

UNIVERSITY OF LJUBLJANA
FACULTY OF COMPUTER AND INFORMATION SCIENCE

Klemen Kloboves

**Implementation of a SIC/XE
Processor on an FPGA and
Supporting System Software**

BACHELOR'S THESIS

UNDERGRADUATE UNIVERSITY STUDY PROGRAM
COMPUTER SCIENCE AND MATHEMATICS

ADVISOR: Assist. Prof. Tomaž Dobravec, PhD

CO-ADVISOR: Assoc. Prof. Patricio Bulić, PhD

Ljubljana 2014

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Klemen Kloboves

**Implementacija procesorja SIC/XE
na FPGA in podporna sistemska
programska oprema**

DIPLOMSKO DELO

UNIVERZITETNI INTERDISCIPLINARNI ŠTUDIJSKI
PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN
MATEMATIKA

MENTOR: doc. dr. Tomaž Dobravec

SOMENTOR: izr. prof. dr. Patricio Bulić

Ljubljana 2014

This work is licensed under a Creative Commons Attribution 4.0 International License. Details about this license are available online at: <http://creativecommons.org>



Source code of the hardware and software developed as part of this work is licensed under the MIT License and is available online at: <https://github.com/kloboves/sicxe>

The Faculty of Computer and Information Science issues the following thesis:

The Simplified Instruction Computer (SIC) is a hypothetical computer designed for educational purposes. SIC/XE is an extended version of this computer with additional features. Both computers were designed by Leland L. Beck and are used in many Systems Programming and Operating Systems courses at university level. In your thesis implement a computer on the Digilent Nexys 2 FPGA development board. The computer should contain a SIC/XE processor, device controllers and other auxiliary components. To support the testing process, design and implement the system software including assembler, linker and simulator utilities for programs written in SIC/XE assembly language.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

SIC je hipotetični računalnik, ki se uporablja v učnem procesu za poučevanje razvoja in uporabe systemske programske opreme. SIC/XE je nadgradnja osnovnega SIC računalnika in vsebuje večje število ukazov ter večji pomnilnik. Računalnika, ki ju je zasnoval Leland L. Beck, se uporabljata kot pripomoček pri mnogih predmetih o systemski programski opremi in o operacijskih sistemih na univerzah po vsem svetu. V diplomski nalogi implementirajte računalnik na Digilent Nexys 2 FPGA razvojni plošči. Računalnik naj vsebuje procesor SIC/XE, krmilnike naprav in pomožne komponente. Razvijte tudi systemsko programsko opremo za razvoj in testiranje SIC/XE programov, vključno z zbirnikom, povezovalnikom in izvajalnim okoljem. Programska oprema naj omogoča poganjanje in razhroščevanje programov napisanih v SIC/XE zbirnem jeziku.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Klemen Kloboves, z vpisno številko **63110208**, sem avtor diplomskega dela z naslovom:

*Implementacija procesorja SIC/XE na FPGA in
podporna sistemska programska oprema*

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca in somentorstvom izr. prof. dr. Patricia Bulića,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 8. julija 2014

Podpis avtorja:

Contents

Abstract

Povzetek

Razširjeni povzetek

1	Introduction	1
2	About the SIC/XE Architecture	5
2.1	System Memory and Devices	5
2.2	Registers	6
2.3	Instruction Formats	7
2.4	Addressing Modes	8
2.5	Interrupts	10
2.6	Object File Format	11
3	SIC/XE on a Xilinx Spartan 3E FPGA	15
3.1	Memory Controller	17
3.2	Personal Computer Interface	18
3.3	Device Subsystem	22
3.4	Central Processing Unit	26
4	SIC/XE System Software	31
4.1	Simulator	31
4.2	Assembler	34

CONTENTS

4.3	Linker	45
4.4	FPGA Interface Utility	46
5	Conclusions	49

Table of Acronyms

acronym	meaning
ALU	arithmetic logic unit
CPU	central processing unit
DRAM	dynamic random-access memory
FPGA	field-programmable gate array
FSM	finite state machine
I/O	input/output
LED	light emitting diode
PSRAM	pseudo-static DRAM
VHDL	VHSIC hardware description language
VHSIC	very high speed integrated circuit

Abstract

SIC/XE is a hypothetical computer architecture designed for teaching systems software programming. In this work, we present an educational SIC/XE computer that we implemented on an FPGA development board. The system consists of a SIC/XE processor, device controllers and other auxiliary components. In addition, we also developed a suite of system software utilities for use with our system. The suite includes assembler, linker and simulator utilities which can be used to create and test programs for the SIC/XE computer. Programs can be transferred from a personal computer to the FPGA board over a serial connection, using a provided software tool.

Keywords: SIC/XE, FPGA, processor, assembler, linker, simulator.

Povzetek

SIC/XE je hipotetična računalniška arhitektura namenjena učenju sistemskega programiranja. V tem delu je predstavljen izobraževalni SIC/XE računalnik, ki smo ga implementirali na FPGA razvojni plošči. Sistem je sestavljen iz SIC/XE procesorja, krmilnikov naprav in ostalih pomožnih komponent. Poleg tega smo razvili tudi zbirko sistemskih programov namenjenih za uporabo s sistemom. Zbirka programov vključuje zbirnik, povezovalnik in simulator, s katerimi lahko ustvarjamo in preizkušamo programe za računalnik SIC/XE. Programe lahko prenesemo iz osebnega računalnika na FPGA ploščo preko serijske povezave z uporabo priloženega programa.

Ključne besede: SIC/XE, FPGA, procesor, zbirnik, povezovalnik, simulator.

Razširjeni povzetek

V diplomski nalogi je predstavljen izobraževalni računalniški sistem SIC/XE, ki smo ga implementirali na FPGA razvojni plošči. Razvili smo tudi zbirko sistemskih programov za uporabo s sistemom. Zbirka programov vključuje zbirnik, povezovalnik in simulator ter orodje za prenos programov z osebnega računalnika na FPGA ploščo preko serijske povezave.

SIC/XE je hipotetični računalniški sistem namenjen poučevanju sistemskega programiranja. Predstavil ga je Leland L. Beck v svoji knjigi “System Software: An introduction to Systems Programming”. Arhitektura ima večino funkcionalnosti, ki so prisotne v resničnih arhitekturah, vendar je manj kompleksna, zaradi česar je bolj primerna za učenje sistemskega programiranja. Ker je SIC/XE hipotetična računalniška arhitektura, ne obstaja veliko strojnih računalnikov s to arhitekturo. Na voljo pa so programski simulatorji, ki posnemajo delovanje SIC/XE računalnika na računalniku z drugo arhitekturo.

Programski simulatorji imajo nekatere omejitve, zato smo želeli narediti strojni SIC/XE računalnik. Pravi strojni sistem omogoča uporabnikom pisanje programov, ki direktno komunicirajo s strojnimi napravami kot so ekrani, tipke, stikala in lučke. Za izdelavo sistema smo uporabili FPGA tehnologijo podjetja Xilinx. FPGA tehnologija omogoča hitro in cenovno ugodno izdelavo prototipov digitalnih vezij. Za osnovo sistema smo izbrali razvojno ploščo Digilent Nexys 2. Na plošči je FPGA čip Xilinx Spartan 3E, PSRAM pomnilniški čip in razne vhodno/izhodne naprave, zaradi česar je plošča primerna izbira za naš sistem. Ker gre za izobraževalni sistem, smo želeli, da

je sistem čim bolj enostaven za uporabo in da je iskanje napak v programih preprosto. Zagotoviti smo morali enostaven način prenosa programov na ploščo, kot tudi način za pregledovanje glavnega pomnilnika. Sistem je zato načrtovan za uporabo skupaj z osebnim računalnikom, s katerim se poveže s serijsko povezavo, preko katere lahko osebni računalnik neposredno dostopa do pomnilnika. Poleg dostopa do pomnilnika lahko osebni računalnik preko povezave pošilja ukaze, s katerimi lahko uporabnik ustavi in zažene procesor, sproži prekinitve ali ponovno zažene sistem. Sistem nima dolgotrajnega pomnilnika, vendar to ne predstavlja težav saj se lahko programe kadarkoli ponovno naloži v pomnilnik s povezanega osebnega računalnika.

Za načrtovanje sistema smo uporabili strojni opisni jezik VHDL. Načrt je sestavljen iz štirih komponent: krmilnik pomnilnika, podsistem naprav, enota za komuniciranje z osebnim računalnikom in CPE (centralna procesna enota). Krmilnik pomnilnika je zadolžen za upravljanje pomnilniškega čipa Micron PSRAM, ki služi kot glavni sistemski pomnilnik. Podsistem naprav sestavlja več krmilnikov naprav, ki krmilijo vhodno/izhodne naprave, med drugim tipke, stikala, lučke, priključek PS/2 in priključek VGA. Enota za komuniciranje z osebnim računalnikom komunicira s priključenim osebnim računalnikom preko serijske povezave, kot je opisano v prejšnjem odstavku. Najbolj pomembna komponenta pa je procesor SIC/XE, ki izvaja ukaze, shranjene v glavnem pomnilniku. Procesor je ne-cevovoden in podpira nekoliko spremenjeno različico arhitekture SIC/XE. Razlike med arhitekturama so majhne in opisane v poglavju 2.

Poleg strojne implementacije sistema SIC/XE smo razvili tudi zbirko sistemskih programov za uporabo sistema. Programe so razvili v programskem jeziku C++ za operacijski sistem Linux, vendar se bi jih brez večjih težav dalo prilagoditi za druge operacijske sisteme podobne sistemu UNIX. Zbirka programov vključuje naslednje programe: zbirnik, povezovalnik, simulator in orodje za komunikacijo s FPGA ploščo.

Zbirnik je program za prevajanje kode iz zbirnega jezika v strojno kodo. Naš zbirnik podpira večino funkcionalnosti zbirnega jezika SIC/XE, med dru-

gim literale, programske bloke in zunanje simbole. Zbirnik strojno kodo izpiše v objektne datoteke. Poleg objektnih datotek pa lahko izpiše tudi dnevniške datoteke, ki vsebujejo izpis izvorne kode skupaj s pripadajočo strojno kodo, in izpis simbolne tabele, tabele blokov in tabele literalov.

