A SIC/XE to Intel Pentium x86 assembly code translator

Benjamin Kastelic
Fakulteta za računalništvo in informatiko
Univerza v Ljubljani
Tržaška 25, 1000 Ljubljana, Slovenija
kastelic.benjamin@gmail.com

Abstract

The Simplified Instruction Computer (SIC) is a hypothetical computer designed for educational purposes. Due to a reasonable number of instructions, sufficient memory space and its scalable structure, SIC is a perfect tool that enables a clear and ballast-free illustration of the basics of the system software (e.g. assembling, linking and loading) and other concepts of computer software and hardware design. Therefore it is frequently used as a demonstration gadget in Systems Programming and Operating Systems courses at university level. The main drawback of a SIC computer is the fact, that it does not really exist in physical form. Hence, when testing a SIC program one has to use a simulator, i.e. a computer program that simulates SIC instructions step-by-step. Besides the fact that such an approach uses a simulated (and thus potentially unreal) environment, the speed of execution is another issue that inhibits a comfortable work. To overcome this we have developed a computer program that translates from SIC to Intel x86 assembly code. Using this translator, the programs developed for a hypothetical computer can be executed in a real environment. Since the target computer has much wider instruction set, the translation is always successful and the resulting programs are fast and reliable. In the development of our translator we managed to convert data from SIC's 24 bit to Intel's 32 bit system and we covered all the input/output operations using files on the target computer.

1 Introduction

The SIC[1] (Simplified Instruction Computer) hypothetical computer was designed to illustrate the concepts and features of common computer hardware, while avoiding the "clutter" found in real computers, which often only confuses the novice. Like many other products even SIC is available in two versions: a standard version and an XE (*Extra Equipment*, also *"eXtra Expensive"*) version. Both editions were designed to be upward compatible – this means that the object code, which was generated for SIC, can also be executed on a SIC/XE computer. This kind of compatibility is often found on real world systems. Because of its simplicity, the SIC computer is mostly used as a teaching example at many schools around the world. Since SIC is just a hypothetical computer, programs can only be executed on a dedicated virtual machine. This means that all of the instructions are simulated step by step by a specific simulator in an unreal environment. After using SIC and a SIC simulator in class, we ~~thought it might~~ be useful and even instructive for students, ~~if we could~~ create a translator, ~~that would be~~ capable of translating SIC programs into a ~~program, capable of running on real computers~~. By doing so, students would be able to run their programs in a real environment and also learn the basics of that machines assembly language, by observing the translations. We have selected the Intel x86 Pentium architecture as the target of our translator.

In this paper we describe the process of finding the correct translations between SIC/XE instructions and the Intel x86 instructions. We describe in detail the problems we encountered and solutions to most of them.

The rest of this paper is organized as follows. Sections 2 and 3 describe the properties of SIC/XE and Intel x86 architectures. Section 4 presents the process of determining correct translations between SIC/XE and x86 instructions. Finally, Section 5 draws our conclusions and points out some future work directions.

2 SIC/XE architecture

2.1 Memory

Memory consists of consecutive 8 bit byte arrays. Three neighbouring bytes form a word. All of the SIC/XE addresses are byte addresses. Words are addressed by the location of their most important bit (*big-endian byte ordering*). There are a total of 1048576 (2^{20}) bytes in the computer memory.

2.2 Registers

There are nine registers, each of them serving it's own purpose. The size of each register is 24 bits, except for register F, which size is 48 bits. Table 1 displays the mnemonics and uses of all the registers.

Mnemonic	Usage
A	accumulator; arithmetic operations
X	index register; addressing
L	linkage register; ~~v ta register ukaz JSUB shrani povratni naslov~~
PC	program counter; holds the address of the next instruction
SW	status word; stores different information – the "condition code" (CC) amongst others
B	base register; addressing
S	general purpose register
T	general purpose register
F	48-bit register; floating point numbers storage

Table 1: Mnemonics and usage of the SIC/XE registers

2.3 Data formats

Integer numbers are stored as 24-bit signed binary numbers, ~~2's~~ complement representation is used for negative values. Characters are stored using their 8-bit ASCII codes. Floating-point numbers are represented with a 48-bit format, which can be seen in Figure 1.

