RMIT International University Vietnam
Centre of Technology

Microcontroller Emulation

COSC2131 - Programming using C++
Assignment 1 – 2012B

This assignment gives a fundamental understanding of how emulation works. Emulator programs or emulator modules have been around since the first home computers were released. At the time they were used to be able to run programs that were designed for other platforms. Nowadays, emulators are used for many purposes: to emulate old computer systems and arcade machines (e.g. the MAME and MESS projects); to help with hardware backward compatibility (PS and Xbox360 support old software through emulation); to help with software backward compatibility (Windows 7 supports old software though Windows XP on x86 emulation); and to run multiple virtual servers on large mainframe systems to reduce operating costs (virtualisation.)

Overview

The aim of this assignment is to emulate a monitor program and a set of simplified microcontrollers. The program is required to be implemented in the C++ programming language, and the emphasis is on using console I/O and basic OOP concepts. The platform required to be used is *nix, and `make` must be used to build the project. Other compilers are not allowed.

A microcontroller is a chip that is used to perform simple functions, such as input, processing and output. Microcontrollers are found in many household appliances ranging from alarm clocks to videocassette recorders, from car engine management units to washing machines, from microwave ovens to remote-control air conditioning units.

The microcontroller is programmable – a programmer loads a program into its memory, and the microcontroller executes that program. Input and output is handled by modifying addresses within memory. Both the program and data for the microcontroller are stored within the same block of memory. This is the common Von Neumann architecture that you find in all microchip-controlled devices.

When designing a program, a programmer for a microcontroller had to check and implement the design by hand. Often, this was done with pen and paper, and later with a piece of software known as a monitor. The monitor allowed the programmer to modify sections of memory directly, to display everything in memory, or to execute the contents of memory as if the contents were a program.

Submission and Plagiarism Notice

Submission

Submissions comprise a report in HTML format, C++ code, and a `Makefile`. A softcopy of your work must be submitted to Blackboard by the deadline. You are required to adhere to the submission standards published on Blackboard.

A hardcopy of the Statement of Authorship must be submitted in the drop-box in front of room SGS 1.4.04.

Plagiarism notice

This is an individual assignment. While discussion about program design, algorithms & data structures, C++ features and quality issues is encouraged, exchange of files and/or exchange of code are regarded plagiarism. Refer to the document "Referencing and Plagiarism in Programming Assignments" for details. Consult with your lecturer for advice on this issue if necessary.

Any copying of any code from any source is strictly forbidden. The submitted work must be 100% your own.

Description of the Program

Development of the program is divided into 4 distinct phases. This helps to evolve the program in small steps, and test each module before moving on to the next.

Project setup and Makefile

Before starting the project, create a subdirectory that will contain all files related to this project. Then, create a `Makefile` with targets:

`all` - this is the default (first) target, it is phony, and builds all targets in the `Makefile`
`main` - the actual emulator start-up code
`clean` - phony target that removes `a.out core *.o *.s *~ main` (etc.)
targets for each module, e.g. `Monitor.o`, `MopsR500.o`, etc.

You should add variables to control compiler behaviour, e.g.:
```
CXXFLAGS = -Wall -g
```

Phase 1 - UI
This phase involves obtaining input and output from the user. The interface will be entirely command-line based, with input read from the user, and displayed directly to the screen.

The prompt for the user must be "> " (>-symbol, followed by a space.) The interface must handle both upper and lowercase commands. All error messages must be displayed to `stderr`, i.e. use `cerr`.

At this stage, the UI only displays a message when a command is typed, i.e. the actual commands are not implemented yet.

The UI should display all output in hexadecimal format. This format is used because it represents a group of exactly 4 bits. All input should be in hexadecimal as well. Decimal is not used. To output and input numbers in hexadecimal format, pass the `hex` manipulator to `cout` and `cin`, e.g.:

```
std::cout << std::hex << 85 << std::endl; // prints 55
```

The output must be padded with `0`s to make pairs of digits (for 1 byte) or 4 digits (for 2 bytes and addresses.)

The following commands must be supported:

Cmd	Description of command
c	Connect to microcontroller ("create") The connect command prompts the user to enter the type of microcontroller they wish to connect to. Valid types of microcontroller are "R500", "PIC32F42", and"34HC22". The response to the user must be the type followed by "selected".
d	Display all memory This command will be implemented in Phase 3.
e	Execute from current PC This command will be implemented in Phase 4.
g	Execution from a specific location ("go") This command asks the user for the location of memory where execution should begin. This command will be implemented in Phase 4.
h	Display help This command should display a list of all commands to the user (this table.) The only requirement for the output format of this command is that it must be neat.
l	Look at memory Look at a specific memory location. This command will be implemented in Phases 2 and 3.
m	Modify memory Modify a specific memory location. This command will be implemented in Phases 2 and 3.
r	Reset microcontroller

	This command resets the microcontroller to an initial start state. What exactly this is depends on the type of microcontroller. This will be implemented in Phases 2 and 3.
s	Display PC and registers ("status") This command will query and display microcontroller status. It will be implemented in Phases 2 and 3.
q	Quit the program The program terminates.