Povezovalnik je program za združevanje več objektnih datotek v en izvršljiv program. Programerju omogoča, da lahko večje programe razbije v več izvornih datotek, ki jih posamezno prevede v objektne datoteke z zbirnikom, te pa nato združi v končni program s povezovalnikom.

Simulator je program, ki omogoča izvajanje SIC/XE programov na splošno namenskem računalniku. Kompatibilen je z našim FPGA sistemom, podpira pa tudi nekaj dodatnih funkcionalnosti, kot naprimer računanje s števili v plavajoči vejici. Simulator ima interaktivni vmesnik z ukazno vrstico. Uporabnik nadzira izvajanje programa v simulatorju z vpisovanjem ukazov. Na voljo so tudi raznovrstni ukazi, s katerimi lahko uporabnik pregleduje in spreminja stanje procesorja in vsebino pomnilnika. Za lažje iskanje napak v programih pa je vgrajen tudi povratni zbirnik, mehanizem prekinitvenih točk in mehanizem za nadzor spremenljivk.

Razvili smo tudi podporni program za komuniciranje s FPGA ploščo preko serijske povezave. To orodje ima enak interaktivni vmesnik z ukazno vrstico, kot ga ima naš simulator. Ko program vzpostavi povezavo s ploščo, lahko uporabnik pregleduje in spreminja vsebino glavnega pomnilnika, tako kot v simulatorju. Vsakič, ko uporabnik vpiše ukaz, program avtomatsko prenaša potrebne podatke med ploščo in osebnim računalnikom. Obstajajo tudi ukazi, s katerimi lahko ustavimo in zaženemo procesor ali sprožimo prekinitve.

Chapter 1

Introduction

In this work, we present an educational computer system based on the SIC/XE architecture, which we implemented on an FPGA development board. We also developed a suite of system software utilities for use with the system. The suite includes assembler, linker and simulator programs, as well as a utility for transferring programs from a personal computer to the FPGA board over a serial connection.

SIC/XE is a hypothetical computer system architecture designed for teaching computer systems programming. It was introduced by Leland L. Beck in his book “System Software: An Introduction to Systems Programming” [1]. The architecture has most of the features present in real world architectures but is less complex, making it better suited for those wanting to learn about systems programming. Because SIC/XE is a hypothetical architecture, there are very few hardware implementations of the architecture. Most people use software emulators, which simulate a virtual SIC/XE computer on a regular general purpose computer.

We wanted to create an educational hardware SIC/XE computer system because software emulators have certain limitations. A hardware system allows users to experiment with writing programs that directly interact with physical hardware devices like screens, buttons, switches and lights. To build our system, we used FPGA technology from Xilinx. FPGA technology allows

fast and cost effective prototyping of digital circuits. The Digilent Nexys 2 development board was chosen as the basis for the system, because it has a Xilinx Spartan 3E FPGA, a PSRAM (pseudo-static dynamic random access memory) chip and various input/output devices making it a great fit for our design. Because we were developing an educational computer system, we wanted to make it easy to use and make debugging of programs as convenient as possible. We needed to provide an easy way of transferring programs to the system, as well as a way of inspecting the contents of main system memory for debugging. This is why we designed our system to be used together with a personal computer. The system is connected to a personal computer with a serial connection, through which the personal computer can directly access the contents of main system memory. In addition to memory access, the personal computer can send commands through the connection, which allow the user to suspend or resume the processor, trigger interrupts or reset the system. There is no non-volatile storage, as programs can be transferred directly into main system memory from the personal computer whenever they are needed.

We implemented the design of our system using the VHDL hardware description language. The design comprises of the following components: memory controller, device subsystem, personal computer interface and the SIC/XE processor. The memory controller component is responsible for operating the Micron PSRAM memory chip, which is used as main system memory. Input/output devices including buttons, LEDs (light emitting diodes), switches, the VGA port and the PS/2 port are managed by device controllers which make up the device subsystem. The personal computer interface computer component handles communication with the personal computer over a serial connection, as described in the previous paragraph. The most important component of the design is the SIC/XE processor, which executes instructions stored in main system memory. The processor is non-pipelined and supports a slightly modified version of the SIC/XE architecture. The differences between our version of the architecture and the original architecture

are described in Chapter 2.

In addition to the hardware SIC/XE system, we developed a suite of system software utilities for use with the system. The programs were developed in C++ programming language for the Linux operating system, but should be easy to port to other UNIX-like systems. The suite contains the following programs: assembler, linker, simulator and FPGA interface utility.

The assembler is a program for translating code from assembly language to binary machine code. It supports most features of the SIC/XE assembly language including literals, program blocks and external symbols. In addition to writing code to object files, it can output nicely formatted log files, which include code and the contents of the symbol table, block table and literal table.

The linker utility combines multiple object files together into a single executable program. The programmer can split up larger programs into multiple assembly source files, convert them to object files using the assembler, and then use the linker utility to combine them together to get the completed program.

The simulator utility emulates a hardware SIC/XE system in software. Our simulator is fully compatible with the FPGA based system described earlier and also has some additional features like floating point support. The simulator has an interactive command line interface. Various commands are available which allow the user to inspect the state of the processor, view the contents of memory and step through the program. The simulator has a built-in disassembler that can print out instructions in assembly language form as they are executed. There is also a breakpoint feature which allows the user to specify addresses at which to stop the program, as well as a watch-list feature which makes it easy to keep an eye on memory locations of interest. These features make the simulator a very useful tool for debugging programs before running them on the FPGA board.

Finally, the FPGA interface utility is used to communicate with the FPGA board over a serial connection. It has the same kind of interactive

easy to use command line interface as the simulator. Once connected to the board, contents of the board's main system memory can be viewed and modified, just like in the simulator. Every time a command is issued, the utility automatically transfers data to and from the FPGA board. There are also control commands available which can be used to suspend, resume or reset the processor.

Chapter 2

About the SIC/XE Architecture

Our SIC/XE computer is based on a slightly modified version of the SIC/XE architecture. We decided to make some changes to the architecture because implementing certain features of the original architecture would significantly complicate the design of our system, yet not make it notably more useful. Most core features of the architecture are left unchanged — main registers, all instruction formats and addressing modes, system memory and I/O (input/output) devices are all present and work the same as in the original architecture. The following features were removed: I/O channels, privileged execution mode and support for multiple processes (there is no process identifier). Interrupts are supported, but are implemented in a different way than in the original architecture. Floating point operations and integer division instructions are available in the simulator, but not on the FPGA based system described in Chapter 3. The modified version of the SIC/XE architecture used in our system is described in this chapter. There is also a short description of the SIC/XE object file format at the end of the chapter.

2.1 System Memory and Devices

Main system memory is used to store programs and data. There is 1 megabyte (2^{20} 8-bit bytes) of main system memory available. Three types of memory

operands are supported: bytes, words and floats. Words are 24 bits wide (3 bytes) and there are no alignment restrictions. Floating point operands are 48 bits wide (6 bytes) and there are also no alignment restrictions. Memory operands are addressed with their lowest byte and have Big-Endian byte order (the most significant byte has the lowest address).

In addition to the memory address space, there is a separate device address space. There are 2^8 (256) device addresses. Data can be sent to and from devices using I/O instructions *WD* (write to device) and *RD* (read from device), one byte at a time.

2.2 Registers

Registers available on the SIC/XE processor are listed in Table 2.1.

name	width (bits)	description
A	24	accumulator
X	24	index register
L	24	linkage register
B	24	base register
S	24	general purpose register
T	24	general purpose register
F	48	floating point register
PC	20	program counter
CC	2	condition code register
I	1	interrupt enable register
IL	20	interrupt linkage register
ICC	2	interrupt condition code register

Table 2.1: SIC/XE registers

Registers A through T contain 24 bits wide signed integers. They can be used as general purpose registers, although registers A, X, L and B have

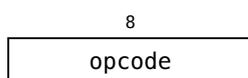
special functions. Register A (accumulator) is used for storing results of many arithmetic and logical instructions. Register L (linkage register) is used for storing the return address when calling subroutines. The JSUB (jump to subroutine) instruction jumps to a given address and stores the previous value of the program counter register to register L. Returning back from the subroutine is accomplished with the RSUB instruction, which copies the value of register L back to the program counter register. Register B (base register) is used as the base address for base relative addressing. The X (index register) is added to the target address when indexed addressing is used. The condition code register (CC) is used to store the result of comparison operations. It can be in one of the following 3 states: 00_2 (less), 01_2 (equal), 10_2 (greater). Conditional jump instructions (JEQ, JLT, JGT) perform a jump only if the condition code register is in the correct state — for instance, JEQ jumps only when the CC register has state 01_2 (equal), and so on.

Registers I, IL and ICC are not present in the original SIC/XE architecture. They were added to our modified SIC/XE architecture as part of the interrupt mechanism. Register I controls whether or not interrupts are enabled. When an interrupt is triggered, the processor starts executing an interrupt handler. After the interrupt handler finishes, it is important that the processor can be restored back to its previous state. Registers IL (interrupt linkage register) and ICC (interrupt condition code register) are used to store old values of registers PC and CC, respectively. The values of all other registers must be saved and restored manually in the interrupt handler.

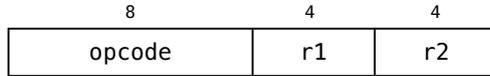
2.3 Instruction Formats

SIC/XE instructions can be in one of the following 5 formats:

Format 1 (1 byte):



Format 2 (2 bytes):



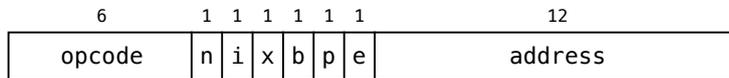
SIC compatibility format (3 bytes):

$$n = 0 \wedge i = 0$$



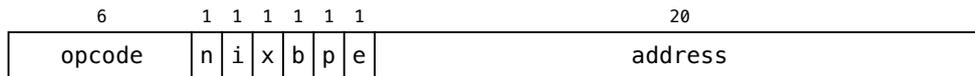
Format 3 (3 bytes):

$$(n \neq 0 \vee i \neq 0) \wedge e = 0$$



Format 4 (4 bytes):

$$(n \neq 0 \vee i \neq 0) \wedge e = 1$$



Instructions in formats 1 and 2 have an 8-bit opcode, while instructions in formats SIC, 3 and 4 have a shorter 6-bit opcode. Format 1 instructions have no operands. Format 2 instructions operate on registers, fields **r1** and **r2** are used to select one of the registers **A**, **X**, **L**, **B**, **S** or **T**. An exception to this rule are bit shift instructions, where field **r2** is used to specify the number of bits register **r1** should be shifted.