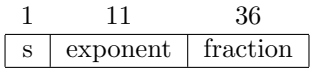

Figure 1: Floating-point number format

The fraction is interpreted as a value between 0 and 1 (assuming the binary point is immediately before the high-order bit). For normalized floating-point numbers, the high-order bit of the fraction must be 1. The exponent is interpreted as an unsigned binary number between 0 and 2047. If the exponent has value e and the fraction has value f, the absolute value of the number represented is:

$$f \times 2^{e-1024}$$

The sign of the floating-point number is indicated by the value of s (0 = positive, 1 = negative). A value of zero is represented by setting all bits (including s, e and f) to 0.

2.4 Instruction formats

All standard SIC instructions use the same 24-bit instruction format (Figure 2)~~, which cannot be used on the SIC/XE computer~~. ~~This is due to increased memory size,~~ as 15 bits alone cannot ~~be used to address~~ the entire memory space. ~~T~~here are two possible options – either ~~use~~ some form of relative addressing, or ~~extend~~ the address field to 20 bits. Both of these options are included in SIC/XE (formats 3 and 4). If bit e (Figure 5 and 6) is set to 0 format 3 is used, otherwise format 4 is used. In addition SIC/XE provides some instructions that do not reference memory at all (format 1 and 2).

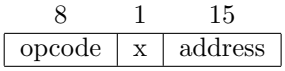

Figure 2: Standard SIC instruction format

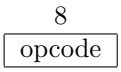

Figure 3: Format 1

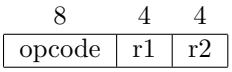

Figure 4: Format 2

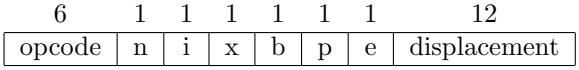

Figure 5: Format 3

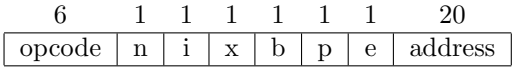

Figure 6: Format 4

2.5 Addressing modes

In general we distinguish ~~two methods of addressing:~~

- according to target address ~~(or TA for short)~~ calculation,
- according to the ~~manner of target address usage, which is used to determine the operand~~

According to target address calculation, there are three modes of addressing known to SIC/XE – one mode of indirect and two modes of relative addressing. The modes of target address calculation ~~can be seen~~ in Table 2. Parentheses are used to indicate the contents of a register or memory location. For example, (B) represents the contents of register B.

Mode	Indication	TA calculation
Direct	$b = 0, p = 0$	$address$
Base relative	$b = 1, p = 0$	$(B) + disp$
PC relative	$b = 0, p = 1$	$(PC) + disp$

Table 2: SIC/XE addressing modes according to target address calculation

For base relative addressing, the displacement field *disp* in a format 3 instruction is interpreted as a 12-bit unsigned integer. For PC relative addressing, this field is interpreted as a 12-bit signed integer, with negative values represented in ~~2~~'s complement notation.
If bits b and p are both set to 0, the *disp* field from the format 3 instruction is used as the target address. For a format 4 instruction, bits b and p are normally ~~set 0~~, and the target address is taken from the address field of the instruction. This is called *direct* addressing, to distinguish it from the relative addressing modes described before. Any of these addressing modes can be combined with the *indexed* addressing – if bit x is set to 1, the ~~contents~~ of register X is added to the target address calculation.
Bits i and n in format 3 and format 4 instructions are used to specify how the target address is used. If bit $i = 1$ and $n = 0$, the target address itself is used as the operand value. This is called *immediate* addressing. If bit $i = 0$ and $n = 1$, the word at the location given by the target address is fetched; the value contained in this word is then taken as the address of the operand value. This is called *indirect* addressing. If bits i and n are bot set to 0 or both set to 1, the target address is taken as the location of the operand. This is called *simple* addressing.
SIC/XE instructions that specify neither immediate nor indirect addressing are assembled with bits n and i both set to 1. Assemblers for the standard version of SIC will, however, set both bits to 0 (this is because the 8-bit binary codes for all of the SIC instructions end ~~in 0~~). All SIC/XE machines have a special hardware feature designed to provide backward compatibility. If bits n and i are both 0, then bits b, p and e are considered to be part of the address field of the instruction, rather than flags indicating addressing modes. This makes instructions of format 3 identical to the format used on the standard version SIC, providing the desired compatibility.