Phase 2 - Microcontrollers

Phase 2 of the assignment requires a model for the microcontrollers, to be implemented using classes. The types of microcontroller are:

1. Mops R500. For detailed information, see Appendix A.
2. Macrochip PIC32F42. For detailed information, see Appendix B.
3. Rotomola 34HC22. For detailed information, see Appendix C.

Each microcontroller has a number of common features:

- The Program Counter (PC.) The PC is an internal CPU register that points to the memory location that contains the next instruction. When a CPU is reset, the PC is set to a particular location where the program starts.
- Each microcontroller contains a small memory, and microcontrollers allow users to read and to modify memory. The memory is a dynamically allocated array of `unsigned char`, which is managed carefully by constructors, destructor, and assignment operator. If a value outside of the memory is being read, 0 should be returned (e.g. there is no memory bank that contains a value.) If data is written to a value outside of memory, nothing happens (e.g. the data is written to a non-existing memory bank.)
- A microcontroller is able to be reset, and each Appendix details what occurs when that particular microcontroller is reset.
- A microcontroller can also execute. It is important to realise that each microcontroller executes (and resets) differently, and this fact must be represented within your model. The details of Execution are implemented in a later phase.
- A number of reporting facilities, for example to query the PC and registers. You should implement a `statusString` function that returns the status of the microcontroller (PC + registers) as a string.

The remaining features of each microcontroller are identified within the Appendices.

Phase 3 - Connect UI and Microcontrollers

If the user has not connected to a Microcontroller, no output should be displayed by any command, except for the Connect and the Help commands.

See the Appendices for examples.

Connect to microcontroller

The connect command asks the user for the type of microcontroller to connect to. The prompt is '`type? `' (type?, followed by a space).

The input from the user must be validated. Valid types of microcontroller are "R500", "PIC32F42" and "34HC22". Only those three types are accepted.

Depending on the input, the response to the user must be “R500 selected”, “PIC32F42 selected” or “34HC22 selected”. If an invalid type is entered, the response to the user must be “Invalid type”.

After validating the input, a new microcontroller object must be created, and the microcontroller reset.

Reset Microcontroller
The microcontroller is reset. The response to the user must be “Microcontroller reset”.

Look at Memory
This command asks the user for the address of memory required. The prompt must be ‘`location?`’ (location?, followed by space).

The address input by the user should be validated. If it is less than 0, or greater than the memory provided by the microcontroller, the command must print “Invalid address” to `cerr`, and return to the user interface.

If the address is valid, the command must print the current value of that address on its own line, in hexadecimal format, and then returning to the user interface.

Modify Memory
This command asks the user for the address of memory they wish to modify. The prompt must be ‘`location?` ’ (location?, followed by a space).

The address input by the user should be validated. If it is less than 0, or greater than the memory provided by the microcontroller, the command must print “Invalid address” to `cerr`, and return to the user interface.

If the address is valid, the command must print “Old value: ” followed by the current value in hexadecimal format, on its own line. After that, it must ask the user ‘`new?` ’ (new?, followed by a space), and then input a number, in hexadecimal format, from keyboard.

As each memory location is a byte size (`unsigned char`), only the lower 8 bits of input should be stored.

Display All Memory
This command displays all locations in memory, 16 columns wide, in hexadecimal format. Address 0 must be the top-most left-most value; Address 1 the top-most, second from-left value, etc.