Instructions in formats SIC, 3 and 4 have a memory operand. Instructions with a 6-bit opcode can be in any of these three formats. Addressing bits **n**, **i**, **x**, **b**, **p** and **e** determine the format and addressing mode used.

2.4 Addressing Modes

Instructions that have a memory operand may be in formats SIC, 3 or 4 and support many different memory addressing modes. The addressing mode is determined by addressing bits from the instruction. All available addressing

n	i	x	b	p	e	format	operand	addressing
1	1	0	0	0	0	3	[address] ¹	direct
1	1	0	0	0	1	4	[address]	direct
1	1	0	0	1	0	3	[PC + <i>sgn</i> (address)]	PC-relative
1	1	0	1	0	0	3	[B + address]	base relative
1	1	1	0	0	0	3	[X + address]	indexed
1	1	1	0	0	1	4	[X + address]	indexed
1	1	1	0	1	0	3	[PC + X + <i>sgn</i> (address)]	PC-rel. indexed
1	1	1	1	0	0	3	[B + X + address]	base rel. inixed
0	0	0	-	-	-	SIC	[address]	direct
0	0	1	-	-	-	SIC	[X + address]	indexed
1	0	0	0	0	0	3	[[address]]	indirect
1	0	0	0	0	1	4	[[address]]	indirect
1	0	0	0	1	0	3	[[PC + <i>sgn</i> (address)]]	indirect PC-rel.
1	0	0	1	0	0	3	[[B + address]]	indirect base rel.
0	1	0	0	0	0	3	address	immediate
0	1	0	0	0	1	4	address	immediate
0	1	0	0	1	0	3	PC + <i>sgn</i> (address)	immed. PC-rel.
0	1	0	1	0	0	3	B + address	immed. base rel.

Table 2.2: SIC/XE addressing modes

modes are listed in Table 2.2. Combinations of addressing bits not present in the table are considered invalid and trigger an invalid addressing error.

To access a memory operand, its actual address needs to be calculated first. This address is called the *target* address, and is obtained as follows. The target address is first set to the value of the **address** field from the instruction. The **address** field is sign extended when PC-relative addressing is selected — it is interpreted as an unsigned integer otherwise. Depending on the addressing mode selected, values of registers PC (PC-relative), B (base

¹[*x*] denotes value at memory address *x*

relative) or **X** (indexed) are added to the target address. Addressing bits **p**, **b** and **x** determine which of these relative addressing modes are enabled.

The main three types of addressing modes are immediate, simple and indirect. Addressing bits **n** and **i** determine which one is used. In immediate addressing mode, the target address itself is the operand. This mode cannot be used with store instructions (triggers invalid addressing error). In simple addressing mode, the memory location pointed to by the target address is the operand. In indirect addressing mode, the memory location pointed to by the target address contains the address of the operand.

If addressing bit **e** is set, the instruction is in format 4 instead of in format 3. Format 4 instructions have a 20 bit long **address** field, which means all memory locations can be addressed directly. PC-relative addressing and base addressing are not allowed in format 4.

2.5 Interrupts

Interrupts make it possible for the processor to respond to asynchronous events. The original SIC/XE architecture described in Leland Beck's book [1] features an interrupt mechanism, but we decided not to use it in our system. Instead, we used our own simpler interrupt mechanism. The main difference between our mechanism and the mechanism used in the original SIC/XE architecture is that our system has fewer interrupts and in the way the system saves and restores the state of the processor. The original SIC/XE architecture has four maskable interrupts. The state of all registers is automatically saved to predefined locations in memory when an interrupt is triggered. In contrast, our system only has one maskable interrupt. When an interrupt is triggered, registers are not automatically saved to memory, but must be manually saved and restored in the interrupt handler by the programmer.

Register **I** controls whether or not interrupts are enabled. All interrupt requests are ignored while interrupts are disabled. The value of register **I** can be changed using instructions **EINT** (enable interrupt) and **DINT** (disable

interrupt).

If register `I` has a value of 1 and an interrupt request is received, the processor stops its current task and jumps to the interrupt handler. The processor first saves the current program counter to register `IL` and copies register `CC` to register `ICC`. It then sets register `I` to 0, disabling further interrupts. Finally, the interrupt handler address is loaded from address `0xffffd` and copied to the program counter register. Execution then continues at the beginning of the interrupt handler.

The processor does not automatically save any other registers, which is why the interrupt handler must restore any registers that were changed back to their prior state before returning. When the interrupt handler wishes to return control back to the program that was interrupted, it can do so using the `RINT` instruction. This instruction restores register `CC` from register `ICC` and register `PC` from register `IL`. The processor then resumes execution of the program from the point where it was interrupted.

2.6 Object File Format

Object files are used to store machine code. They also contain metadata, which make it possible to link and relocate code contained in the object file. Although the majority of commonly used object file formats are binary files, SIC/XE object files are plain text files. This is inefficient in terms of file size, but makes it easy to read and modify files using a simple text editor.

SIC/XE object files are composed of multiple sections, each line of the file is one section. The first character of each line determines which type of section the line represents. There are 6 types of sections: start section, text section, symbol export section, symbol import section, modification section and end section. Every program must have exactly one start section and one end section, which must be the first and the last sections in the file, respectively.

An example object file is given below.

```

HSAMPLE0003E8000BBB                                ] start section
T0003E812030FA09010C094014B100000031000009040    ]
T0003FB16A0453320033F2FED900173200040934A456D5C3F2FFD ] text sections
T000FA003123456                                     ]
DBAZ  0003F0QUX  000403SAMPLE0003E8                ] sym. export section
RBAR  F00                                             ] sym. import section
M0003E903                                           ]
M0003F105+F00                                       ] modification
M0003F505+BAR                                       ] sections
E0003E8                                             ] end section

```

There are three fields in a start section: program name, start address and code size. The program name must be 6 characters long. The start address and code size are 6 character long hexadecimal numbers.

```

HSAMPLE0003E8000BBB
  program   start   code
  name     address size

```

Each text section contains a small fragment of program code. Text sections start with an address field followed by a size field. The address field determines at which address the section should be loaded into memory. The size field determines the number of bytes of data in the code field. The code field contains program data in hexadecimal format.

```

T0003FB16A0453320033F2FED900173200040934A456D5C3F2FFD
  address size                                     code

```

Symbol export sections tell the linker which symbols are defined in the object file. A symbol export section can contain more than one symbol as shown in the example below. Each symbol has a name field and an address field.

```

DBAZ  0003F0QUX  000403SAMPLE0003E8
  symbol  symbol  symbol  symbol  symbol  symbol
  name   address name   address name   address

```

Symbol import sections tell the linker which external symbols are referenced from the object file. Symbol import sections can also contain more than one symbol.

```

RBAR  F00
  symbol  symbol
  name   name

```

Modification sections make it possible to link and relocate code. There are two types of modification sections: simple modification sections and symbol modification sections. Both have an address field and a size field. The address field determines the address at which code should be modified. The number of nibbles (half bytes) that need to be modified is given in the size field. Simple modification records tell the linker to add the start address of the program to a memory location. They are used to correct instructions with absolute addressing when the program is linked or relocated.

M0003E903
 address size

Symbol modification sections tell the linker to add (or subtract) the address of a symbol to a memory location. The sign field determines whether the address should be added or subtracted. They are used to correct instructions that reference memory locations from other object files.

M0003F105+F00
 address size sign symbol
 name

The end section marks the end of an object file. The only field in the end section is the entry point field, which is the address where program execution should begin.

E0003E8
 entry
 point

Chapter 3

SIC/XE on a Xilinx Spartan 3E FPGA

In this chapter we present the design of an educational SIC/XE computer system we implemented on a Digilent Nexys 2 development board. In addition to the Xilinx Spartan 3E FPGA chip, the board has a Micron PSRAM memory chip and various I/O devices, which we incorporated into the design of our system. The clock signal is provided by a 50 MHz crystal oscillator installed on the board [3]. An overview schematic of the system is shown in Figure 3.1.

The system is composed of four main components: the SIC/XE processor, memory controller, device subsystem and personal computer interface. The most important component is the CPU (central processing unit), which executes instructions stored in main system memory. It is connected to the memory controller, through which it accesses system memory, and to the device subsystem so it can communicate with device controllers. When the *enable* signal is inactive, the processor enters a *suspended* state and temporarily pauses program execution. This signal can be toggled on or off using a button on the board. Device controllers can make interrupt requests using the *interrupt* signal. When the processor encounters an error like a malformed instruction, it stops and activates the *error* signal which displays an