According to target address usage, there are also three modes of addressing known to SIC/XE, which are shown in Table 3.

Mode	Indication	Target address usage
Simple	none	$operand = (TA)$
Immediate	#	$operand = TA$
Indirect	@	$operand = ((TA))$

Table 3: SIC/XE addressing modes according to target address usage

3 Intel Pentium x86 architecture

3.1 Memory

Memory in the x86 architecture can be described in at least two ways. At the physical level, memory consists of 8-bit bytes. Every address adresses a byte. Two consecutive bytes form a *word*, four bytes form a *doubleword* (also called a *dword*). Some operations are more efficient when operands are aligned in a particular way (for example, an operand that begins at a byte address that is a multiple of 4).
Programmers usually view the ~~the~~ x86 memory as a collection of *segments*. From this point of view, an address consists of two parts – a segment number and an offset that points to a byte within the segment. Segments can be of different sizes, and are often used for different purposes. Some segments may contain executable instructions, and others may be used to store data. Some data segments may be treated as stacks that can be used to save register contents, pass parameters to subroutines and for other purposes.

3.2 Registers

There are eight general-purpose registers: *EAX*, *EBX*, *ECX*, *EDX*, *EBP*, *ESI*, *EDI* and *ESP*. Each of these registers is 32 bits long (one doubleword). Registers *EAX*, *EBX*, *ECX* and *EDX* are generally used for data manipulation; it is possible to access individual words or bytes from these registers (when accessing different parts, the registers get a different name, ~~which can be seen in~~ Figure 7). The other four registers can also be used for data manipulation, but are more commonly used to hold addresses.
There are also several different types of special-purpose registers. The *EIP* register is a 32-bit register that contains a pointer to the next instruction to be executed. The *EFLAGS* register is also 32 bits long and contains many different bit flags. Some of

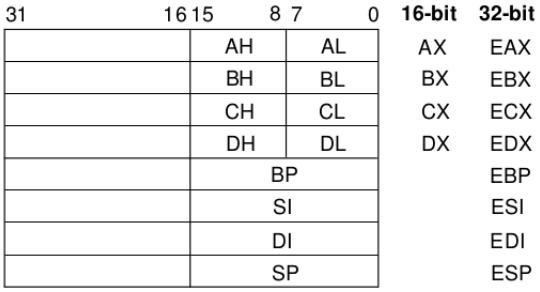

Figure 7: General-purpose registers of the Intel Pentium x86 computer

these flags indicate the status of the processor, others are used to record the results of comparisons and arithmetic operations.

There are also six 16-bit *segment* registers that are used to locate segments in memory. Segment register *CS* contains the address of the currently executing code segment, and register *SS* contains the address of the current stack segment. The other registers (*DS*, *ES*, *FS* and *GS*) are used to indicate the addresses of data segments.

Floating-point computations are performed using a special *FPU* (floating-point unit). This unit contains eight 80-bit data registers, that resemble a stack, and several other control and status registers. More details about the FPU can be found in [3].

Besides all these registers, that are available only to application programs, there are also several other registers, that are used only by system programs (e.g. operating system) and a few others, that control the operation of the processor.