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
etc
```

On the left-hand side, there should be a column that shows the memory address of the first byte on that line.

Phase 4 - Microcontroller instructions

The purpose of this phase is to emulate the execution process of the microcontrollers. In this phase, you must implement the E and G commands.

Each microcontroller recognizes a particular instruction set (called 'opcodes.') Each opcode has a defined method of execution. The details of the opcodes and their methods of execution are provided for each microcontroller in the Appendices.

It is important to understand the instruction set of the microcontrollers, and to test each operation properly. This can be done by writing short programs, executing them, and analysing the result by looking at the microcontroller status and memory.

When running the E command, the microcontroller assumes that the current memory location pointed to by the PC has an opcode, and begins execution from that point. If the microcontroller was not reset, and the PC does not point to a valid opcode, the microcontroller will crash.

At this moment, the microcontroller is in a fetch-decode-execute cycle. The microcontroller fetches an instruction from memory, decodes its meaning (using a `switch` statement), and then executes the instruction. Execution sometimes implies fetching additional data from memory, and sometimes means writing a result to memory. The PC is updated as part of the instruction.

When running the G command, the user is prompted for an address to begin executing from. The prompt must be '`location?` ' (location?, followed by a space). The address provided must be a valid address – if it is not, the user must be prompted (via `cerr`) with "Invalid address", and the user should be returned to the user interface. If the address is valid, the microcontroller sets the program counter to that address, and begins executing as per the E command.

If the PC goes past the top of memory (i.e. the PC is greater than top of memory), execution must halt – this is termed "running off into the weeds", and the user must be prompted (via `cerr`) with the message "SIGWEED. Program executed past top of memory".

If an invalid opcode is encountered by the microcontroller, the microcontroller must halt execution, and prompt the user with the message "SIGOP. Invalid opcode. Program Counter = " followed by the program counter, displayed in hexadecimal format.

If the microcontroller encounters the halt opcode (as defined for each microcontroller), it must halt execution, and prompt the user (via `cout`) with "Program halted". The microcontroller must not reset at this point. The user must be returned to the user interface.

Extra features

Some marks are allocated to extra features. Examples of extra features are described below.

- A classical problem in software engineering is adding new plug-ins to a program without having to search through all code to find out where and how to plug them in. We have this problem in our emulator as well because there are multiple microcontrollers, and adding new ones requires adding them to the UI by adding code as various places. This is inelegant and makes the code harder to maintain.

Come up with a way to centrally create all microcontrollers, and pass them to the UI as a dynamic data structure. When the UI displays which controllers are available (Help command), it should query the data structure. When it asks the user which controller to create (Connect command), the UI should query the data structure to find out if the microcontroller is available, and create one (Factory pattern?) Adding new microcontrollers still requires re-compiling the code, however, the UI is de-coupled from pluggable microcontrollers, and the code has a central place where microcontroller plug-ins are managed.

- The microcontrollers in this assignment do not have any input/output and therefore do not allow for very interesting programs (barcode scanners do not require console output!)

  Simple CPU based systems often use memory-mapped I/O. This means that if the CPU is writing data to a specific memory location, a component called a memory decoder re-routes that data to I/O circuitry, which in its turn can write the data into video memory. This requires a relatively simple change to your microcontroller's `write_memory` opcode(s).

  In theory, if a video chip has a 40x25 characters display, you could reserve a 1,000 bytes space in memory range 0x1000-0x13e7 (hexadecimal), and re-route memory writes in this range to the corresponding locations on the screen. Space Invaders works similarly.

  More practically, it should be possible to reserve 1 location in memory, let's say 0x1000, and re-route data written to this location directly to `cout`. You can use this to output characters to the console. "Hello, world!" from a barcode scanner!!!

  Implement this for one of the microcontrollers (PIC or Rotamola.) Keep it simple.

- The UI in this assignment does not allow saving/restoring of each microcontroller's state which makes it tedious to load/save programs. You can save the state of a microcontroller relatively easily by writing PC, registers and memory to disk.

  Alternatively, you can make writing and loading programs easier by having your UI support text input from a file. The main idea is, you write a text file that contains all commands that you would normally give to the program. When you start the program, you re-direct the contents of the file to your program using a UNIX pipe, e.g.:

  ```
  ./main < prog.34hc22
  ```

  The lecturer will provide example input files that can be used as test input for your microcontrollers. Using those requires your program to adhere strictly to the UI specification above.

Report & Statement of Authorship

As part of this assignment, a short report must be submitted. The report is submitted as a softcopy together with the source/project files. It does not need to be submitted as hardcopy.

The report is required to be submitted in HTML4+ or XHTML format. It should display properly in any of the major browsers.

Statement of Authorship

The SOA is submitted as a hardcopy only. Submit it in the drop-box in front of room 1.4.04. Without a SOA, an assignment is not officially submitted ☹. Also note that you are officially required to keep a copy of your submission in case the submission gets lost.

Report contents

The report needs to be properly structured in sections and subsection. It needs to be informative and functional. Fanciness does not yield marks (unless you get the program to work in the HTML!!!)

The report should have the following sections:

Introduction – a few lines about what this page is, as well as the title of the program/assignment, date, student name and ID, course code and name, assignment number and name, lecturer name, etc.

How to build/install – how to download and/or build and/or install the program.

Implementation – Identify the files and major modules. Include a Class diagram with the classes and most important members (omit the obvious.) Explain algorithms and data structures if there is anything outside of the assignment description.

Extra features – what did you implement (and you think should be rewarded!) beyond the minimum required.

Problems and Solutions – what were the major difficulties during the project, how did you solve them, and what do you plan to do next project to prevent these problems?

– End of Assignment - Appendices on the following pages –

Appendix A - Mops R500 Microcontroller

Features

- Big endian, 2 byte memory addressing, 1 byte opcodes
- 1024 bytes of memory

Reset

Upon reset, the microcontroller sets the program counter (PC) to location 0. The microcontroller clears all internal memory and initialises it to 0. This provides the programmer with a clean slate.

Instruction Set Summary

Opcode	Action
0x0A	Add Value to Memory The first byte after the opcode is the value that is to be *added* to memory. The second byte after the opcode is the high byte of the address, and the third byte is the low byte of the address. The memory address can be determined by (high byte << 8) | low byte. After execution, the PC points to the fourth byte after the opcode.
0x13	Subtract Value from Memory The format of the instruction is the same as above. The value should be *subtracted* from the memory this time.
0x16	Go to address (always branch) The first byte after the opcode is the high byte of the address. The second byte after the opcode is the low byte of the address. After execution, the PC points to that address.
0x17	Branch relative The program branches to a new location, relative to the current location. The first byte after the opcode is the value that will be added to the PC. The value must be treated as a signed value, i.e.: 128 = -128).
0xFF	Halt opcode Execution stops and the PC is not incremented.

Appendix B - Macrochip PIC32F42

Features

- Big endian, 2 byte memory addressing, 1 byte opcodes
- 1536 bytes of memory
- Special purpose byte-sized register called W

Reset

Upon reset, the program counter (PC) is set to 0. The W register is set to 0.

Instruction Set Summary

Opcode	Action
0x50	Move Value to W The first byte after the opcode is the value to be written to the W register. The PC is set to the address of the second byte after the opcode.
0x51	Move W to memory The first byte after the opcode is the high byte of the memory address, and the second byte is the low byte of the memory address. The memory address can be determined by (high byte << 8) | low byte. The W register's contents are written to this memory address. The PC is set to the address of the third byte after the opcode.
0x5A	Add Value to W The first byte after the opcode is the value to be added to the W register. The PC is set to the address of the second byte after the opcode.
0x5B	Subtract Value from W The first byte after the opcode is the value to be subtracted from the W register. The PC is set to the address of the second byte after the opcode.
0x6E	Goto address (branch always) The first byte after the opcode is the high byte of the address. The second block of memory after the opcode is the low byte of the address. After execution, the PC points to that address.
0x70	Branch if not equal The next value after the opcode is the comparison value. The second block of memory after the opcode is the high byte of the branch target, and the third block of memory is the low byte of the branch target. If the W register is not the same as the comparison value, then the program counter is set to equal the branch target. Otherwise, the program counter is set to equal the fourth block of memory after the opcode.
0xFF	Halt opcode Execution stops and the PC is not incremented.

Appendix C - Rotamola 34HC22

Features

- Big endian, 2 byte memory addressing, 1 byte opcodes
- 512 bytes of memory
- Special Purpose byte-sized registers called A and B

Reset

Upon reset, the program counter (PC) is set to address 509_{10} (top of memory - 2). Memory is not initialised. Registers A and B remain in their previous state.

Instruction Set Summary

Opcode	Action
0x0C	Move A to memory The first byte after the opcode represents the high byte of the memory address. The second byte after the opcode represents the low byte of the memory address. The memory address can be determined by (high byte << 8) | low byte. The contents of the A register are moved to this memory address. The PC is set to the third byte after the opcode.
0x37	Load A with Value The first byte after the opcode is the value to load into the A register. The PC is set to the second byte after the opcode.
0x38	Load B with Value The first byte after the opcode is the value to load into the B register. The PC is set to the second byte after the opcode.
0x53	Increment Register A The value of register A is incremented, and the result is stored back into A. The value of the PC is incremented, so it points to the next byte after the opcode.
0x5A	Branch Always The first byte after the opcode represents the high byte of the memory address. The second byte after the opcode represents the low byte of the memory address. The PC is set to this memory address.
0x5B	Branch if A < B The first byte after the opcode represents the high byte of the memory address. The second byte after the opcode represents the low byte of the memory address. If the content of A is less than B, the PC is set to this memory address. Otherwise, the PC is set to the third byte after the opcode.
0x5D	Branch if Less than A The first byte after the opcode represents the comparison value. The second byte after the opcode represents the high byte of the memory address. The third byte represents the low byte of the memory address. If the comparison value is less than the value of the A register, the PC is set to this memory address. Otherwise, the PC is set to the fourth byte after the opcode.
0x64	Halt opcode Execution stops and the PC is not incremented.