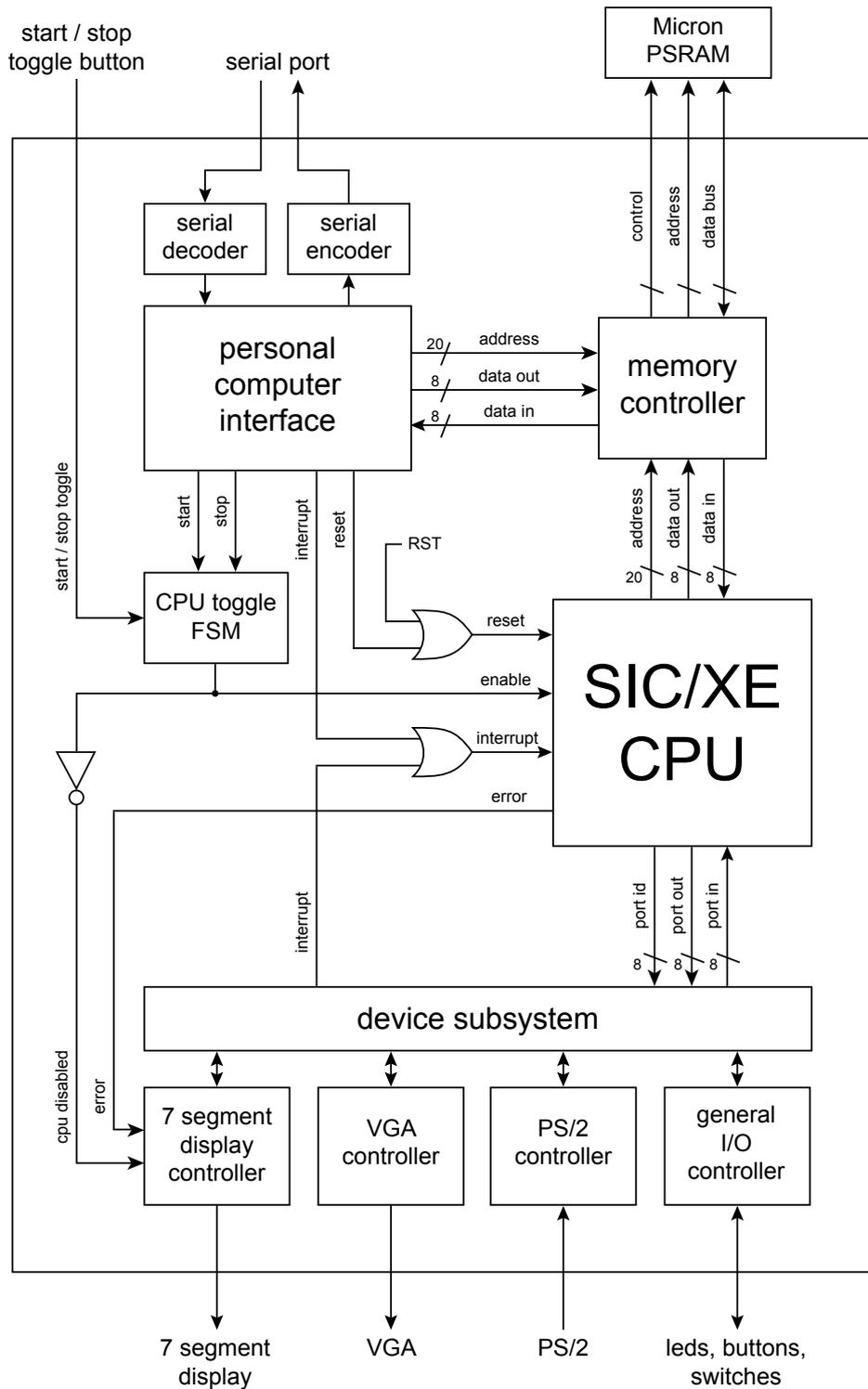


Figure 3.1: System design schematic

error message on the 7 segment display. The 7 segment display also displays a “STOP” message when the processor is suspended.

The system is connected to a personal computer with a serial connection, which allows the contents of main system memory to be accessed from the personal computer. The personal computer interface component handles communication with the personal computer. A simple communication protocol described later in this chapter is used to transmit data and commands over the connection. The component is connected to the memory controller, through which it can access system memory.

The memory controller is responsible for operating the Micron PSRAM chip, which is used for main system memory. It is designed to allow the processor and the desktop computer interface to access memory interchangeably. In case both devices simultaneously initiate a memory cycle, the personal computer interface is given priority while the processor has to wait.

The device subsystem is composed of four device controllers. They control I/O devices present on the board, including buttons, switches, LEDs and a 7 segment display. An external PS/2 keyboard and a VGA display can also be connected. Interrupts are triggered whenever an input device receives new data.

3.1 Memory Controller

The SIC/XE architecture requires the system to have 1 megabyte of main system memory. To accommodate this, we decided to use the Micron PSRAM chip present on the board [3] for main system memory. The chip has 16 megabytes of capacity which is much more than we needed. However, the chip has a 16 bit wide data bus, which would make accessing unaligned 3 byte SIC/XE words problematic. We decided to use a simple solution to overcome the problem. We only use the lower 8 bits of the data bus, essentially using the chip as if it was half the size and only had an 8 bit data bus. In this arrangement, our system occupies the lower 2 megabytes of available space

on the chip. Three memory operations are needed to access a 3 byte word.

The Micron PSRAM chip has multiple modes of operation including asynchronous mode, burst mode and page mode. We chose to use asynchronous mode as it is the simplest. The access time is 70 ns [5], which means a memory operation can be completed in 5 clock cycles at 50 MHz (one clock cycle is 20 ns).

In our system, both the processor and the personal computer interface need to access main system memory. To make this possible, we created the memory controller component. In addition to allowing both devices to access memory interchangeably, it also takes care of the details of operating the Micron PSRAM chip. Internally, the memory controller is made up of a few multiplexers controlled by a simple FSM (finite state machine). When a device requests a memory operation, it waits until it receives a signal from the memory controller indicating that the operation was successfully completed. If a device requests a memory operation while another memory operation is in progress, the memory controller makes it wait until both operations are finished. If both devices request a memory operation at the same time, the personal computer interface is given priority. This decision was made because it accesses memory far less frequently than the processor and thus causes little disruption. It also guarantees that the personal computer interface can not be blocked off by the processor continuously accessing memory.

3.2 Personal Computer Interface

The personal computer interface component communicates with the desktop computer and services requests on its behalf. The desktop computer is connected to the board with a serial cable. A serial encoder circuit and a serial decoder circuit handle the details of serial communication. The speed of transmission used on the serial connection is 115200 baud (bits per second).

A simple communication protocol is used to transfer data over the serial connection. The personal computer starts a session by transmitting a one

byte *command*. The device responds to the command by either acknowledging it or rejecting it. Depending on the command, additional data may be transmitted by the personal computer or the device to follow up the command. After a command is fully executed the session is over and the device goes back to listening for new commands. The device never initiates a session, it only listens and responds to commands from the personal computer.

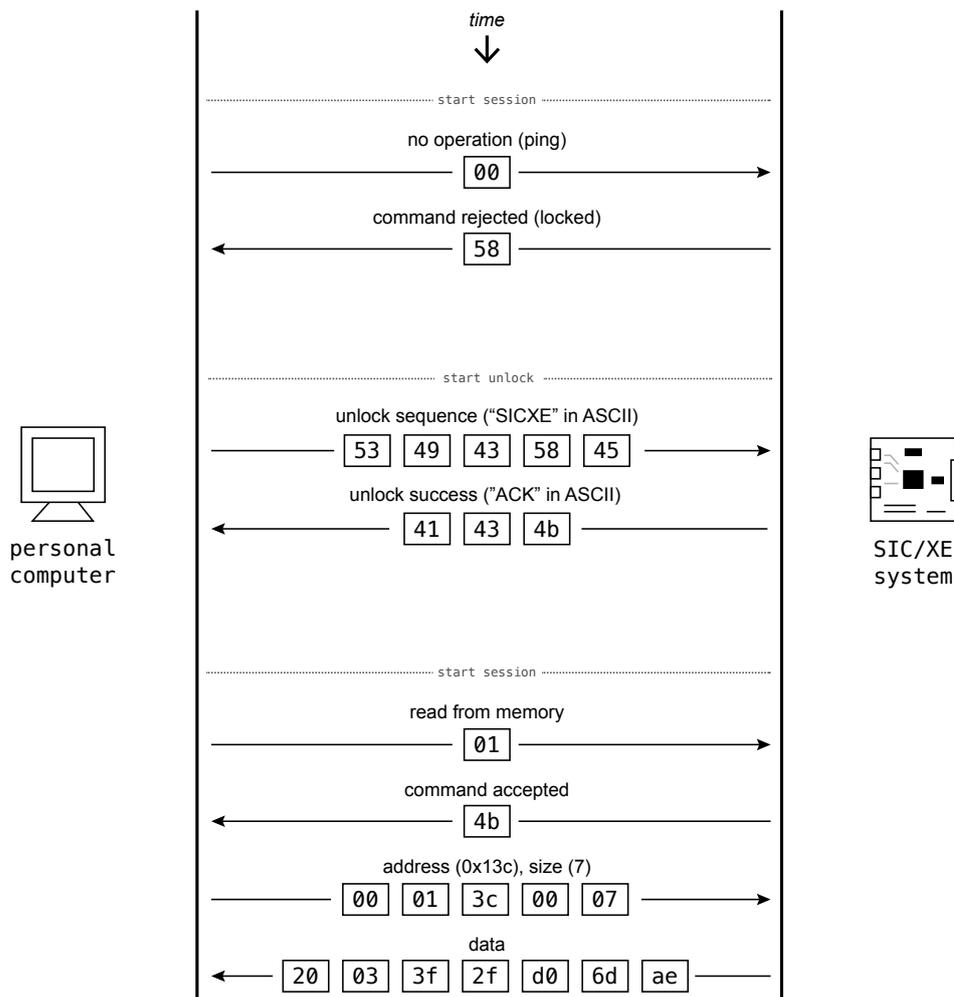


Figure 3.2: Communication protocol illustration

The communication protocol is illustrated in Figure 3.2. The personal computer starts out by sending a one byte command to the device. The

device replies with a one byte response containing the value 0x4b (ASCII letter "K") if the command is accepted, or the value 0x58 (ASCII letter "X") if the command is rejected. The device starts up in a *locked* state, in which it rejects all incoming commands as shown in the first example in Figure 3.2. Before any commands can be accepted, the device needs to be unlocked by sending an unlock sequence (ASCII letters "SICXE"). The device responds to the unlock sequence by responding with an unlock success sequence (ASCII letters "ACK") and transitioning to the *unlocked* state in which it accepts commands.

