAD	address, register register, register	Load from source address into destination register
STORE	register, address register, register	Store source value into destination address
XCHG	register, register	Exchange the value of two registers

5 Assembler

Machines built with this Instruction Set Architecture will run specific code which can be written by the programmer using specific assembly language. The instruction format is the following:

label:_OPCODE_operand1,operand2

The programmer can combine all the supported instruction in every way in order to build a program that can compute whatever result. This assembly program must be translated in machine code in order to be able to be executed inside the machine, that's why we have developed an Assembler in Java language. The assembler simply take the assembly code and assemble it in machine code producing a binary file and a debug file.

The assembler operates in two different phases. First of all the assembly code is scanned in order to remove empty lines and to store jump labels and their addresses into a structure, which will be used later on to replace instruction like *jmp label* to *jmp 0xADDR*.
A new temporary file will be created as a result of the first phase.

The second phase consists in scanning the new file and compute for each line the appropriate binary translation. This operation will be performed taking first the OPCODE which will tell us both the specific instruction and the kind of its operands, depending on the instruction format. We remember the OPCODE structure here:

FORMAT, ID

Once retrieved the kind of operands and the instruction we can proceed translating the operands. We can have immediate decimal operands ($), address hexadecimal operands (0x) and register operands (%). The assembler will translate immediate and address operands into the corresponding binary values and register operands to the corresponding binary register IDs.

At this point we have all the necessary to build the instruction which is 32 bit long. For those instructions shorter than 32 bit we just add a padding of 0s in order to maintain a static dimension of 32 bit for each instruction in the Instruction Set.

Let's run the Assembler once on the following program:

```
load 0xf0f0, %ax
cmp  $0, %ax
je end1
cmp  $1, %ax
je end1
mov  $1, %dx
```

```
mov  $1, %cx
loop: inc %cx
mul  %cx, %dx
cmp  %ax, %cx
jne  loop
store %dx, 0xf0f2
hlt
end1: mov $1, %ax
store %ax, 0xf0f2
hlt
```

This simple test program computes the factorial of the number stored at the memory location `0xf0f0` and store the result in the location `0xf0f2`.
We just pass the file to the assembler and run it. As result we obtain a binary file which is the one that the machine can understand. More specifically the machine will store those instructions somewhere in memory and when this code will be executed the CPU will be able to understand the instruction correctly and make the correct computation out of this code.

The resulting binary code is the following:

```
01101110111100001111000000000000
01100011000000000000000000000000
00100000000000000001101000000000
01100011000000000000000100000000
00100000000000000001101000000000
01100000000000000000000100000011
01100000000000000000000100000010
01010010000000000000001000000000
10000100000000000000001000000011
10000011000000000000000000000010
00100001000000000000111000000000
01101111111100001111001000000011
00000000000000000000000000000000     *
01100000000000000000000100000000
01101111111100001111001000000000
00000000000000000000000000000000     *
```

Which is of course pretty much not understandable for humans, as it should be.
We can recognize for example in the row marked with a * the translation of the `hlt` instruction which is 32bit equal to 0. Of course one can manually check each instruction comparing them with the ISA specification but in order to have a proof of the correctness of the Assembler we have developed a Disassembler, which has the purpose to translate the machine code back to assembly language, in order to check if the assembler is working properly.

The assembler will provide further information when executed on the terminal, revealing to the programmer many details of each instruction. This feature is of course not needed in a real world assembler, just because programmers are not interested in it, but we have decided to show them for educational purpose. The extra verbosity of the assembler can be removed by simply commenting out those print lines.

6 Disassembler

While the assembler produces a raw binary file interpreting a plain text human-written file, it is useful to have an opposite program able to convert back the raw binary to an human-readable file. The main reasons of a disassembler are:

1. the possibility to have a tool able to confirm the correctness of the assembler,
2. a simple way to obtain back a program source code.

Anyway, compiling a program is not a completely reversible process because, although it is possible to determine a correct mnemonic code for each binary instruction, it is impossible to recover those information that are not present inside the binary file that were present in the original source code, such as the comments and the labels. This mean that, in the decompiled source code, there are no comments or labels and those instructions, that originally used a label, in the decompiled source simply use an address (the address the label referenced).