3.3 Data formats

The x86 architecture provides ~~for the storage of integers, floating-point values, characters and strings.~~ Integers are stored as 8-, 16- or 32-bit binary numbers. Both signed and unsigned integers are supported; like on SIC/XE, the ~~2~~'s complement is used for negative values representation.

There are three different floating-point data formats. The *single-precision* format is 32 bits long. It stores 24 significant bits of the floating-point value and allows for a 7-bit exponent (the remaining bit is used to store the sign of the floating-point value). The *double-precision* format is 64 bits long. It stores 53 significant bits and allows for a 10-bit exponent. The last, *extended-precision* format is 80 bits long that stores 64 significant bits and allows for a 15-bit exponent.

Characters are stored as 8-bit ASCII codes.

3.4 Instruction formats

All of the x86 machine instructions use variations of the same basic format. This format begins with optional prefixes containing flags that modify the operation of the instruction. For example, some flags specify a repetition count for an instruction, and others specify a segment register that is to be used for addressing. ~~F~~ollowing is a number of bytes that specify the operands and addressing modes to be used.
The operation code is the only element that is always present in every instruction. Other elements may or may not be present. Thus there are a large number of different instruction formats, varying in length from 1 to 10 bytes.

3.5 Addressing modes

The x86 architecture provides a large number of addressing modes. An operand value may be specified as part of the instruction itself (*immediate mode*), or it may be in a register. Operands stored in memory are ofter specified using variations of the general target address calculation:

$$TA = (base\ register) + displacement + (index\ register) \times (scale\ factor)$$

Any general-purpose register may be used as a base register. Likewise, any general-purpose register except *ESP* can be used as an index register. The scale factor can have the value 1, 2, 4, or 8 and the displacement can be an 8-, 16- or 32-bit value. The base and index register numbers, scale factor and displacement are encoded as parts of the operand specifiers in the instruction. Because these items are not required they can be omitted, we get eight different addressing modes. The address of an operand in memory may also be specified ~~an~~ an absolute location (*direct* mode) or as a location relative to the *EIP* register (*relative* mode).

4 Transformations

4.1 Data types

As ~~was~~ mentioned before, one byte on the SIC/XE architecture is a sequence of 8 bits and one word is composed of 3 bytes. Therefore, byte mapping (`BYTE` instructions) didn't present a problem, since the x86 bytes are also 8 bits long. That was not the case with the `WORD` instruction. On the x86 architecture a word is 16 bits long, which is not enough for a successful mapping. That is why a SIC/XE word is translated

SIC	Intel
`BYTE val`	`.byte val`
`WORD val`	`.long val`
`RESB n`	`.space n`
`RESW n`	`.space 4*n`
`l EQU e`	`.equ l, e`

Table 4: Data type translations ~~SIC~~ – Intel

SIC	Intel
A	EAX
X	EDI
L	EBP
PC	EIP
SW	EFLAGS
B	EBX
S	ECX
T	EDX
F	top of FPU stack

Table 5: Registers translations ~~SIC~~ – Intel

into a doubleword (`.long` instruction), which is 32 bits long.
The next problem with translating the previously mentioned data types is the way, how both architectures store and address words. SIC addresses words by the location of their most important bit, which is exactly opposite to the x86 architecture. This has a major impact on storing and accessing of arrays elements.
The rest of the translations didn't present any particular problems. The final mappings ~~can be found~~ in Table 4.

4.2 Registers

The translation of registers is pretty straightforward, since the x86 architecture possess a ~~few~~ more registers than SIC architecture. ~~All that is left to do is assign the correct translations. The results can be seen in~~ Table 5.