The unlock sequence mechanism has been implemented to prevent unintended commands from being executed in case the personal computer sends unexpected data. The unlock procedure only has to be done once, after which any number of commands can be executed. In case the personal computer somehow breaks the protocol (for example by sending an invalid command) the device returns to the locked state. If this happens, the device has to be unlocked again before more commands can be executed.

cmd.	description	follow up (computer)	follow up (device)
0x00	no operation (ping)		
0x01	read from memory	address, size	data
0x02	write to memory	address, size, data	
0x10	reset CPU		
0x11	start CPU		
0x12	stop CPU		
0x13	interrupt		
0xff	back to locked state		

Table 3.1: Computer interface commands

Table 3.1 lists all available commands. Most commands are completed when the device responds with the acknowledgement value 0x58. However, the read and write commands require follow up data to be sent from the

personal computer. In both commands, the computer must send a 3 byte address indicating at which memory address data should be read or written, followed by a 2 byte size indicating how many bytes should be transferred. When performing a write command, the computer proceeds to send the required number of bytes which are then written to memory. In the case of a read command, the device reads memory contents and transmits the required number of bytes back to the personal computer (shown in the last example in Figure 3.2)

Our FPGA interface utility described in Chapter 4.4 implements the protocol described above and can be used to communicate with the device, but users can also write their own programs to communicate with the device as long as they follow the described protocol.

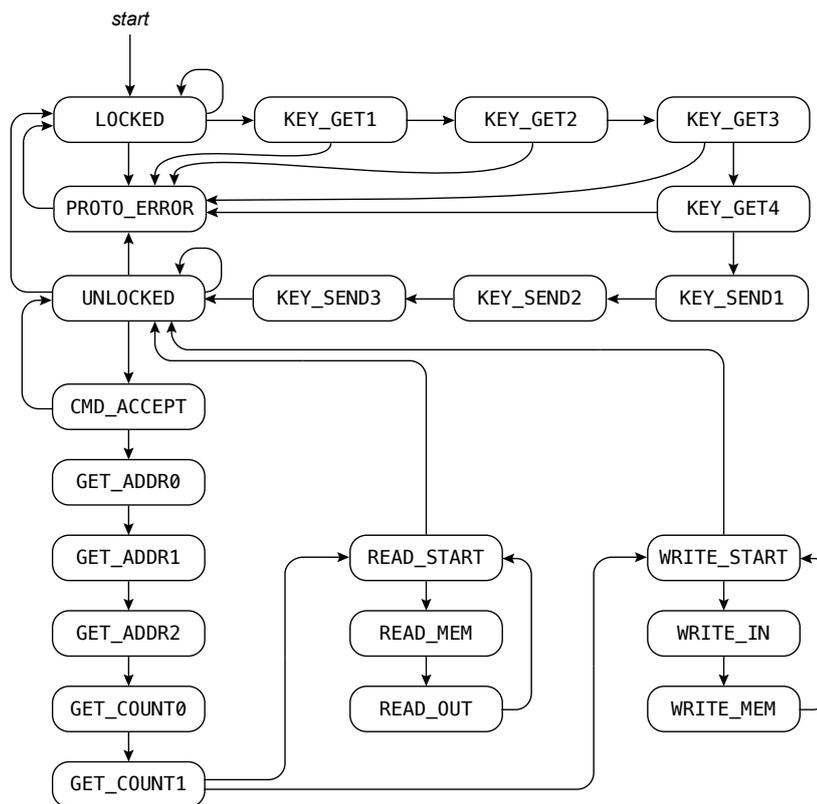


Figure 3.3: Control FSM state diagram

Internally, the personal computer interface component is made up of a number of registers which temporarily store data and an FSM which controls the operation of the component. The state diagram of the control FSM is shown in Figure 3.3.

3.3 Device Subsystem

The device subsystem consists of four device controller circuits which are responsible for operating I/O devices present on the board. As mentioned in Chapter 2.1, devices can be accessed from SIC/XE programs using instructions `WD` and `RD`. These instructions allow one byte of data to be transferred to or from a device at a given 8 bit wide *device address*. The device address space is separate from the memory address space.

device address	type	description
0x02	input	switches
0x03	input	buttons
0x04	input	PS/2
0x05	output	LEDs
0x06	output	seven-segment display mode
0x07	output	seven-segment hex right half
0x08	output	seven-segment hex left half
0x09	output	seven-segment digit 0
0x0a	output	seven-segment digit 1
0x0b	output	seven-segment digit 2
0x0c	output	seven-segment digit 3
0x0d	output	VGA row
0x0e	output	VGA column
0x0f	output	VGA color

Table 3.2: Device address bindings

Table 3.2 shows device address assignments on our system. A single device controller may be assigned multiple device addresses which control different aspects of the device it controls. Accessing a device address not present in Table 3.2 does not result in an error but is simply ignored. Input devices will request an interrupt whenever they receive new input, which will result in interrupts being triggered if they are enabled. Details on how to use devices in programs and descriptions of their respective device controllers are given in the following chapters.

3.3.1 Seven-Segment Display Controller

The Nexys 2 board features a four-digit seven-segment display [3]. It is controlled by the seven segment display controller circuit (see Figure 3.1). For convenience, the controller supports two modes of display operation. In the *hexadecimal* operating mode, a two byte value is displayed in hexadecimal representation across all four digits of the display. An alternative *direct* operating mode is provided in case full manual control of individual display segments is needed.

The controller allows the operating mode to be set individually for the left and right halves of the display. The mode can be set by writing a byte to device address 0x06. The lower two bits of the value written determine the modes of the left and right halves of the display. For example, the user might choose to operate the left half (digits 2 and 3) of the display in hexadecimal mode, while the right half of the display (digits 0 and 1) operates in direct mode. This arrangement can be achieved by writing value 0x03 to device address 0x06 (bit 1 is set, indicating hexadecimal mode, while bit 0 is not set, indicating direct mode).

The display controller refreshes the value on the display from a set of 6 internal 8 bit wide control registers. These registers can be set by writing to device addresses 0x07 through 0x0c, as shown in Table 3.2. When the display operates in hexadecimal mode it displays values from control registers at addresses 0x07 (right half) and 0x08 (left half). Similarly, it displays values

from registers at addresses 0x09 through 0x0c when operating in direct mode; each control register corresponds to one digit on the display. In direct mode, display segments can be individually turned on or off. Figure 3.4 shows the mapping of display segments to bits in the control register corresponding to the digit.

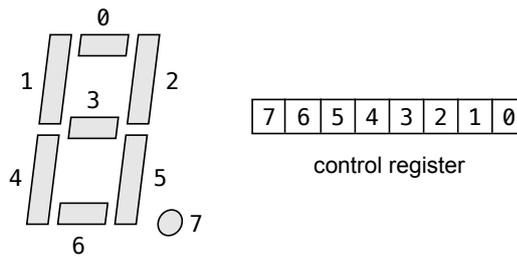


Figure 3.4: Control register bits and display digit segments

In addition to the two normal modes of operation described above, there are two special modes of operation which are activated in certain circumstances. If the processor is suspended, “STOP” is displayed on the display. The display goes back to displaying its previous value after the processor resumes execution. Likewise, “Err” is displayed on the display if the processor encounters an error. These special states were implemented to make it easy for the user to see the current state of the system.

3.3.2 VGA Controller

The VGA controller draws a 40×30 pixel 256 color image from an internal frame-buffer to the VGA output port. The actual output resolution is 640 pixels \times 480 pixels — each pixel from the frame-buffer is drawn as a 16×16 square to the screen. Pixel colors are represented by 8 bit values, which are interpreted as $RRRGGGBB_2$ (3 bits for red, 3 bits for green and 2 bits for blue components).

The number of pixels in the frame-buffer exceeds the number of available device addresses, which means pixels can not be mapped into the device

address space. To overcome this problem, the VGA controller has 2 internal control registers: the row register and the column register. They can be set by writing to device addresses 0x0d (row) and 0x0e (column). Writing to device address 0x0f sets the color of the pixel at the position determined by the current values of the row and column control registers. This makes it possible to set the color of any pixel in the frame-buffer by first setting the row and column control registers to the desired coordinates, and then writing the desired color to device address 0x0f.

3.3.3 PS/2 Controller

The PS/2 controller is responsible for the PS/2 port, which allows a keyboard to be connected to the board. As keys are pressed on the keyboard, the keyboard sends 8 bit *scan codes* to the board which are decoded by the PS/2 controller [2].

The controller has an internal control register, which stores the value of the last scan code received from the keyboard. The value of this register can be accessed by reading from device address 0x04. Every time a scan code is decoded, the controller makes an interrupt request.

3.3.4 General I/O Controller

The general I/O controller is a simple circuit that controls LEDs, switches and buttons. There are 8 LEDs present on the board, which display the value of an internal 8 bit register which is part of the controller. The value of this register can be set by writing to device address 0x05.

In addition to LEDs, there are also 8 switches and 4 buttons present on the board. Two of the buttons have special functions (reset and start/stop toggle), which is why only the remaining two buttons are available for general purpose use. The current state of switches can be read as an 8 bit value from device address 0x02. Similarly, the current state of buttons can be read from device address 0x3. The controller makes an interrupt request when buttons

are pressed and when the state of switches changes.

To prevent misreads, each switch and button has a small de-bouncer circuit which prevents rapid state changes. Changes are registered only when the button remains in one state for a certain period of time.

3.4 Central Processing Unit

The CPU is the most important component of the system. It carries out programs stored in main memory and performs basic arithmetic, logical and input/output operations [7]. The processor is non-pipelined, which means it executes only one instruction at a time. Instructions are carried out sequentially — the processor starts executing the next instruction only when all steps of the current instruction have been completed.

Figure 3.1 shows how the processor is connected to other components of the system. It is connected to the memory controller with an address bus, input data bus and output data bus. The address bus is 20 bits wide, while data buses are both 8 bits wide. Accessing operands longer than one byte thus requires multiple memory operations. The processor uses three control signals to request memory operations (not shown in Figure 3.1). The processor requests a memory operation by activating either the *read* or *write* control signal. It then waits until the memory controller activates the *done* signal, which indicates the operation has completed. As mentioned in Chapter 3.1, memory operations take 5 clock cycles.

To facilitate input/output operations, the processor is connected to the device subsystem with a port id bus, input data bus and output data bus. All three buses are 8 bits wide. The processor uses two control signals to request input/output operations (not shown in Figure 3.1). Input/output operations take two clock cycles to complete.

Three control signals control the operation of the processor itself. The *reset* signal clears the internal state of the processor. When the signal is released, the processor continues execution from address 0. The *enable* signal

can be used to temporarily suspend the operation of the processor. When this signal becomes inactive, the processor completes the current instruction and stops fetching new instructions until the signal is active again. The *interrupt* signal is used to perform interrupt requests. If this signal is active while interrupts are enabled (register I has value 1), an interrupt is triggered.