For instance, the following program which calculates the Fibonacci sequence

```
            load 0xf0f0, %dx
            cmp $0, %dx
            je end0
            cmp $1, %dx
            je end1

            mov $2, %cx
            mov $0, %ax
            mov $1, %bx

loop:xchg %ax, %bx
            add %ax, %bx
            inc %cx
```

```
            cmp %cx, %dx
            je end

end0: mov   $0, %ax
            store %ax, 0xf0f2
            hlt

end1: mov $1, %ax
            store %ax, 0xf0f2
            hlt

end:  store %bx, 0xf0f2
            hlt
```

appears a little different once assembled and than disassembled:

```
load 0xf0f0, %dx
cmp 0x0000, %dx
je 0x001a
cmp 0x0001, %dx
je 0x0020
mov 0x0002, %cx
mov 0x0000, %ax
mov 0x0001, %bx
xchg %ax, %bx
add %ax, %bx
inc %cx
cmp %cx, %dx
je 0x0026
mov 0x0000, %ax
store %ax, 0xf0f2
hlt
mov 0x0001, %ax
store %ax, 0xf0f2
hlt
store %bx, 0xf0f2
hlt
```

The disassembler works in 3 steps repeated as many times as needed until the conversion is completed:

1. Since all the binary instructions have the same fixed size of 32 bits, in the initial step the tool reads 4 byte from the binary file and store them in an array.
2. A new instance of the class Instruction is initialized thanks to the previous array.
3. The logic of the conversion happens thanks to the code of the class methods.
    a. The array is splitted in 3 parts and those data are interpreted to populate 3 member variables: the opcode, the first operand and the second operand.

b. The instruction type is determined analyzing the opcode.
c. The mnemonic code is determined knowing type, opcode and operands thanks to a static conversion table that is the same used by the assembler.

6.1 Example of a program able to calculate the exponentiation

The following source code is from an assembly program able to calculate the exponentiation.

```
            load  0xf0f0, %ax
            load  0xf0f1, %bx
            cmp   $0, %bx
            je    end0
            mov   $1, %dx
            mov   $0, %cx

loop:       mul   %ax, %dx
            inc   %cx
            cmp   %cx, %bx
            jne   loop

end:        store %dx, 0xf0f2
            hlt

end0:       mov   $0, %ax
            store %ax, 0xf0f2
            hlt
```

After the compiling phase, the raw binary can be textually represented as a sequence of binary numbers like the following:

```
01101110111100001111000000000000
01101110111100001111000100000001
01100011000000000000000000000001
00100000000000000001100000000000
01100000000000000000000100000011
01100000000000000000000000000010
10000100000000000000000000000011
01010010000000000000001000000000
10000011000000000000001000000001
00100001000000000000110000000000
01101111111100001111001000000011
00000000000000000000000000000000
01100000000000000000000000000000
01101111111100001111001000000000
```

```
00000000000000000000000000000000
```

Then, the disassembler outputs the following source code:

```
load 0xf0f0, %ax
load 0xf0f1, %bx
cmp 0x0000, %bx
je 0x0018
mov 0x0001, %dx
mov 0x0000, %cx
mul %ax, %dx
inc %cx
cmp %cx, %bx
jne 0x000c
store %dx, 0xf0f2
hlt
mov 0x0000, %ax
store %ax, 0xf0f2
hlt
```

7 Testing

The aim of the test phase is to show that a programmer is able to build a program that makes computations in order to achieve a result and to check the correctness of the Assembler. The test programs that have been developed are able to compute the factorial, the modulus and the exponentiation of a number, and also to compute the n-th numbers of the Fibonacci sequence. In this way we have verified that the Instruction Set is really usable to write any kind of program.
In order to check the correctness of the Assembler, we have used in a different way the same test programs. Infact, the easiest way to verify that the Assembler is working properly, is to assemble and then disassemble them. As already said in this document, we have built a human-readable debug file, that we have used to check if each instruction is translated in the right way. But the most important thing is to check if the binary file generated by the Assembler is correct. For this reason the Disassembler was developed and then used to compare its output to our source test program. Since the Disassembler is able to rebuild the program, we have the proof that each binary file generated by our Assembler is ready to be executed once the bytes of the binary file are in memory.