4.3 Addressing

As ~~was~~ mentioned in Section 2.5, SIC/XE knows three addressing modes according to target address usage – simple, immediate and indirect addressing. Since the x86 architecture knows simple and immediate addressing, except for certain instructions, there were no difficulties with translating these addressing modes. A small problem ~~present those~~ instructions~~, which~~ do not support immediate addressing. When dealing with ~~those~~ instructions, ~~we have~~ to ~~first~~ store the operand value as a separate variable

	SIC	Intel
simple	`LDA _X`	`mov %eax, [_X]`
immediate	`LDA #5`	`mov %eax, 5`
indirect	`LDA @_X`	`mov %esi, [_X]` `mov %eax, [%esi]`

Table 6: Addressing modes translations ~~SIC~~ – Intel

and then use simple addressing to read the value of that variable. There was a similar situation with indirect addressing. When using indirect addressing on SIC/XE, the operand value is at the address, ~~designated~~ by the contents of a memory location at TA address ($operand = ((TA))$). On the x86 architecture an operand value can be addressed indirectly, only if the operand address is stored in advance in a chosen register. This is called *register indirect addressing*. Luckily, x86 possesses more registers than SIC/XE. This enabled us to reserve the *ESI* register for use with indirect addressing. Thus, an instruction, using indirect addressing, is translated into a sequence of two instructions: loading the operand address into *ESI* register and reading the final operand value from the address contained in register *ESI*. Like on ~~SIC~~, we can also use indexing with simple and indirect addressing. We do this by adding the contents of the index register to the target address. In our case, the index register is *EDI*. Example translations can be seen in Table 6.

4.4 Instructions

`LDx` instructions

`LDA m`, `LDB m`, `LDCH m`, `LDL m`, `LDS m`, `LDT m`, `LDX m`

All of the above instructions are simply translated into `mov` instructions:

`mov %`X`, [m]`

where X is a chosen register. The `LDCH` instruction is a bit different, as it is used to load a byte into register A instead of a word. That is why the 32-bit *%eax* register is replaced with the 8-bit `%al` register.

`STx` instructions

`STA m`, `STB m`, `STCH m`, `STL m`, `STS m`, `STT m`, `STX m`

~~Similar to `LDx` instructions, also all of these instructions are~~ easily translated into `mov` instructions:

`mov [m], %`X

where X is a chosen register. The `STCH` instruction is translated similarly as the `LDCH` instruction.

Jump instructions

`J m`, `JEQ m`, `JGT m`, `JLT m`

The first instruction differs from the rest, as it presents an unconditional jump and is therefore translated into `jmp m` instruction. The rest are all conditional instructions and are translated into `je m`, `jg m` and `jl m`, ~~respectfully~~. A safeguard has been added to all translated jump instructions in order to prevent infinite loops. ~~In case~~ an infinite loop is found~~,~~ a jump instruction is replaced with a sequence of instructions that end~~s~~ the program (`sys_exit`), thus preventing infinite execution.

Instructions for device manipulation

`TD m`, `RD m`, `WD m`

Our translator treats ~~SIC~~ devices as text files. For example, if a ~~SIC~~ program ~~were to read~~ from a device ~~labelled as~~ `0xAA`, the translated program will actually read from a text file named `AA.txt`.
All three instructions are translated into a sequence of instructions, that trigger the `0x80` interrupt. This interrupt is used to signal the operating system to execute a certain system call operation (Table 7). The `TD` instruction is translated into a sequence of instructions, that trigger the interrupt, responsible for executing the `sys_open` system call, which creates a text file with a specified name and returns the newly created file's descriptor. Or, if the file already exists, only the file descriptor is returned. This information is stored in a separate variable, which can later be used by the other two instructions. Once the file descriptor is obtained, reading or writing from the created file is actually quite easy. To read and write from a file, one has to use the `sys_read` and the `sys_write` systems calls, ~~respectfully~~. For every system call~~,~~ we have to preload the designated (usually *EAX, EBX, ECX* and *EDX*) registers with ~~necessary~~ information, like system call ID, file descriptor, character array address and so on. Devices 0 (stdin), 1 (stdout) and 2 (stderr) are a bit different. Since these devices are permanently available, one does not have to create a new file and can be used immediately for reading and writing.
Additional information pertaining the usage of system calls in a Linux environment can be found in [2] and [4].