On most computers, an error in the program (for example a malformed instruction) causes an interrupt. This makes it possible to recover from the error in the interrupt handler, allowing the computer to continue functioning. However, if there is no interrupt handler implemented, the processor may immediately encounter another error at the location where it was expecting the interrupt handler, resulting in an endless loop of errors. If this happens, the system simply stops responding and there is no clear indication about what caused the problem. This is why we decided to implement a different mechanism for handling errors in our SIC/XE processor. When the processor encounters an error in the program, it stops fetching new program instructions and activates the *error* signal. This signal is connected to the seven segment display, which displays an error message to the user. The processor has to be reset before program execution can continue.

3.4.1 Datapath

The processor consists of two main parts: the datapath and the control unit [4]. All arithmetic and logical operations required to carry out instructions are performed within the datapath. A diagram showing its structure is shown in Figure 3.5. The datapath is composed of registers, multiplexers and the ALU (arithmetic logic unit). It is connected to the control unit with a number of control signals (not shown on Figure 3.5), which allow the control unit to control its operation.

The ALU is a combinational circuit which can perform basic arithmetic operations like addition, subtraction and multiplication, as well as logical operations. The ALU is used to carry out arithmetic and logical instructions, as well as to perform calculations on addresses; for example incrementing the

program counter and calculating the target address for PC-relative, indexed and base addressing modes. The ALU operates on two 24 bits wide operands, which are selected by two multiplexers from a range of internal registers. The result bus is wired to a number of registers so the result can be stored.

The register block contains the main six architectural registers. In addition to architectural registers, the datapath contains several hidden registers which are used to temporarily hold values while instructions are executed. The **RES** register temporarily holds the result of ALU operations before it is stored to one of the architectural registers. Temporary registers **T1**, **T2** and **T3** are used to temporarily store operands for certain operations. The **TARGET** register is used for calculating the target address for instructions with PC-relative, indexed or base addressing. The memory data register **MEM** is used to hold data that is read from memory or must be written to memory. Similarly, the device data register **DEV** is used to hold data sent to or received from devices. Instructions are temporarily stored in the instruction (**INSN**) register as they are read from memory, so they can be decoded and executed.

As shown in Figure 3.5, the memory bus and the device bus are connected to the datapath. The memory address bus is connected to a multiplexer, which is used to select either the program counter or the target address as the memory address. The program counter is used when reading instructions, while the target address is selected when accessing memory operands. The port id bus is connected to the lower eight bits of the target address, so it can be used as the device address. The memory output bus is only one byte wide, which is why a multiplexer is used to select which byte of the **MEM** register should be connected to it.

3.4.2 Control Unit

The processor performs instructions in multiple steps. The exact steps required differ from instruction to instruction, but most instructions consist of the following main operations: fetching the instruction from memory, decoding the instruction, performing calculations and/or accessing memory and

storing results back to a register. The datapath has all the components necessary to carry out all individual instruction steps. However, it is a passive component — it only performs operations in accordance to control signals it receives [4]. It can only carry out instruction steps correctly if it gets the right control signals from the control unit.

The control unit is implemented as an FSM, which provides the necessary control signals to the datapath. Combinations of control signals required to perform instruction steps, as well as the correct order of steps needed to execute instructions, are both encoded in the control unit FSM. Because SIC/XE has many instruction formats and addressing modes, the FSM is quite complex — it has more than 50 states.

Chapter 4

SIC/XE System Software

In this chapter we present a suite of system software utilities that we developed for use with the FPGA based SIC/XE system described in the previous chapter. The suite consists of the following four programs: simulator, assembler, linker and FPGA interface utility.

The utilities were developed in C++ programming language for the Linux operating system, but it should be relatively easy to port them to other UNIX-like operating systems. We used the CMake build system for build automation and Google's C++ Testing Framework for unit testing.

4.1 Simulator

The simulator is a software utility which emulates (imitates) a SIC/XE computer system. It can be used to run and debug SIC/XE programs on a personal computer. Our simulator supports all features available on the SIC/XE processor from Chapter 3. It also has two additional features not available on the FPGA system: support for floating point operations and integer division instructions (`DIV` and `DIVR`). Hardware devices described in Chapter 3 (buttons, switches...) are not simulated. I/O instructions are supported, but are implemented as reading and writing to files.

The simulator has two main components: a SIC/XE virtual machine and

an interactive console interface. There are two versions of the simulator — the `sicvm` tool includes only the virtual machine without the interactive console interface. The `sicsim` tool includes both components and is intended to be used as an interactive debugger. An example `sicsim` session is given below.

```

1  sicsim> step count=5
2  000133  ac 10          RMO          X, A
3  000135  29 00 00      COMP         #0
4  000138  33 20 13      JEQ          19 + PC
5  00013b  01 00 01      LDA          #1
6  00013e  94 01          SUBR         A, X
7  sicsim> cpu ?
8  Menu types: M - submenu, C - command
9  TYPE      NAME          PARAMS
10 C         print
11 C         set           register= value=
12 sicsim> cpu print ?
13 Print the CPU state (all registers).
14 sicsim> cpu print
15 REGISTER   HEX          UNSIGNED   SIGNED    SPECIAL
16 PC         000140      320
17 A          000001      1         1
18 X          000004      4         4
19 L          000143      323       323
20 B          000000      0         0
21 S          000030      48        48
22 T          000000      0         0
23 F          0000000000000
24 CC         2
25 I          0
26 IL         000000      0
27 ICC        0
28 sicsim> start
29 00012d          Breakpoint loop_end
30 Number of instructions executed: 810
31 sicsim> watchlist print
32 NUMBER  NAME      ADDRESS  VALUE
33 0       foo      0001a2  0001ba  442      442
34 1       bar      001000  00      0
35 sicsim> cpu set register=A value=0x1234
36 sicsim> step
37 00012d  3f 2f d0      J          -48 + PC
38 sicsim> step
39 000100  0e 20 9f      STA         @[159 + PC]
40 sicsim> step
41 000103  4b 20 78      JSUB        120 + PC
42 sicsim>

```

The user controls the simulator by entering commands. Commands consist of a number of *command words*, which are followed by a list of *arguments*. Command words are hierarchically organized in a tree of menus. The help operator (“?”) can be used to find out more information about a command. For example, entering “cpu ?” displays which commands are available in the “cpu” sub-menu. Entering “cpu print ?” displays a short description of the command, because “cpu print” is not a sub-menu but a complete command (see the example on the previous page). Some commands require additional input from the user, which is given in the form of arguments. Arguments are entered after command words in the following format: *parameter name = value*. The order of arguments is not important.

Some command words and parameter names are quite long. For convenience, the simulator includes a prefix matching system. Command words and parameter names do not need to be entered exactly — only the first few letters which uniquely identify a command word or parameter name need to be entered. In case the entered prefix is too short, the simulator will display an error explaining that the input is ambiguous.

The basic commands for operating the simulator are “step” and “start”, which allow the user to step through the program or let it run until it encounters an error or a breakpoint. Breakpoints can be set in the “breakpoint” sub-menu. Sub-menus “cpu” and “memory” contain commands for viewing and modifying the state of the virtual machine’s processor and memory.

A disassembler facility is built into the simulator. The disassembler translates machine code instructions into a human readable assembly language. It supports two modes of operation: automatic mode and manual mode. In manual mode, the user requests code at a specified address to be disassembled. When automatic mode is enabled, instructions are disassembled as they are executed when stepping through the program. Disassembler features are available in the “disassembler” sub-menu.

When debugging a program, it is often necessary to monitor values at certain memory locations, but manually entering the same addresses over

and over is tedious. This is why the simulator includes a memory watch-list feature. Memory locations added to the watch-list can be easily inspected and modified using commands in the “watchlist” sub-menu. The user can specify the data type (word, byte, float) of each watch-list entry, which allows the simulator to correctly interpret and display the value stored at the memory location.

4.2 Assembler

The assembler is a program that translates code from SIC/XE assembly language to binary machine code. It produces object files that contain machine code together with metadata, which allow linking and program relocation. In addition to object files, the assembler can optionally produce log files. Log files are text files which contain nicely formatted assembly source code annotated with the corresponding output machine code, as well as additional information useful for debugging.

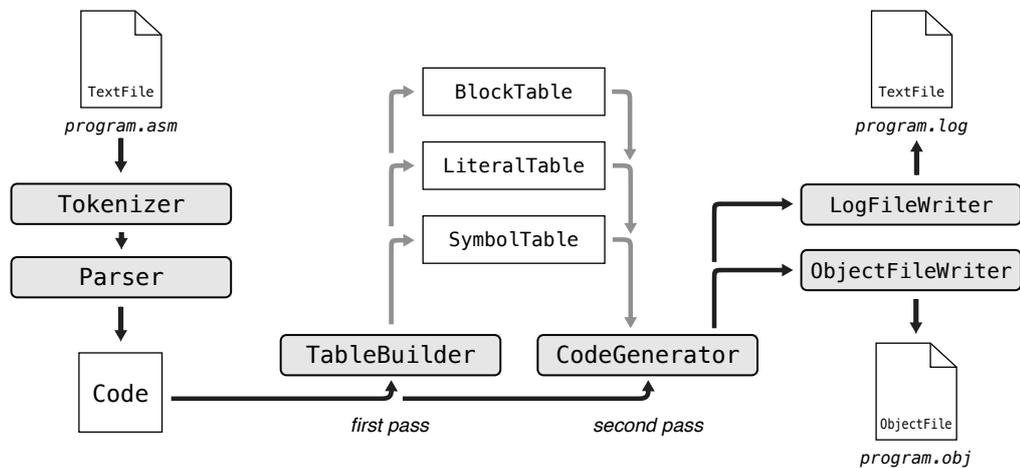


Figure 4.1: Assembler design overview

Figure 4.1 illustrates the internal design of the assembler. It has a modular design — each step of the assembly process is implemented within a C++

class. The assembler first reads an input source file and converts it into an internal in-memory representation, which is easier to process than the raw source file. This step is implemented in classes `Tokenizer` and `Parser`. The result is a `Code` class instance, which contains the in-memory representation of the parsed assembly source code.

Our assembler is a two-pass assembler. In the first pass, it analyzes the code and produces a symbol table, block table and literal table. Information from these tables is used in the second pass, in which output machine code is generated. The first pass is implemented in the `TableBuilder` class. The tables it generates are represented by classes `BlockTable`, `LiteralTable` and `SymbolTable`. The second pass is implemented in the `CodeGenerator` class.

As mentioned before, the assembler needs to write out machine code in different formats (object file and log file). When the `CodeGenerator` class generates code, it sends code fragments as well as other metadata to a set of registered classes, which implement the `CodeOutputWriter` interface. We implemented two such classes — the `ObjectFileWriter` class creates object files and the `LogFileWriter` creates log files. This mechanism is good because code generation logic is separate from logic related to a certain output file format. It also makes it easy to add new output file formats if they are needed in the future.

4.2.1 Assembly Language

Before getting into more details of how the assembler works, we will first briefly introduce the SIC/XE assembly language. An example assembly program is given below.

```

1  SAMPLE      START      0
2  . This is a sample assembly program
3              LDT        #SIZE
4              CLEAR     X
5              CLEAR     S
6  LOOP       LDA        TABLE, X
7              ADDR     A, S
8              LDA        #3
9              ADDR     A, X
10             COMPR    X, T
11             JLT      LOOP
12             STS      RESULT
13
14  . Endless loop
15  WAIT      J         WAIT
16
17  RESULT    WORD      0
18  TABLE    WORD      12
19             WORD      16
20             WORD      36
21             WORD      10
22  TABEND    EQU       *
23  SIZE      EQU       TABEND - TABLE
24
25             END      SAMPLE

```

Assembly programs are made up of instructions and directives — each non-empty line of the source file represents either an instruction or a directive. Source code can be annotated with comments, which are marked with the dot (“.”) character. All characters in a line following a dot character are ignored.

Every line of source code consists of three parts: *label*, *mnemonic* and *operands*. The mnemonic is a short symbolic name associated either with an instruction or with a directive. Even though directives and instructions share a similar syntax, they have a different purpose. Instructions are directly translated to machine code instructions and are executed when the program

is run. On the other hand, directives are commands for the assembler itself and are performed at assembly time [6]. A list of available directives and their mnemonics is shown in Table 4.1.

mnemonic	description
START	start of program
END	end of program, emit pending literals
ORG	begin new section
EQU	set symbol value
USE	switch to block
LORG	emit pending literals
BASE	set base address and enable base addressing
NOBASE	disable base addressing
EXTDEF	export symbols
EXTREF	import symbols
BYTE	initialize memory (byte)
WORD	initialize memory (word)
RESB	reserve memory (bytes)
RESW	reserve memory (words)

Table 4.1: Assembler directives

A label can be placed before mnemonics of instructions and data directives (`BYTE`, `WORD`, `RESB` and `RESW`) to associate the address of the instruction (or data variable in the case of data directives) with an arbitrary symbol name. The memory location can then be referred to from other parts of the program by its symbol name. Our assembler does not impose a length limit on internal symbol names. However, external symbol names are limited to 6 characters, due to restrictions of the object file format (see Chapter 2.6). Symbols can be exported with directive `EXTDEF` and imported with directive `EXTREF`. Addresses of imported symbols are unknown to the assembler. Their addresses are resolved at link time by the linker, which uses information from

the object file's modification sections to correctly modify object code where necessary.

directive(s)	operands
LORG, NOBASE	(none)
USE	[block name]
EXTDEF, EXTREF	symbol name [, symbol name...]
START, END, ORG, EQU, BASE	expression
BYTE, WORD, RESB, RESW	expression

Table 4.2: Directive operand syntax

Most instructions and directives require operands after the mnemonic. The syntax of operands varies depending on the instruction or mnemonic used. Table 4.2 shows the expected syntax of directive operands. Most directives expect an *expression* operand. Expression operands are simple mathematical expressions which consist of integer constants, symbol names and operators +, -, * and /.

format	operands
1	(none)
2	register, register
2	register, integer
2	register
SIC/3/4	(none)
SIC/3/4	[# or @ or =] expression [, X]

Table 4.3: Instruction operand syntax

Instruction operands are encoded into fields of the machine code instruction when instructions are translated. This is why the assembly syntax of instruction operands depends on the format of the machine instruction (instruction formats are discussed in Chapter 2.3). Table 4.3 shows all six

possible types of instruction operand syntax.

Instructions that access memory operands (formats SIC/3/4) expect an expression operand, which is evaluated to get the memory address. The expression can be prefixed with one of the addressing operators #, @ or = to select the addressing mode as shown in Table 4.4. By default, the assembler emits instructions in format 3. Format 4 can be selected by prepending + to the instruction mnemonic. Format SIC is selected automatically when the memory address is too large to fit in a format 3 instruction and simple addressing is used.

operator	addressing mode
(none)	simple
#	immediate
@	indirect
=	literal pool

Table 4.4: Addressing operators

Literal pool addressing is not a real addressing mode, but a pseudo addressing mode provided by the assembler. When this addressing mode is used, the assembler reserves a data variable in memory called a *literal*, which is initialized with the value of the expression. The literal is then addressed using direct addressing. From the programmer’s perspective, this addressing mode behaves similarly to immediate addressing.

Appending “, X” to the expression enables indexed addressing. The assembler automatically selects base relative addressing or PC-relative addressing when possible. To use base addressing, the programmer needs to inform the assembler of the run-time value of the base register. This is done using the **BASE** directive. The assembler assumes that the base register has the specified value until it is changed again with the **BASE** directive or until base addressing is disabled with the **NOBASE** directive. The programmer must ensure that the actual value of the base register at run time matches the address

given to the assembler, otherwise the code will not execute correctly.

Instructions with direct addressing present an additional challenge for the assembler, because they need to be updated if the program is relocated. However, updating *all* instructions with direct addressing is not an option, because in some cases the programmer may have actually intended to access a fixed memory location. The assembler needs to figure out which instructions really need to be updated, and then write this information to the output object file in the form of modification records. This is why the assembler differentiates between two different kinds of expressions and symbols: *relative* and *absolute*. The value of absolute expressions does not change when the program is relocated, while relative expressions need to be modified by adding the new start address of the program to the value of the expression. When direct addressing is used and the expression operand is relative, the assembler writes a modification to the object file.

4.2.2 Tokenizer and Parser

Our assembler processes code in two passes. However, instead of processing the raw source file twice, the source file is first converted to an internal in-memory representation, which is better suited for processing by later stages of the assembler. This conversion is performed by classes `Tokenizer` and `Parser` (see Figure 4.1).

The resulting parsed code is stored in a `Code` class instance, which contains a list of `Node` classes. Each node in the list corresponds to a line of code from the source file. `Node` is an abstract class — lines of code are represented by various `Node` subclasses, depending on the contents of the line. The inheritance tree of the family of `Node` classes is shown in Figure 4.2. We implemented the *visitor pattern* on `Node` classes to make it easier to iterate over the data structure. The visitor pattern is well suited for this situation, because it allows the algorithm (in our case different assembler passes) to be well separated from the data structure on which it operates [9]. This separation makes code more readable and easier to maintain.

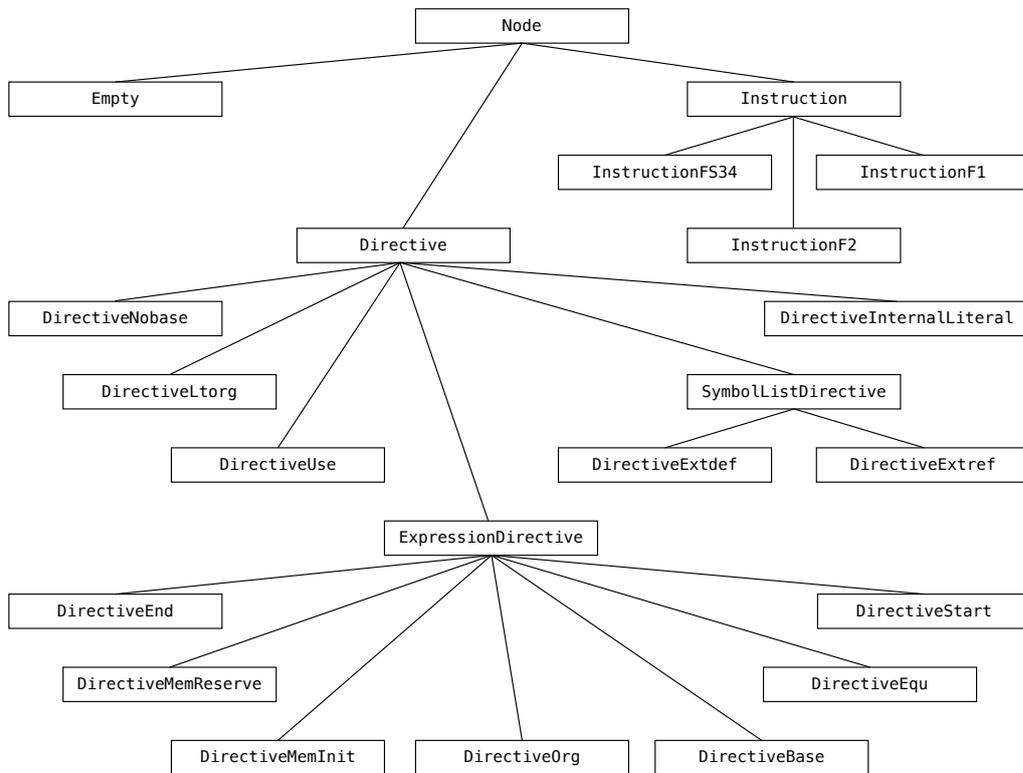


Figure 4.2: Node class inheritance tree

The conversion of source code to the in-memory representation happens in two steps. Every line of the program is first converted to a list of tokens by a simple lexical analyzer, which is implemented in the `Tokenizer` class. Even though there are many tools available (for example Flex) which can automatically generate lexical analyzers, we used a hand written one for our assembler instead. We decided not to use a lexical analyzer generator because we wanted to avoid dependencies on external tools and libraries.