Other instructions

`DIVR r1, r2`

The `div` instruction of the x86 architecture divides the *EAX* register with some register or an in-memory operand. ~~SIC~~'s `DIVR` is a bit different - both operands are registers. As long as *r1* is equal to the *A* register, the instruction can be translated as if using normal division. If that is not the case, a few other instructions have to be added. First, we have to store the contents of the *EAX* register on the stack, then copy *r2* into *EAX* and only then we can divide by *r1*. When we are finished, the result is copied back to *r2* and *EAX* is restored to its original state.

```
int sys_open(
   char* filename, int flags, int mode
)

ssize_t sys_read(
   int fd, char* buf, size_t count
)

ssize_t sys_write(
   int fd, char* buf, size_t count
)
```

Table 7: Linux file I/O system calls

We have gone through almost a half of all the instructions by now. Because the remaining half doesn't belong to any major group and because they translate easily into x86 assembly language, we will present their translations in Table 8.
If you look closely at Table 8, you might see, that some instructions are missing. Those instructions are: `HIO`, `LPS`, `NORM`, `SIO`, `SSK`, `STI`, `STSW`, `SVC` and `TIO`. The reason for omitting these instructions is the lack of documentation and their absence in example programs. By not knowing what exactly they do, it is impossible to find correct translation for the Intel computer.

5 Conclusion - TODO

Namen pretvornika je, da omogoča prevedbo ter posledično tudi izvajanje programov, napisanih za izmišljen računalnik, na pravem stroju. Tako bi dobili zadovoljstvo opazovanja izvajajanja programa na pravem računalniku, hkrati pa bi se s proučevanjem seznanili z novim zbirnim jezikom.
During the development phase we encountered several problems. The most noticeable ones were the translation of SIC's 24-bit word to Intel's 16-bit word and the translation between SIC's big- and Intel's little-endian byte ordering. We were able to solve the first problem but not the second. It is up to the programmer to keep this in mind when writing SIC programs, to adjust the arrays properly, in order for

SIC	Intel
ADD m	add %eax, m
ADDF m	fadd m
ADDR m	add %r2, %r1
AND m	and %eax, m
CLEAR r1	mov %r1, 0
COMP m	cmp %eax, m
COMPF m	fcom m
COMPR r1, r2	cmp %r1, %r2
DIV m	div m
DIVF m	fdiv m
FIX	fist m
FLOAT	fild m
JSUB m	call m
LDF m	fld m
MUL m	imul %eax, m
MULF m	fmul m
MULR m	imul %r2, %r1
OR m	or %eax, m
RMO r1, r2	mov %r2, %r1
RSUB	ret
SHIFTL r1, n	shl %r1, n
SHIFTR r1, n	shr %r1, n
STF m	fst m
SUB m	sub %eax, m
SVC n	int n
SUBF m	fsub m
SUBR r1, r2	sub %r2, %r1
TIX m	add %edi, 1 cmp %edi, m
TIXR r1	add %edi, 1 cmp %edi, %r1

Table 8: Instructions translations ~~SIC~~ – Intel

the translated programs to work on Intel's architecture.
That is why we believe that this problem should be the first of many improvements that would have to made, in order to relieve the programmers (in our case students) of such low-level problems and allowing them to focus more on the language and programs itself.
TODO: optimization, ...

References

[1] Leland L. Beck. *System Software: An Introduction to Systems Programming (3rd Edition)*. Addison-Wesley, 1996.

[2] Richard Blum. *Professional Assembly Language (Programmer to Programmer)*. Wrox, 2005.

[3] Intel Corporation. *Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 1997. http://download.intel.com/design/intarch/manuals/24319001.pdf.

[4] Intel Corporation. *Intel Architecture Software Developer's Manual Volume 3: System Programming*, 1999. http://communities.intel.com/servlet/JiveServlet/downloadBody/5061-102-1-8118/Pentium_SW_Developers_Manual_Vol3_SystemProgramming.pdf.