Tokens produced by the `Tokenizer` class are then processed by a simple top-down recursive descent parser implemented in the `Parser` class. For every line of the input source file, the parser generates a `Node` subclass instance which represents the contents of the line. Finally, generated node classes are stored in a `Code` class instance, which represents the whole input program.

4.2.3 First Pass

After the input file has been converted to the in-memory representation, the first assembler pass can be performed. Machine code can not yet be generated in the first pass, because certain things like symbol addresses are not yet known. The main goal of the first assembler pass is to calculate all the information necessary to allow machine code to be generated in the second pass.

The first assembler pass is implemented in the `TableBuilder` class. During this pass, three tables are generated: symbol table, block table and literal. The symbol table is implemented in the `SymbolTable` class and is used to store information about symbols. In addition to the symbol names and their addresses, other details like whether or not the symbol is undefined, exported or imported, are stored. If the symbol is defined inside a program block, the corresponding block number is also recorded. Information about program blocks is stored in the block table. Every block has a block number and, except for the default block, also a unique name. Directive `USE` is used to switch between program blocks. Sizes of blocks and their start addresses are calculated during the first assembler pass. At the end of the pass, when start addresses of blocks are known, final addresses of symbols defined in program blocks can also be calculated and are updated in the symbol table.

The literal table, implemented in the `LiteralTable` class, is used to keep information about literals. Every time literal pool addressing is used, a literal is added to the literal table. Literals also get a symbol name and an entry in the symbol table to keep track of their address. The literal table de-duplicates literals with the same value whenever possible to conserve space. When a `LTOrg` directive or the end of the program is encountered, all pending literals present in the literal table are emitted. A special `LiteralInternalDirective` node is inserted into the code for each emitted literal, which marks that the literal should be written out during the second assembler pass.

4.2.4 Second Pass

In the second assembler pass, code is processed again for a second time. Using information generated during the first pass, machine code is generated. The second pass is implemented in the `CodeGenerator` class.

Generating code for instructions in format 1 and format 2 is relatively straightforward. Instructions in formats SIC/3/4 present more of a challenge. Using data from the symbol table, expressions can be resolved to actual memory addresses. Taking into account various constraints, the `CodeGenerator` class then selects the most appropriate addressing mode for the instruction. If the expression is relative or it references external symbols, modification records also need to be generated. Literals and data directives are written out in this pass as well.

The assembler can output generated machine code to object files and log files. For illustration, a log file produced from the example program from Chapter 4.2.1 is given on the next page. The log file contains an annotated listing of the source code, along with a printout of the block table and symbol table. The literal table is not present because literal pool addressing is never used in the program.

The log file produced from the example program from Chapter 4.2.1 is given below.

```

1 ***** CODE *****
2 000000:          SAMPLE          START          0
3 000000:          . This is a sample assembly program
4 000000: 75 00 0c          LDT          #SIZE
5 000003: b4 10          CLEAR          X
6 000005: b4 40          CLEAR          S
7 000007: 03 a0 15      LOOP          LDA          TABLE, X
8 00000a: 90 04          ADDR          A, S
9 00000c: 01 00 03          LDA          #3
10 00000f: 90 01          ADDR          A, X
11 000011: a0 15          COMPR         X, T
12 000013: 3b 2f f1          JLT          LOOP
13 000016: 7f 20 03          STS          RESULT
14 000019:
15 000019:          . Endless loop
16 000019: 3f 2f fd      WAIT          J          WAIT
17 00001c:
18 00001c: 00 00 00      RESULT         WORD          0
19 00001f: 00 00 0c      TABLE         WORD          12
20 000022: 00 00 10          WORD          16
21 000025: 00 00 24          WORD          36
22 000028: 00 00 0a          WORD          10
23 00002b:          TABEND         EQU          *
24 00002b:          SIZE          EQU          TABEND - TABLE
25 00002b:
26 00002b:          END          SAMPLE
27 ***** BLOCKS *****
28 NAME          START          SIZE
29 <default>      000000  0          43
30 ***** SYMBOLS *****
31 NAME          TYPE  VALUE          BLOCK
32 LOOP          IR    000007  7          <default>  7
33 RESULT        IR    00001c  28         <default>  28
34 SAMPLE        X    000000  0          <default>  0
35 SIZE          IA    00000c  12         <default>  43
36 TABEND        IR    00002b  43         <default>  31
37 TABLE        IR    00001f  31         <default>  31
38 WAIT          IR    000019  25         <default>  25

```

4.3 Linker

A linker is a program for combining multiple object files into a single executable program [8]. Larger programs are made up of many subroutines, which are usually arranged in multiple assembly source files. Source files that make up a program are individually assembled into object files, which are then linked together to form the completed program. The same program could be created by placing all subroutines in a single large assembly source file, but there are many advantages to using multiple source files. When changes are made to the program, only the changed source files need to be re-assembled, which can shorten the time required to build the program. Programs made up of multiple source files can be organized better, which makes programming easier. Subroutines defined in a source file can also easily be reused when writing new programs by simply linking the corresponding object file into the new program.

Linked programs produced by our linker utility are written out in object file format. Code from input object files is arranged into the address space of the output program as illustrated in Figure 4.3.

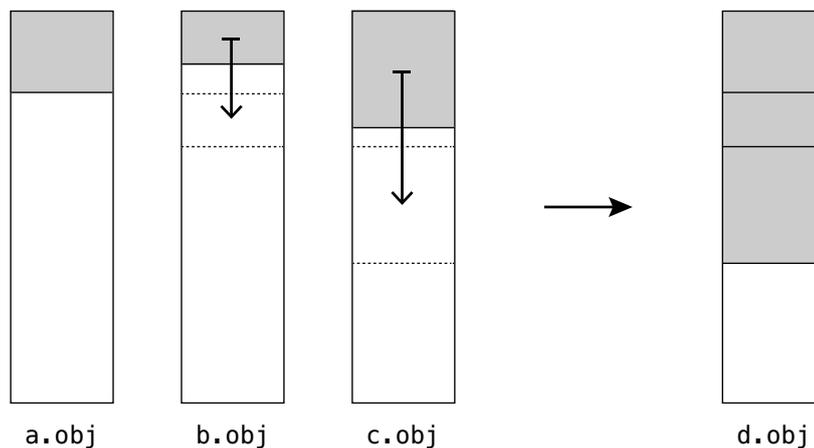


Figure 4.3: Linker arranges object files

In addition to linking code, our linker writes out modification records to

the output object file. This makes it possible to relocate the program, for example by a loader when the program is loaded into memory. The linker also supports a *partial* linking mode, which allows undefined symbols to remain in the output program. Normally, the linker would display an error message if undefined symbols were detected. However, in this mode the linker resolves only the symbols it can, and writes the remaining undefined symbols to the output object file.

Just like the assembler, the linker also operates in two passes. In the first pass, it analyzes information contained in input object files and generates a symbol table and a relocation table. Symbol addresses stored in the symbol table are adjusted to reflect final addresses in the output program's address space. All modification records from input object files are gathered and stored in the relocation table.

In the second pass, the linker copies code sections one by one from input files to the output file. For each code section, it searches the relocation table for modification records whose addresses match the address range of the code section. If any records are found, code is modified as required by the modification records. Symbol addresses are looked up in the symbol table when processing modification records which reference symbol names.

4.4 FPGA Interface Utility

The FPGA interface utility is a small utility program for communicating with our FPGA based SIC/XE system described in Chapter 3. When started, it attempts to establish a connection to the FPGA board through the personal computer's serial port. The utility communicates with the board according to the protocol described in Chapter 3.2. If a connection to the board is made, the user is presented with an interactive console interface (the same kind of interface as in the simulator tool).

Like in the simulator, the contents of the board's system memory can be viewed and modified with commands in the "memory" sub-menu. The

processor can be started and stopped using commands “start” and “stop”. The user can also reset the processor with the “reset” command and trigger interrupts with the “interrupt” command. SIC/XE programs stored in object files on the personal computer’s hard drive can be loaded into the board’s main memory with the “load” command.

Chapter 5

Conclusions

The hardware and software described in this thesis provide a comprehensive platform for developing, testing and running SIC/XE programs. Users can write programs in SIC/XE assembly language on a personal computer and convert them to machine code using the assembler. Larger programs can be split up into multiple object files and then linked together with the linker. In addition to running programs inside a SIC/XE virtual machine on a personal computer, users can also run programs directly on our FPGA based hardware SIC/XE system.

We developed and tested our software utilities for the Linux operating system, but we would also like to add support for other operating systems in the future. Another interesting feature which could be added is a graphical user interface. Because our programs are written in C++, the code could even be directly integrated into an iOS application for an iPad or an iPhone.

There is currently no non-volatile memory available on our FPGA based system. Because of this, the system can not function autonomously when not connected to a personal computer. The Nexys 2 board has a flash memory chip which could be used to add non-volatile storage to the system. However, because the flash chip shares the same bus with the PSRAM memory chip, significant changes to the design would probably be needed to accomplish this.

Bibliography

- [1] Leland L. Beck. *System Software: An Introduction To Systems Programming*. Addison Wesley, 1997.
- [2] Adam Chapweske. *The PS/2 Mouse/Keyboard Protocol*. <http://www.computer-engineering.org/ps2protocol>.
- [3] Digilent Inc. *Nexys 2 Reference Manual*. http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf.
- [4] Dušan Kodek. *Arhitektura in organizacija računalniških sistemov*. Bi-Tim, 2008.
- [5] Micron Technology Inc. *128Mb CellularRAM™ MT45W8MW16BGX Data Sheet*.
- [6] Wikipedia. Assembly language — wikipedia, the free encyclopedia, 2014. [Online; accessed 30-June-2014].
- [7] Wikipedia. Central processing unit — wikipedia, the free encyclopedia, 2014. [Online; accessed 25-June-2014].
- [8] Wikipedia. Linker (computing) — wikipedia, the free encyclopedia, 2014. [Online; accessed 30-June-2014].
- [9] Wikipedia. Visitor pattern — wikipedia, the free encyclopedia, 2014. [Online; accessed 30-June-2014].